Efficient Simulation of Physical System Models Using Inlined Implicit Runge–Kutta Algorithms

by

Vicha Treeaporn

A Thesis Submitted to the Faculty of the

Department of Electrical and Computer Engineering

In Partial Fulfillment of the Requirements
For the Degree of

Master of Science

In the Graduate College

The University of Arizona

2 0 0 5

Statement by Author

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Signed: _____

Approval by Thesis Director

This thesis has been approved on the date shown below:

_____	_____
Dr. François E. Cellier Professor of Electrical and Computer Engineering	Date

Acknowledgments

I am grateful to my advisor Professor François E. Cellier for his guidance, encouragement, and patience. I am fortunate to have worked with such an extremely knowledgeable individual who is always available and always ready to lend a helping hand. His classes are always challenging and fulfilling, and the wealth of knowledge gained is invaluable. Every class has a common theme starting with the initial understanding of the problem at hand, to the problem solving methodology and finally to the application of a solution.

To my parents.

Table of Contents

Table of Contents—*Continued*

List of Figures

List of Figures—*Continued*

List of Tables

Abstract

Stiff systems commonly occur in science and engineering, and the use of an implicit integration algorithm is typically needed to simulate them. As model complexity increases, the need for efficient ways to solve these types of systems is becoming of increasing importance. Using a technique called inline–integration with implicit Runge–Kutta (IRK) algorithms and tearing may lead to a more efficient simulation. To further increase the efficiency of the simulation, the step–size of the integration algorithm can be controlled. By using larger integration steps when allowable, the simulation can progress with fewer computations while still maintaining the desired accuracy.

In this thesis, for the purpose of step–size control, two new embedding methods will be proposed. The first will be for HW-SDIRK(3)4, a singly diagonally implicit Runge–Kutta (SDIRK) algorithm and the second for Lobatto IIIC(6). These two embedding methods will then be compared with those previously found for the Radau IIA family and Lobatto IIIC(4), all fully–implicit Runge–Kutta algorithms.

Efficient Simulation of Physical System Models Using Inlined Implicit Runge–Kutta Algorithms

Vicha Treeaporn, M.S.
The University of Arizona, 2005

Director: Dr. François E. Cellier

Stiff systems commonly occur in science and engineering, and the use of an implicit integration algorithm is typically needed to simulate them. As model complexity increases, the need for efficient ways to solve these types of systems is becoming of increasing importance. Using a technique called inline–integration with implicit Runge–Kutta (IRK) algorithms and tearing may lead to a more efficient simulation. To further increase the efficiency of the simulation, the step–size of the integration algorithm can be controlled. By using larger integration steps when allowable, the simulation can progress with fewer computations while still maintaining the desired accuracy.

In this thesis, for the purpose of step–size control, two new embedding methods will be proposed. The first will be for HW-SDIRK(3)4, a singly diagonally implicit Runge–Kutta (SDIRK) algorithm and the second for Lobatto IIIC(6). These two embedding methods will then be compared with those previously found for the Radau IIA family and Lobatto IIIC(4), all fully–implicit Runge–Kutta algorithms.

Chapter 1

Preliminaries

1.1 Model Representations

A mathematical model is a representation of a system in which experiments can be performed on that model to learn about the system [1]. One way a physical system can be represented is by a system of Ordinary Differential Equations (ODEs). This type of model description is called a state–space model and can be expressed by:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, t) \tag{1.1}$$
$$\mathbf{x}(t = t_0) = \mathbf{x_0}$$

where $\mathbf{x}$ is the state vector, $\mathbf{u}$ is the input vector, and t denotes the time variable. Another representation is by Differential Algebraic Equations (DAEs) and can be written in general form as:

$$0 = \mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{u}, t) \tag{1.2}$$
$$\mathbf{x}(t = t_0) = \mathbf{x_0}$$

where $\mathbf{x}$ is a vector of unknown variables that can also appear in differentiated form, $\mathbf{u}$ is the input vector, and t is the time variable.

1.2 Solution by Numerical Methods

There are two different schemes of numerical integration, *implicit methods* and *explicit methods*. Explicit integration algorithms only depend on past values of state variables and state derivatives to compute the next state. On the other hand, implicit integration algorithms, depend on both past and current values of the state variables and state derivatives. Explicit methods have the advantage over implicit methods

in that they are relatively easy to implement. The simplest of these is the Forward Euler (FE) algorithm:

$$x(t+h) \approx x(t) + \dot{x}(t) \cdot h \tag{1.3}$$

where h is the step size. From Eq.(1.3), it can be seen that it is just a matter of procedural computation to compute the next step from the current step. This is true in general for all explicit algorithms. This relative ease, however, comes at a price. In order to compute the next step, an explicit integration may need an excessively small step size to maintain numerical stability if solving a *stiff* system. Compared with explicit techniques, implicit techniques may have better numerical properties but have additional computational load. This additional load comes from the need to simultaneously solve a non–linear set of equations during every step. Even with this additional computational load, implicit methods may allow the use of larger step sizes when solving stiff systems. The simplest implicit integration method is the Backward Euler (BE) algorithm:

$$x(t+h) \approx x(t) + \dot{x}(t+h) \cdot h \tag{1.4}$$

From this equation it can be seen that additional computational load comes in the form of a nonlinear *algebraic loop*. The problem is that before $x(t+h)$ can be computed, $\dot{x}(t+h)$ must be known, but in order to compute $\dot{x}(t+h)$ the value of $x(t+h)$ must be known.

Both the FE and BE algorithms are *single–step* methods since when moving the solution from time t to $t+h$, neither method uses values from any previous time instants $t-h$, $t-2h$, etc. Integration algorithms that use values from multiple time instants are referred to as *multi–step* algorithms. The Adams–Bashforth–Moulton algorithms and the widely used Backward Difference Formulae (BDF) are examples of multi–step methods. Much work has been done to solve these systems by numerical methods. To evaluate the performance of these methods, benchmark test problems can be used.

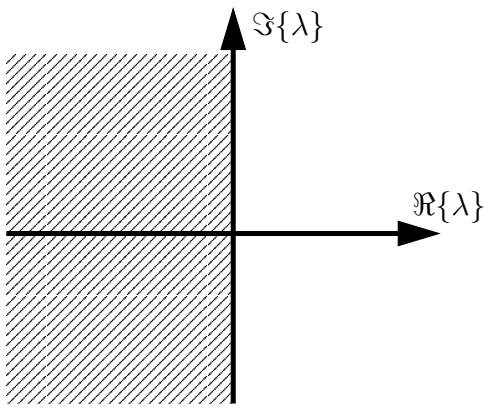

FIGURE 1.1. Domain of analytical stability

1.3 Numerical Stability

The numerical stability of a solver is evaluated using a homogeneous time–invariant linear test problem:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} \tag{1.5}$$
$$\mathbf{x}(t = t_0) = \mathbf{x_0}$$

This system can be solved analytically with any solution given by:

$$\mathbf{x}(t) = \exp(\mathbf{A}t) \cdot \mathbf{x_0} \tag{1.6}$$

and if all of the trajectories (orbits) stay bounded as $t \to \infty$, then the solution is *analytically stable*. If all of the eigenvalues λ of $\mathbf{A}$ have negative real parts then the system is analytically stable. The domain of analytical stability is shown in Figure 1.1 where the stable region is shaded. In order to find the numerical stability, the integration algorithm is applied to the linear test problem. For example, the numerical stability domain of BE can be found by first creating an equivalent discrete–time system. Substituting Eq.(1.5) into Eq.(1.4) results in:

$$\mathbf{x}(t + h) = \mathbf{x}(t) + \mathbf{A}\mathbf{x}(t + h) \cdot h \tag{1.7}$$

or written more compactly as:

$$\mathbf{x_{k+1}} = [\mathbf{I} - \mathbf{A}h]^{-1}\mathbf{x_k} \tag{1.8}$$

where $\mathbf{I}$ is the identity matrix and $\mathbf{k}$, by indexing the simulation time is the $\mathbf{k}^{\text{th}}$ integration step. In this new representation $\mathbf{x}(t)$ corresponds to $\mathbf{x_k}$ and the value at the next time step $\mathbf{x}(t+h)$ corresponds to $\mathbf{x_{k+1}}$. The discrete–time system of BE can then be expressed with a discrete state matrix $\mathbf{F}$ as:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} \tag{1.9}$$

$$\mathbf{F} = [\mathbf{I} - \mathbf{A}h]^{-1} \tag{1.10}$$

If and only if all of the eigenvalues of $\mathbf{F}$ are inside the unit circle, then the system is numerically stable. Shown in Figure 1.2 is the numerical stability domain for BE. The region inside the circle is the unstable region and everything outside is the stable region. The domain of numerical stability tries to approximate the domain of analytical stability and the BE algorithm doesn't do very well. Comparing Figure 1.1 and Figure 1.2, it can be seen that the analytically unstable region is for the most part numerically stable when using BE. From [2], a method is called *A–stable* or absolute stable if it contains the entire left half of the $\lambda \cdot h$–plane as part of its numerical stability domain. The BE algorithm is very stable, thus it is an A–stable method. A method is called *L–stable* if it is A–stable and has damping properties such that as the eigenvalues approach negative infinity, the damping approaches infinity.

1.4 Stiffness

Many of the interesting problems in science and engineering lead to equations that are *stiff*. A rough idea of stiffness is that somehow the computation of the numerical solution to these equations is in some sense *ill–conditioned* over the range of integration.

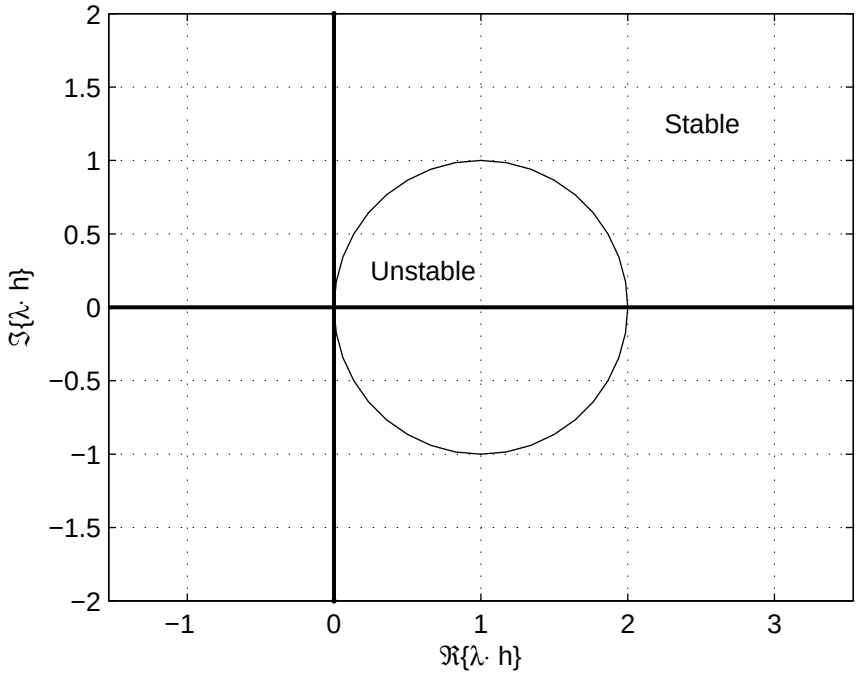

FIGURE 1.2. Stability domain of BE

An example from [3] illustrates this. The following:

$$\dot{x} = -10^3 \left(x - \exp\left(-t\right)\right) - \left(\exp\left(-t\right)\right) \qquad 0 \leq t \leq t_f \tag{1.11}$$
$$x(0) = 0$$

is a prototypical stiff equation. Shown in Figure 1.3 is the solution for $t_f = 1$. At first glance, the solution seems to behave perfectly fine, but upon closer inspection there is a transient region for small t that can be better seen in Figure 1.4. The analytical solution of (1.11) is given by:

$$x(t) = \exp(-t) - \exp(-10^3 t) \tag{1.12}$$

From this equation it can be seen that there is a fast component given by $\exp(-10^3 t)$ and a slow component given by $\exp(-t)$. After a short time the solution looks like the dominant term $\exp(-t)$ since the fast component has vanished. From the point of view of an explicit integration algorithm, even though the fast component has de-

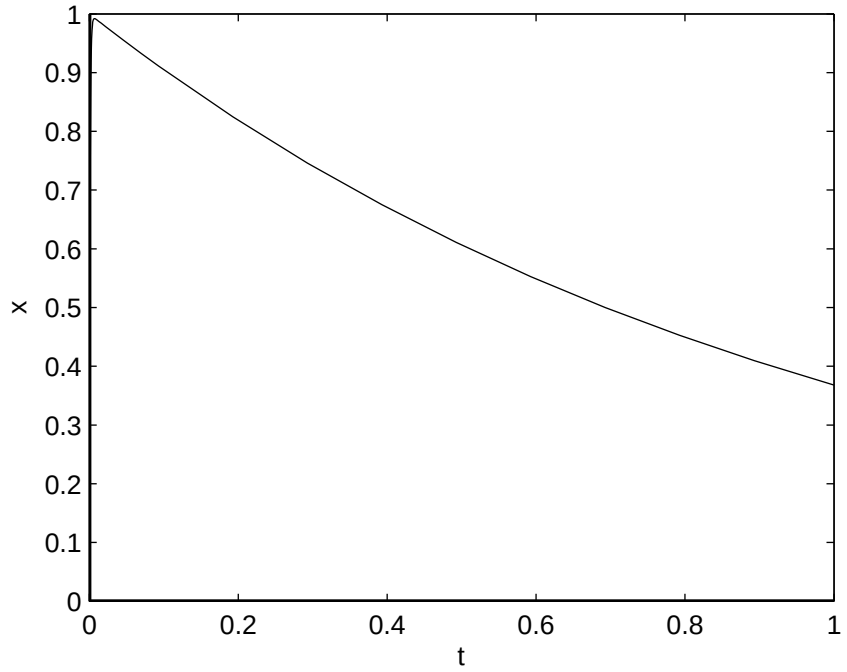

FIGURE 1.3. Stiff ODE on $0 \leq t \leq 1$

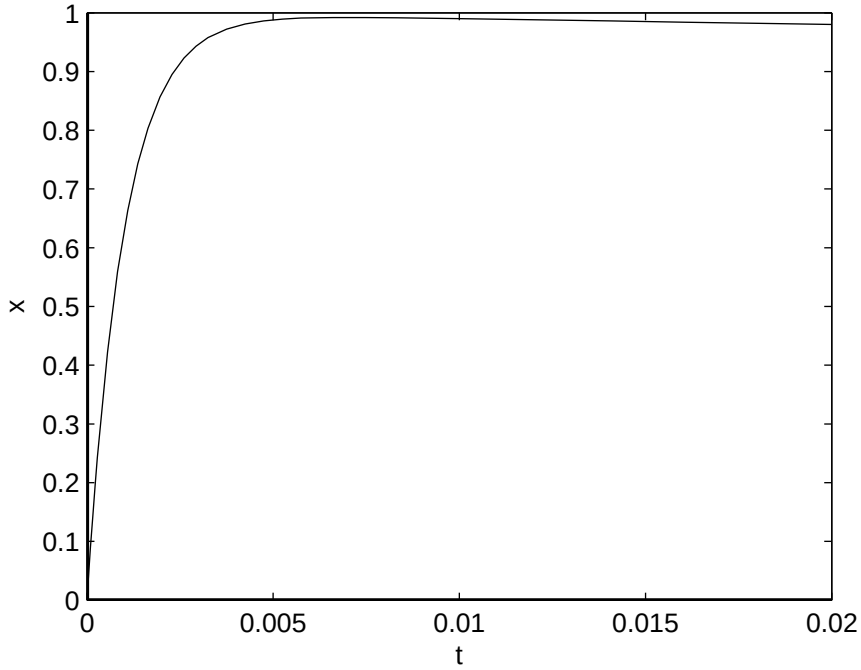

FIGURE 1.4. Stiff ODE on $0 \leq t \leq 0.02$

cayed after the transient region, the step size must still be kept small. This is what makes the problem stiff. Thus, if t_f limited to the transient region, then the problem is not considered stiff. There are various definitions of stiffness because several phenomena can occur [4]. In general, stiff equations result from sets of equations that have eigenvalues that vary greatly in magnitude such that there are both slow and fast modes. The solution then behaves like it exists in multiple regions of different scales. Although another attempt to better quantify stiffness has been made in [4], the definition found in [2] is sufficient:

Definition 1.4.1. A system is called stiff if, when integrated with any explicit RKn algorithm and a local error tolerance of 10^{-n}, the step size of the algorithm is forced down to below a value indicated by the local error estimate due to constraints imposed on it by the limited size of the numerically stable region.

Stiff problems occur commonly and implicit integration algorithms are better suited to solve these types of problems. Since implicit methods may have better numerical properties, larger step sizes can be used leading to a more efficient simulation.

1.5 Approximation Errors

When a model is simulated, errors can occur in various places. The first type of error occurs because only a finite number of terms of the Taylor series are used and is called the *truncation error*. A second source of error comes from the use of a digital computer to perform a simulation. A computer can only represent numbers with finite precision so this type of error is called *roundoff error*. With the combination of truncation and roundoff errors, the computed value of the trajectory at the next point in time cannot be exact. Over multiple integration steps this error propagates as an error in the initial conditions from one step to the next. This type of error is called *accumulation error*.

TABLE 1.1. Generalized Butcher Tableau

$\mathbf{c}$	$\mathbf{A}$
x	$\mathbf{b}'$

It should be noted that there can also be errors in the model itself. The system may have dynamics that are not included in the model or the model may not represent the physical system accurately. The former type of errors are *structural model errors* and the latter are *parametric model errors*. Both types of errors do not affect the numerical integration method since they are model errors and are dealt with through *model validation* discussed in [5].

1.6 Fully–Implicit Runge–Kutta Algorithms

Problems of interest will primarily occur in DAE form and will invariably be stiff. Therefore, implicit algorithms must be used for their simulation. One class of implicit integration algorithms that may lead to efficient implementations together with a technique called *inline–integration* are the implicit Runge–Kutta algorithms. Runge–Kutta algorithms are single–step methods but compute intermediate values at various time instants within the step called predictors. Each time instant when a predictor is computed is called a stage. At the end of the integration step, some combination of those predictions, called the corrector, is used to compute the state vector.

Runge–Kutta methods can be represented in a compact form shown in Table 1.1 called a *Butcher Tableau* [6]. In the generalized Butcher Tableau, the $\mathbf{c}$–vector denotes the time instant when each stage is evaluated. The $\mathbf{b}'$–vector denotes the weights for the corrector stage, and the $\mathbf{A}$–matrix contains the weights for the predictor stages. The Butcher Tableau of an m–stage algorithm with $\mathbf{b}', \mathbf{c} \in \mathbb{R}^m$ and $\mathbf{A} \in \mathbb{R}^{m \times m}$ has

the property $c_i = \sum_{j=1}^{m} a_{ij}$ for $i = 1, \ldots, m$ and can be expanded into:

$$
\begin{aligned}
1^{\text{st}} \text{ stage:} \quad & \mathbf{x^{P_1}} = \mathbf{x_k} + \sum_j a_{1j} h \dot{\mathbf{x}}^{\mathbf{P_j}} \\
& \dot{\mathbf{x}}^{\mathbf{P_1}} = \mathbf{f}(\mathbf{x^{P_1}}, t_{k+c_1}) \\
2^{\text{nd}} \text{ stage:} \quad & \mathbf{x^{P_2}} = \mathbf{x_k} + \sum_j a_{2j} h \dot{\mathbf{x}}^{\mathbf{P_j}} \\
& \dot{\mathbf{x}}^{\mathbf{P_2}} = \mathbf{f}(\mathbf{x^{P_2}}, t_{k+c_2}) \\
\vdots \quad & \qquad \vdots \\
m^{\text{th}} \text{ stage:} \quad & \mathbf{x^{P_m}} = \mathbf{x_k} + \sum_j a_{mj} h \dot{\mathbf{x}}^{\mathbf{P_j}} \\
& \dot{\mathbf{x}}^{\mathbf{P_m}} = \mathbf{f}(\mathbf{x^{P_m}}, t_{k+c_m}) \\
\text{corrector:} \quad & \mathbf{x_{k+1}} = \mathbf{x_k} + \sum_j b_j h \dot{\mathbf{x}}_{\mathbf{k}}^{\mathbf{P_j}}
\end{aligned}
$$

For IRKs the resulting predictor equations are coupled together and must be solved simultaneously.

1.6.1 Radau IIA

A family of very compact fully–implicit Runge–Kutta algorithms are the Radau IIA algorithms. The third order accurate technique, Rad3, uses only two stages, while only three stages are needed in the fifth order accurate method, Rad5. The Rad3 algorithm is defined by the Butcher Tableau shown in Table 1.2. Although the Rad5 algorithm is fifth–order accurate, both irrational coefficients and time steps may make this method less attractive. The Butcher Tableau shown in Table 1.3 describes the Rad5 algorithm. Since the second and third stages of Rad3 and Rad5, respectively, are evaluated at the end of the step, the corrector equations collapse with the respective equations that describe those stages.

1.6.2 Lobatto IIIC

Other fully–implicit Runge–Kutta algorithms are the Lobatto IIIC methods. The commonly used fourth order accurate method requires three stages and is described by the Butcher Tableau shown in Table 1.4. The sixth order accurate method uses four stages and is analyzed in a later chapter. Like the Rad5 algorithm, the third stage of this algorithm is evaluated at the end of the integration step. Thus, the corrector equation also collapse with the equation that describes the third stage.

1.6.3 HW-SDIRK

One interesting algorithm is HW-SDIRK(3)4, not fully–implicit, but a diagonally implicit Runge–Kutta. This algorithm is 4^{th}–order accurate using five stages. Also part of this algorithm is a 3^{rd}–order accurate embedding method that can be used for step size control. Unlike Rad5, both methods contained in HW-SDIRK have rational coefficients. What makes this algorithm interesting is that when used with inlining, each stage of this algorithm can be iterated on separately. This algorithm will be analyzed in a later chapter.

TABLE 1.2. Radau IIA(3)

1/3	5/12	−1/12
1	3/4	1/4
x	3/4	1/4

TABLE 1.3. Radau IIA(5)

$\frac{4-\sqrt{6}}{10}$	$\frac{88-7\sqrt{6}}{360}$	$\frac{296-169\sqrt{6}}{1800}$	$\frac{-2+3\sqrt{6}}{225}$
$\frac{4+\sqrt{6}}{10}$	$\frac{296+169\sqrt{6}}{1800}$	$\frac{88+7\sqrt{6}}{360}$	$\frac{-2-3\sqrt{6}}{225}$
1	$\frac{16-\sqrt{6}}{36}$	$\frac{16+\sqrt{6}}{36}$	$\frac{1}{9}$
x	$\frac{16-\sqrt{6}}{36}$	$\frac{16+\sqrt{6}}{36}$	$\frac{1}{9}$

TABLE 1.4. Lobatto IIIC(4)

0	1/6	−1/3	1/6
1/2	1/6	5/12	−1/12
1	1/6	2/3	1/6
x	1/6	2/3	1/6

Chapter 2

Background

Traditionally, the model to be simulated was kept separate from the simulation engine. This was done since, at that time, the idea of generating a model was something that a modeler did, whereas performing experiments on a model or simulating a model was something that was done on a computer. Since these were two separate things, it could happen that the model that was created may not lend itself to being implemented very easily for simulation.

The simulation engines in use typically required first–order ordinary differential equations (ODEs), a state–space representation, as the model description. For simple models this may not be a problem, but as model complexity increases, it may be more convenient to use an equivalent differential algebraic equation (DAE) model representation.

2.1 Differential Algebraic Equations

Physical systems are not easily represented by state–space models since the equations describing a physical system are typically algebraically coupled. Because of this, physical systems are usually described by a mixture of differential and algebraic equations called *Differential Algebraic Equations (DAEs)*. This can, for example, be seen in the following set of equations that can describe the simple circuit shown in Figure 2.1:

$$u_0 = f(t) \tag{2.1}$$

$$u_1 = R_1 \cdot i_1 \tag{2.2}$$

$$u_2 = R_2 \cdot i_2 \tag{2.3}$$

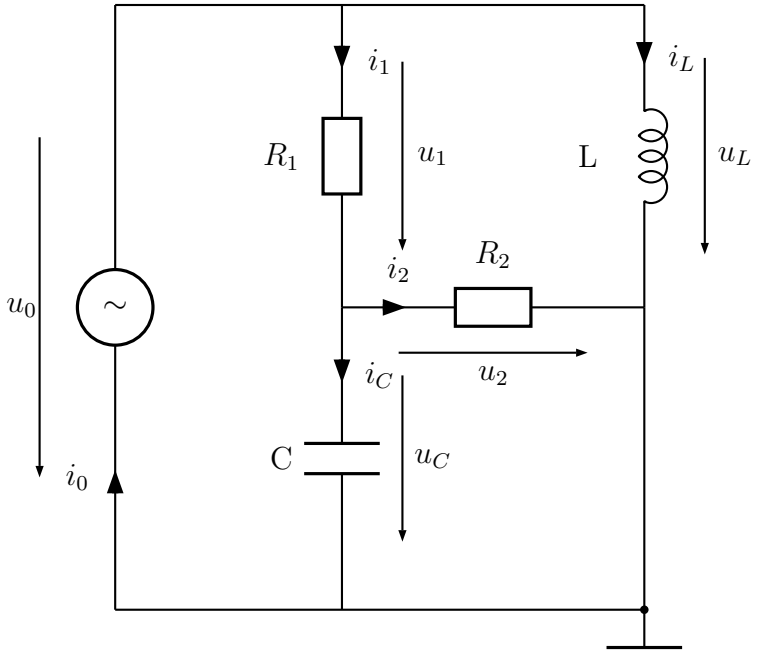

FIGURE 2.1. Simple RLC Circuit

$$u_L = L \cdot \frac{di_L}{dt} \tag{2.4}$$

$$i_C = C \cdot \frac{du_C}{dt} \tag{2.5}$$

$$u_0 = u_1 + u_C \tag{2.6}$$

$$u_L = u_1 + u_2 \tag{2.7}$$

$$u_C = u_2 \tag{2.8}$$

$$i_0 = i_1 + i_L \tag{2.9}$$

$$i_1 = i_2 + i_C \tag{2.10}$$

These equations can also be represented in the form of a *structure incidence matrix* [2]. Each row of the structure incidence matrix $\mathbf{S}$ represents an equation and each column represents a variable. When the i^{th} equation (row) contains the j^{th} variable (column), then the element $\mathbf{S}_{i,j}$ is set to 1, otherwise it is set to 0. With the columns

ordered by $u_0, i_0, u_1, i_1, u_2, i_2, u_L, \frac{di_L}{dt}, \frac{du_C}{dt}$, and i_C, the structure incidence matrix of this circuit can be written as:

$$\mathbf{S_{RLC}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.11}$$

The complexity of these equations is denoted by the *index* of the system. An *index–0* system has a structure incidence matrix that is lower triangular after sorting. This means that the system doesn't have any algebraic loops or structural singularities. An *index–1* system doesn't have structural singularities but does have algebraic loops. The structure incidence matrix for this case is block lower triangular after sorting. Systems that have an index > 1 have structural singularities and are *higher–index* problems.

2.2 Inline–Integration

As noted in §1.2 it is straightforward to solve Eq.(1.1) using an explicit integration algorithm. However, when the system to be solved is stiff it is more appropriate to use an implicit integration algorithm. The problem is that implicit algorithms lead to nonlinear equations that, in general, need to be solved by a Newton iteration. By taking advantage of the structure of the model equations and the integration algorithm, the efficiency of the simulation can be improved [7]. This can by accomplished by a technique called *inline integration*. Using inline integration, an implicit integration algorithm can be merged with the model. In this way the differential equations are eliminated and the resulting model becomes a set of difference equations (ΔEs)[7].

Using implicit Runge–Kutta algorithms together with inline integration and tearing [8] may lead to efficient implementations. This is desirable when a sufficiently large and complex system needs to be simulated, for instance, in real–time. This is the case since only a Newton iteration needs to be performed on the set of ΔEs at each time step. To illustrate the process of inline integration, the circuit shown in Figure 2.1 will be inlined with three different IRKs.

2.2.1 Radau IIA

By inlining the Rad3 algorithm with the model of the circuit shown in Figure 2.1, the set of DAEs given by Eqs.(2.1)–(2.10) is replicated once for each stage of the integration algorithm and Rad3 integrator equations are added for each of the derivatives. From Table 1.2, the integrator equations can be expressed as:

$$\mathbf{x_{k+\frac{1}{3}}} = \mathbf{x_k} + \frac{5h}{12} \cdot \dot{\mathbf{x}}_{\mathbf{k+\frac{1}{3}}} - \frac{h}{12} \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{2.12}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{3h}{4} \cdot \dot{\mathbf{x}}_{\mathbf{k+\frac{1}{3}}} + \frac{h}{4} \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{2.13}$$

and inlining the circuit model with Rad3 results in the following equations:

$$v_0 = f\left(t_k + \frac{h}{3}\right) \tag{2.14}$$

$$v_1 = R_1 \cdot j_1 \tag{2.15}$$

$$v_2 = R_2 \cdot j_2 \tag{2.16}$$

$$v_L = L \cdot dj_L \tag{2.17}$$

$$j_C = C \cdot dv_C \tag{2.18}$$

$$v_0 = v_1 + v_C \tag{2.19}$$

$$v_L = v_1 + v_2 \tag{2.20}$$

$$v_C = v_2 \tag{2.21}$$

$$j_0 = j_1 + j_L \tag{2.22}$$

$$j_1 = j_2 + j_C \tag{2.23}$$

$$u_0 = f(t_k + h) \tag{2.24}$$

$$u_1 = R_1 \cdot i_1 \tag{2.25}$$

$$u_2 = R_2 \cdot i_2 \tag{2.26}$$

$$u_L = L \cdot di_L \tag{2.27}$$

$$i_C = C \cdot du_C \tag{2.28}$$

$$u_0 = u_1 + u_C \tag{2.29}$$

$$u_L = u_1 + u_2 \tag{2.30}$$

$$u_C = u_2 \tag{2.31}$$

$$i_0 = i_1 + i_L \tag{2.32}$$

$$i_1 = i_2 + i_C \tag{2.33}$$

$$j_L = i_{L_{k-1}} + \frac{5h}{12} \cdot dj_L - \frac{5h}{12} \cdot di_L \tag{2.34}$$

$$i_L = i_{L_{k-1}} + \frac{3h}{4} \cdot dj_L + \frac{h}{4} \cdot di_L \tag{2.35}$$

$$v_C = u_{C_{k-1}} + \frac{5h}{12} \cdot dv_C - \frac{5h}{12} \cdot du_C \tag{2.36}$$

$$u_C = u_{C_{k-1}} + \frac{3h}{4} \cdot dv_C + \frac{h}{4} \cdot du_C \tag{2.37}$$

where $i_{L_{k-1}}$ and $u_{C_{k-1}}$ are the state values computed during the previous time step. In this case, the time derivatives present in Eqs.(2.4) and (2.5) have been eliminated in the resulting 24 equations in 24 unknowns. Since the two stages of Rad3 are coupled together, all 24 equations of this ΔE system must be solved simultaneously.

Similarly, the set of equations for the same circuit inlined with the Rad5 algorithm becomes:

$$w_0 = f\left(t + \frac{4 - \sqrt{6}}{10} \cdot h\right) \tag{2.38}$$

$$w_1 = R_1 \cdot m_1 \tag{2.39}$$

$$w_2 = R_2 \cdot m_2 \tag{2.40}$$

$$w_L = L \cdot dm_L \tag{2.41}$$

$$m_C = C \cdot dw_C \tag{2.42}$$

$$w_0 = w_1 + w_C \tag{2.43}$$

$$w_L = w_1 + w_2 \tag{2.44}$$

$$w_C = w_2 \tag{2.45}$$

$$m_0 = m_1 + m_L \tag{2.46}$$

$$m_1 = m_2 + m_C \tag{2.47}$$

$$v_0 = f\left(t + \frac{4+\sqrt{6}}{10} \cdot h\right) \tag{2.48}$$

$$v_1 = R_1 \cdot j_1 \tag{2.49}$$

$$v_2 = R_2 \cdot j_2 \tag{2.50}$$

$$v_L = L \cdot dj_L \tag{2.51}$$

$$j_C = C \cdot dv_C \tag{2.52}$$

$$v_0 = v_1 + v_C \tag{2.53}$$

$$v_L = v_1 + v_2 \tag{2.54}$$

$$v_C = v_2 \tag{2.55}$$

$$j_0 = j_1 + j_L \tag{2.56}$$

$$j_1 = j_2 + j_C \tag{2.57}$$

$$u_0 = f(t+h) \tag{2.58}$$

$$u_1 = R_1 \cdot i_1 \tag{2.59}$$

$$u_2 = R_2 \cdot i_2 \tag{2.60}$$

$$u_L = L \cdot di_L \tag{2.61}$$

$$i_C = C \cdot du_C \tag{2.62}$$

$$u_0 = u_1 + u_C \tag{2.63}$$

$$u_L = u_1 + u_2 \tag{2.64}$$

$$u_C = u_2 \tag{2.65}$$

$$i_0 = i_1 + i_L \tag{2.66}$$

$$i_1 = i_2 + i_C \tag{2.67}$$

$$\begin{aligned} m_L = \operatorname{pre}(i_L) &+ \frac{88 - 7\sqrt{6}}{360} \cdot h \cdot dm_L \\ &+ \frac{296 - 169\sqrt{6}}{1800} \cdot h \cdot dj_L + \frac{-2 + 3\sqrt{6}}{225} \cdot h \cdot di_L \end{aligned} \tag{2.68}$$

$$\begin{aligned} j_L = \operatorname{pre}(i_L) &+ \frac{296 + 169\sqrt{6}}{1800} \cdot h \cdot dm_L \\ &+ \frac{88 + 7\sqrt{6}}{360} \cdot h \cdot dj_L + \frac{-2 - 3\sqrt{6}}{225} \cdot h \cdot di_L \end{aligned} \tag{2.69}$$

$$\begin{aligned} i_L = \operatorname{pre}(i_L) &+ \frac{16 - \sqrt{6}}{36} \cdot h \cdot dm_L \\ &+ \frac{16 + \sqrt{6}}{36} \cdot h \cdot dj_L + \frac{1}{9} \cdot h \cdot di_L \end{aligned} \tag{2.70}$$

$$\begin{aligned} w_C = \operatorname{pre}(u_C) &+ \frac{88 - 7\sqrt{6}}{360} \cdot h \cdot dw_C \\ &+ \frac{296 - 169\sqrt{6}}{1800} \cdot h \cdot dv_C + \frac{-2 + 3\sqrt{6}}{225} \cdot h \cdot du_C \end{aligned} \tag{2.71}$$

$$\begin{aligned} v_C = \operatorname{pre}(u_C) &+ \frac{296 + 169\sqrt{6}}{1800} \cdot h \cdot dw_C \\ &+ \frac{88 + 7\sqrt{6}}{360} \cdot h \cdot dv_C + \frac{-2 - 3\sqrt{6}}{225} \cdot h \cdot du_C \end{aligned} \tag{2.72}$$

$$\begin{aligned} u_C = \operatorname{pre}(u_C) &+ \frac{16 - \sqrt{6}}{36} \cdot h \cdot dw_C \\ &+ \frac{16 + \sqrt{6}}{36} \cdot h \cdot dv_C + \frac{1}{9} \cdot h \cdot du_C \end{aligned} \tag{2.73}$$

where pre(·) denotes the value computed for a variable during the previous time step. After inlining the original model of 10 equations with Rad5, there are now 36 equations in 36 unknowns that must be solved simultaneously. From this set of equations it can be seen that irrational coefficients appear in a number of different places. The augmented incidence matrix for one stage of Rad5 inlined equations can be written as:

$$\mathbf{S} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{2.74}$$

where the columns are now ordered by $u_0, i_0, u_1, i_1, u_2, i_2, u_L, di_L, du_C, i_C, i_L$, and u_C. Together with the incidence matrix for the coupling between the different stages:

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \tag{2.75}$$

the structure incidence matrix for the Rad5 set of inlined equations can be written as:

$$\mathbf{S_{rad5}} = \begin{pmatrix} \mathbf{S} & \mathbf{C} & \mathbf{C} \\ \mathbf{C} & \mathbf{S} & \mathbf{C} \\ \mathbf{C} & \mathbf{C} & \mathbf{S} \end{pmatrix}. \tag{2.76}$$

2.2.2 Lobatto IIIC(4)

Finally, the same circuit inlined with Lobatto IIIC(4) result in the following equations:

$$w_0 = f(t) \tag{2.77}$$
$$w_1 = R_1 \cdot m_1 \tag{2.78}$$
$$w_2 = R_2 \cdot m_2 \tag{2.79}$$
$$w_L = L \cdot dm_L \tag{2.80}$$
$$m_C = C \cdot dw_C \tag{2.81}$$
$$w_0 = w_1 + w_C \tag{2.82}$$
$$w_L = w_1 + w_2 \tag{2.83}$$
$$w_C = w_2 \tag{2.84}$$
$$m_0 = m_1 + m_L \tag{2.85}$$
$$m_1 = m_2 + m_C \tag{2.86}$$

$$v_0 = f\left(t + \frac{h}{2}\right) \tag{2.87}$$
$$v_1 = R_1 \cdot j_1 \tag{2.88}$$
$$v_2 = R_2 \cdot j_2 \tag{2.89}$$
$$v_L = L \cdot dj_L \tag{2.90}$$
$$j_C = C \cdot dv_C \tag{2.91}$$
$$v_0 = v_1 + v_C \tag{2.92}$$
$$v_L = v_1 + v_2 \tag{2.93}$$
$$v_C = v_2 \tag{2.94}$$
$$j_0 = j_1 + j_L \tag{2.95}$$
$$j_1 = j_2 + j_C \tag{2.96}$$

$$u_0 = f(t+h) \tag{2.97}$$

$$u_1 = R_1 \cdot i_1 \tag{2.98}$$

$$u_2 = R_2 \cdot i_2 \tag{2.99}$$

$$u_L = L \cdot di_L \tag{2.100}$$

$$i_C = C \cdot du_C \tag{2.101}$$

$$u_0 = u_1 + u_C \tag{2.102}$$

$$u_L = u_1 + u_2 \tag{2.103}$$

$$u_C = u_2 \tag{2.104}$$

$$i_0 = i_1 + i_L \tag{2.105}$$

$$i_1 = i_2 + i_C \tag{2.106}$$

$$m_L = \mathrm{pre}(i_L) + \frac{h}{6} \cdot dm_L - \frac{h}{3} \cdot dj_L + \frac{h}{6} \cdot di_L \tag{2.107}$$

$$j_L = \mathrm{pre}(i_L) + \frac{h}{6} \cdot dm_L + \frac{5h}{12} \cdot dj_L - \frac{h}{12} \cdot di_L \tag{2.108}$$

$$i_L = \mathrm{pre}(i_L) + \frac{h}{6} \cdot dm_L + \frac{2h}{3} \cdot dj_L + \frac{h}{6} \cdot di_L \tag{2.109}$$

$$w_C = \mathrm{pre}(u_C) + \frac{h}{6} \cdot dw_C - \frac{h}{3} \cdot dv_C + \frac{h}{6} \cdot du_C \tag{2.110}$$

$$v_C = \mathrm{pre}(u_C) + \frac{h}{6} \cdot dw_C + \frac{5h}{12} \cdot dv_C - \frac{h}{12} \cdot du_C \tag{2.111}$$

$$u_C = \mathrm{pre}(u_C) + \frac{h}{6} \cdot dw_C + \frac{2h}{3} \cdot dv_C + \frac{h}{6} \cdot du_C \tag{2.112}$$

Compared with Rad5, the only differences in the equations are the time instants when the equations are evaluated and the coefficients for the integration algorithm. Since the equations are otherwise the same, both Rad5 and Lobatto IIIC(4) have the same structure incidence matrix. The procedure is similar for inlining with any integration algorithm.

2.3 Sorting

Before the inlined model can be simulated the equations must first be sorted and causalized. For an *acausal* equation, the equal sign is used in the sense of equality. Looking at Eqs.(2.77) and (2.82), both have the same variable, w_0, on the left hand side of the equal sign, but only one of the equations can be used to compute w_0. The remaining equation must be used to solve for another variable. The variable that was selected to be solved for in a particular equation can by marked by enclosing it in square brackets $[\cdot]$. Since that variable is now computed it can be considered as known in the remaining equations and is denoted by underlining. Unfortunately, inlining with an implicit integration algorithm leads to algebraic loops. Thus, the equations cannot be completely sorted and using the Rad3 example above, the partially sorted equations are:

$$[v_0] = f\left(t_k + \frac{h}{3}\right) \tag{2.113}$$

$$v_1 = R_1 \cdot j_1 \tag{2.114}$$

$$v_2 = R_2 \cdot j_2 \tag{2.115}$$

$$\underline{v_L} = L \cdot [dj_L] \tag{2.116}$$

$$j_C = C \cdot dv_C \tag{2.117}$$

$$\underline{v_0} = v_1 + v_C \tag{2.118}$$

$$[v_L] = v_1 + v_2 \tag{2.119}$$

$$v_C = v_2 \tag{2.120}$$

$$[j_0] = j_1 + \underline{j_L} \tag{2.121}$$

$$j_1 = j_2 + j_C \tag{2.122}$$

$$[u_0] = f(t_k + h) \tag{2.123}$$

$$u_1 = R_1 \cdot i_1 \tag{2.124}$$

$$u_2 = R_2 \cdot i_2 \tag{2.125}$$

$$\underline{u_L} = L \cdot [di_L] \tag{2.126}$$

$$i_C = C \cdot du_C \tag{2.127}$$

$$\underline{u_0} = u_1 + u_C \tag{2.128}$$

$$[u_L] = u_1 + u_2 \tag{2.129}$$

$$u_C = u_2 \tag{2.130}$$

$$[i_0] = i_1 + \underline{i_L} \tag{2.131}$$

$$i_1 = i_2 + i_C \tag{2.132}$$

$$[j_L] = i_{L_{k-1}} + \frac{5h}{12} \cdot \underline{dj_L} - \frac{5h}{12} \cdot \underline{di_L} \tag{2.133}$$

$$[i_L] = i_{L_{k-1}} + \frac{3h}{4} \cdot \underline{dj_L} + \frac{h}{4} \cdot \underline{di_L} \tag{2.134}$$

$$v_C = u_{C_{k-1}} + \frac{5h}{12} \cdot dv_C - \frac{5h}{12} \cdot du_C \tag{2.135}$$

$$u_C = u_{C_{k-1}} + \frac{3h}{4} \cdot dv_C + \frac{h}{4} \cdot du_C \tag{2.136}$$

There is only one variable in Eq.(2.113) so this equation must be used to compute v_0 and is then underlined in Eq.(2.118). The variable i_0 only appears in Eq.(2.131) so this equation must be used to compute i_0. Only 10 of the equations can immediately be sorted and the remaining 14 equations must be solved together. Although the model is linear, a Newton iteration may still be used to solve the remaining equations. For linear models, a Newton iteration will converge in one iteration step but this is still not efficient since more iteration variables are being used than necessary. To improve simulation efficiency, a method called *tearing* can be used to find a smaller set of iteration variables. With tearing, the remaining equations can be causalized and the number of iteration variables will be reduced. After all of the equations have been sorted and made causal, the equal sign is now used in the sense of assignment, where there is an individual equation to compute each variable.

2.4 Tearing

Physical system models, such as those that describe electrical circuits, may contain algebraic loops that are large. However, the subset of equations to be solved simultaneously that form the algebraic loop usually only have a small number of variables in each equation. In this case, by reducing the dimensionality of the system of equations to be solved, a solution can be found more efficiently. This is done by a technique called tearing in which the reduction can be done symbolically and then passed to the numerical solver [8]. In practice, the model may possibly contain thousands of equations in thousands of unknowns. For the numerical solver, a Newton iteration is efficient when there is a *small* set of iteration variables but becomes inefficient when there is a *large* number of iteration variables. Here, *large* describes systems with thousands of equations in thousands of iteration variables. With tearing, the number of iteration variables needed can be reduced.

Once the equation sorting algorithm stalls, the *tearing algorithm* needs to make assumptions about one or more variables to be known. The solution of the system is not changed by tearing; it only makes finding the solution more efficient. After inlining with an implicit Runge–Kutta algorithm, and only after inlining, tearing can be used to reduce the size of the Jacobian making the problem more efficient to solve. For the Rad3 example given in the previous section the equations can be completely sorted after tearing:

$$[v_0] = f\left(t_k + \frac{h}{3}\right) \tag{2.137}$$

$$\underline{v_1} = R_1 \cdot [j_1] \tag{2.138}$$

$$\underline{v_2} = R_2 \cdot [j_2] \tag{2.139}$$

$$\underline{v_L} = L \cdot [dj_L] \tag{2.140}$$

$$\underline{j_C} = C \cdot [dv_C] \tag{2.141}$$

$$\underline{v_0} = [v_1] + \underline{v_C} \tag{2.142}$$

$$[v_L] = \underline{v_1} + \underline{v_2} \tag{2.143}$$

$$\underline{v_C} = [v_2] \tag{2.144}$$

$$[j_0] = \underline{j_1} + \underline{j_L} \tag{2.145}$$

$$\underline{j_1} = \underline{j_2} + [j_C] \tag{2.146}$$

$$[u_0] = f(t_k + h) \tag{2.147}$$

$$[u_1] = R_1 \cdot \underline{i_1} \tag{2.148}$$

$$\underline{u_2} = R_2 \cdot [i_2] \tag{2.149}$$

$$\underline{u_L} = L \cdot [di_L] \tag{2.150}$$

$$[i_C] = C \cdot \underline{du_C} \tag{2.151}$$

$$\underline{u_0} = \underline{u_1} + [u_C] \tag{2.152}$$

$$[u_L] = \underline{u_1} + \underline{u_2} \tag{2.153}$$

$$\underline{u_C} = [u_2] \tag{2.154}$$

$$[i_0] = \underline{i_1} + \underline{i_L} \tag{2.155}$$

$$[i_1] = \underline{i_2} + \underline{i_C} \tag{2.156}$$

$$[j_L] = i_{L_{k-1}} + \frac{5h}{12} \cdot \underline{dj_L} - \frac{5h}{12} \cdot \underline{di_L} \tag{2.157}$$

$$[i_L] = i_{L_{k-1}} + \frac{3h}{4} \cdot \underline{dj_L} + \frac{h}{4} \cdot \underline{di_L} \tag{2.158}$$

$$[v_C] = u_{C_{k-1}} + \frac{5h}{12} \cdot \underline{dv_C} - \frac{5h}{12} \cdot \underline{du_C} \tag{2.159}$$

$$\underline{u_C} = u_{C_{k-1}} + \frac{3h}{4} \cdot \underline{dv_C} + \frac{h}{4} \cdot [du_C] \tag{2.160}$$

In this set of equations, the variables v_c and i_1 are assumed known and are called *tearing variables*. These two variables are the iteration variables for this problem. The tearing variables are computed from Eqs.(2.156) and (2.159) which are called *residual equations*. The tearing variables were found using a heuristic procedure [2] that always

results in a small number of tearing variables, but not necessarily the smallest number of tearing variables. A heuristic procedure must be used since finding the minimum number of tearing variables is an *np*–complete problem. After tearing, the size of the Jacobian for this set of equations has been reduced from a 24×24 matrix to a 2×2 matrix. While this set of equations can now be used for simulation, the simulation efficiency can again be improved by using *step–size* control.

2.5 Step–Size Control

One last piece for efficient simulation is step–size control of the integration algorithm. In general, smaller step sizes result in smaller integration errors and larger step sizes tend to lead to larger integration errors. However, choosing a smaller step size comes at the cost of a higher computational load. For this reason, using a *variable–step integration algorithm* may be desirable.

The concept for step size control is simple. First, perform the same integration step using two integration algorithms. Next, take the difference between the two computed values to find the estimated error ε. If ε is bigger than some specified error tolerance tol, then reject that step and repeat the same step with a smaller step size. Finally, if after a some number of consecutive integration steps the estimated error remains smaller than the tolerated error, then the step size is increased. For step size control, it is always possible to use a second separate integration algorithm in parallel and independent of the first algorithm. This is hardly efficient because the solver would then have the additional computational load of the second independent algorithm.

Traditionally, for explicit Runge–Kutta algorithms, an embedding method was found and used for step–size control. An embedded method is a second integration algorithm that has stages in common with the integration algorithm that it is embedded in so that the computational load is shared between them. Since fully implicit

algorithms like Radau IIA are so compact and optimized there doesn't exist any freedom to find an embedding method with the existing information. In [2], using information from two steps, embedding methods that can be used for step–size control of Rad3, Rad5 and Lobatto IIIC(4) have been found.

It should be mentioned that order–control is not even considered because accuracy is typically reduced by a factor of 10 when the order of the method is decreased by one [2]. Hardly any efficiency is gained and its impact on the accuracy of the solution does not justify the use of order–control. In addition, some implicit Runge–Kutta algorithms are not suited for use with order–control, since an integration algorithm within the same family may not exist at all one order lower. For instance, if a fifth order accurate Radau IIA method is used, there doesn't exist a forth-order accurate Radau IIA method to drop to [11].

2.5.1 Radau IIA

For Rad3, the embedding method found in [2] turned out to be 3^{rd}–order accurate and is given by:

$$\mathbf{x_{k+1}} = -\frac{1}{13}\mathbf{x_{k-1}} + \frac{2}{13}\mathbf{x_{k-\frac{2}{3}}} + \frac{14}{13}\mathbf{x_k} - \frac{2}{13}\mathbf{x_{k+\frac{1}{3}}} + \frac{11h}{13}\dot{\mathbf{x}}_{\mathbf{k+\frac{1}{3}}} + \frac{3h}{13}\dot{\mathbf{x}}_{\mathbf{k+1}} \tag{2.161}$$

While the embedding method found for Rad5 turned out to be 5^{th}–order accurate and is given by:

$$\begin{aligned}\mathbf{x_{k+1}} = {} & c_1\mathbf{x_{k-1}} + c_2 h\dot{\mathbf{x}}_{\mathbf{1_{k-1}}} + c_3\mathbf{x_{2_{k-1}}} + c_4 h\dot{\mathbf{x}}_{\mathbf{2_{k-1}}} + c_5\mathbf{x_k} \\ & + c_6\mathbf{x_{1_k}} + c_7 h\dot{\mathbf{x}}_{\mathbf{1_k}} + c_8\mathbf{x_{2_k}} + c_9 h\dot{\mathbf{x}}_{\mathbf{2_k}} + c_{10} h\dot{\mathbf{x}}_{\mathbf{k+1}}\end{aligned} \tag{2.162}$$

where the coefficients are:

$$\begin{aligned} c_1 &= -0.00517140382204 & c_2 &= -0.00094714677404 \\ c_3 &= -0.04060469717694 & c_4 &= -0.01364429384901 \\ c_5 &= +1.41786808325433 & c_6 &= -0.17475783086782 \end{aligned}$$

$$c_7 = +0.48299282769491 \qquad c_8 = -0.19733415138754$$
$$c_9 = +0.55942205973218 \qquad c_{10} = +0.10695524944855$$

2.5.2 Lobatto IIIC

The embedding method found in [2] for Lobatto IIIC(4) is 4^{th}–order accurate and is given by:

$$\begin{aligned}\mathbf{x_{k+1}} = &\frac{63}{4552}\mathbf{x_{k-1}} - \frac{91h}{81936}\dot{\mathbf{x}}_{\mathbf{1_{k-1}}} + \frac{1381h}{81936}\dot{\mathbf{x}}_{\mathbf{2_{k-1}}} + \frac{3101}{2276}\mathbf{x_k} - \frac{393}{4552}\mathbf{x_{1_k}} \\ &+ \frac{775h}{3414}\dot{\mathbf{x}}_{\mathbf{1_k}} - \frac{165}{569}\mathbf{x_{2_k}} + \frac{62179h}{81936}\dot{\mathbf{x}}_{\mathbf{2_k}} + \frac{12881h}{81936}\dot{\mathbf{x}}_{\mathbf{k+1}}\end{aligned} \tag{2.163}$$

With these error methods, the step size can only change after two consecutive steps. This is a minor restriction as stated in [2] and the new step can be computed using a standard rule:

$$h_{new} = h_{old} \cdot \delta \cdot \left(\frac{Tol}{err}\right)^{1/(\hat{n}+1)} \tag{2.164}$$

where Tol is the specified tolerance, δ is a safety factor, err is the estimated error, and $\hat{n}$ is the order of the error estimate [10, 11]. The estimated error is given by:

$$err = \|\hat{\mathbf{x}} - \mathbf{x}\|_2 \tag{2.165}$$

where $\mathbf{x}$ is computed by the integration algorithm and $\hat{\mathbf{x}}$ is computed from the embedding method. To prevent the step size from changing drastically, the new step size can be limited to values between $[h/2, 2h]$.

2.6 HW-SDIRK and Lobatto IIIC(6)

In this thesis, two different implicit Runge–Kutta algorithms will be studied: HW-SDIRK(3)4 and Lobatto IIIC(6). Following the idea of using data from a previous step, it may be possible to find an alternative embedding method for HW-SDIRK and to find an embedding method for Lobatto IIIC(6). Together with the above techniques, all of the integration methods will then be compared with each other using benchmark ODEs [12].

Chapter 3

HW–SDIRK

A diagonally implicit Runge–Kutta (DIRK) algorithm contains zero elements above the main diagonal of the Butcher tableau. A method of this type that has non–zero elements of equal value on the main diagonal is called a singly diagonally implicit Runge–Kutta (SDIRK) algorithm.

An SDIRK algorithm containing third and fourth order accurate methods is HW-SDIRK(3)4 [11]. The Butcher tableau describing these algorithms is shown in Table 3.1 with the third order accurate method denoted by $\hat{x}$.

3.1 Numerical Stability

Plugging the HW-SDIRK algorithm into the standard test problem of Eq.(1.5), the following set of equations in ODE form results:

$$\mathbf{k_1} = \left[\mathbf{I^{(n)}} - \frac{5\mathbf{A}h}{4} + \frac{5(\mathbf{A}h)^2}{8} - \frac{5(\mathbf{A}h)^3}{32} + \frac{5(\mathbf{A}h)^4}{256} - \frac{(\mathbf{A}h)^5}{1024}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \mathbf{A}h + \frac{3(\mathbf{A}h)^2}{8} - \frac{(\mathbf{A}h)^3}{16} - \frac{(\mathbf{A}h)^4}{256}\right)\mathbf{A}h \tag{3.1}$$

Table 3.1. HW-SDIRK(3)4

$\frac{1}{4}$	$\frac{1}{4}$	0	0	0	0
$\frac{3}{4}$	$\frac{1}{2}$	$\frac{1}{4}$	0	0	0
$\frac{11}{20}$	$\frac{17}{50}$	$\frac{-1}{25}$	$\frac{1}{4}$	0	0
$\frac{1}{2}$	$\frac{371}{1360}$	$\frac{-137}{2720}$	$\frac{15}{544}$	$\frac{1}{4}$	0
1	$\frac{25}{24}$	$\frac{-49}{48}$	$\frac{125}{16}$	$\frac{-85}{12}$	$\frac{1}{4}$
$\hat{x}$	$\frac{59}{48}$	$\frac{-17}{96}$	$\frac{225}{32}$	$\frac{-85}{12}$	0
x	$\frac{25}{24}$	$\frac{-49}{48}$	$\frac{125}{16}$	$\frac{-85}{12}$	$\frac{1}{4}$

$$\mathbf{k_2} = \left[\mathbf{I^{(n)}} - \frac{5\mathbf{A}h}{4} + \frac{5(\mathbf{A}h)^2}{8} - \frac{5(\mathbf{A}h)^3}{32} + \frac{5(\mathbf{A}h)^4}{256} - \frac{(\mathbf{A}h)^5}{1024}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \frac{\mathbf{A}h}{2} - \frac{(\mathbf{A}h)^3}{32} - \frac{(\mathbf{A}h)^4}{256}\right)\mathbf{A}h \tag{3.2}$$

$$\mathbf{k_3} = \left[\mathbf{I^{(n)}} - \frac{5\mathbf{A}h}{4} + \frac{5(\mathbf{A}h)^2}{8} - \frac{5(\mathbf{A}h)^3}{32} + \frac{5(\mathbf{A}h)^4}{256} - \frac{(\mathbf{A}h)^5}{1024}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \frac{7\mathbf{A}h}{10} + \frac{13(\mathbf{A}h)^2}{100} + \frac{3(\mathbf{A}h)^3}{800} - \frac{(\mathbf{A}h)^4}{492}\right)\mathbf{A}h \tag{3.3}$$

$$\mathbf{k_4} = \left[\mathbf{I^{(n)}} - \frac{5\mathbf{A}h}{4} + \frac{5(\mathbf{A}h)^2}{8} - \frac{5(\mathbf{A}h)^3}{32} + \frac{5(\mathbf{A}h)^4}{256} - \frac{(\mathbf{A}h)^5}{1024}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \frac{3\mathbf{A}h}{4} + \frac{29(\mathbf{A}h)^2}{170} - \frac{2(\mathbf{A}h)^3}{259} - \frac{(\mathbf{A}h)^4}{1088}\right)\mathbf{A}h \tag{3.4}$$

$$\mathbf{k_5} = \left[\mathbf{I^{(n)}} - \frac{5\mathbf{A}h}{4} + \frac{5(\mathbf{A}h)^2}{8} - \frac{5(\mathbf{A}h)^3}{32} + \frac{5(\mathbf{A}h)^4}{256} - \frac{(\mathbf{A}h)^5}{1024}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \frac{\mathbf{A}h}{4} - \frac{(\mathbf{A}h)^2}{8} + \frac{(\mathbf{A}h)^3}{96} + \frac{7(\mathbf{A}h)^4}{768}\right)\mathbf{A}h \tag{3.5}$$

$$\hat{\mathbf{x}}_{\mathbf{k+1}} = \mathbf{x_k} + h\left(\frac{59\mathbf{k_1}}{48} - \frac{17\mathbf{k_2}}{96} + \frac{225\mathbf{k_3}}{32} - \frac{85\mathbf{k_4}}{12}\right) \tag{3.6}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + h\left(\frac{25\mathbf{k_1}}{24} - \frac{49\mathbf{k_2}}{48} + \frac{125\mathbf{k_3}}{16} - \frac{85\mathbf{k_4}}{12} + \frac{\mathbf{k_5}}{4}\right) \tag{3.7}$$

where:

$\mathbf{k_1}$ = the state derivative 1/4 into the step

$\mathbf{k_2}$ = the state derivative 3/4 into the step

$\mathbf{k_3}$ = the state derivative 11/20 into the step

$\mathbf{k_4}$ = the state derivative 1/2 into the step

$\mathbf{k_5}$ = the state derivative at the end of the step

These equations can be expressed in the form of a discrete time system:

$$\mathbf{x_{k+1}} = \mathbf{F} \cdot \mathbf{x_k} \tag{3.8}$$

where $\mathbf{F}$ is the new discrete state matrix and the simulation time is now indexed.

Therefore, the third–order accurate HW-SDIRK method is characterized by the following **F**–matrix:

$$\begin{aligned}\mathbf{F} = \mathbf{I^{(n)}} + \left[\mathbf{I^{(n)}} - \frac{5\mathbf{A}h}{4} + \frac{5(\mathbf{A}h)^2}{8} - \frac{5(\mathbf{A}h)^3}{32} + \frac{5(\mathbf{A}h)^4}{256} - \frac{(\mathbf{A}h)^5}{1024}\right]^{-1} \\ \cdot \left(\mathbf{I^{(n)}} - \frac{3\mathbf{A}h}{4} + \frac{(\mathbf{A}h)^2}{6} - \frac{(\mathbf{A}h)^3}{768} - \frac{(\mathbf{A}h)^4}{439}\right)\mathbf{A}h\end{aligned} \tag{3.9}$$

A Taylor series can be developed around $h = 0$:

$$\mathbf{F} \approx \mathbf{I^{(n)}} + \mathbf{A}h + \frac{(\mathbf{A}h)^2}{2} + \frac{(\mathbf{A}h)^3}{6} + \frac{13(\mathbf{A}h)^4}{256} \tag{3.10}$$

Comparing this equation with the Taylor series of the analytical solution:

$$\mathbf{F} = \exp(\mathbf{A}t) = \mathbf{I^{(n)}} + \mathbf{A}h + \frac{(\mathbf{A}h)^2}{2} + \frac{(\mathbf{A}h)^3}{6} + \frac{(\mathbf{A}h)^4}{24} + \frac{(\mathbf{A}h)^5}{120} + \frac{(\mathbf{A}h)^6}{720} + \dots \tag{3.11}$$

it can be seen that this method is indeed third–order accurate and that the error coefficient is:

$$\varepsilon = \frac{7(\mathbf{A}h)^4}{768} \tag{3.12}$$

The fourth-order accurate HW-SDIRK method is characterized by the following **F**–matrix:

$$\begin{aligned}\mathbf{F} = \mathbf{I^{(n)}} + \left[\mathbf{I^{(n)}} - \frac{5\mathbf{A}h}{4} + \frac{5(\mathbf{A}h)^2}{8} - \frac{5(\mathbf{A}h)^3}{32} + \frac{5(\mathbf{A}h)^4}{256} - \frac{(\mathbf{A}h)^5}{1024}\right]^{-1} \\ \cdot \left(\mathbf{I^{(n)}} - \frac{3\mathbf{A}h}{4} + \frac{(\mathbf{A}h)^2}{6} - \frac{(\mathbf{A}h)^3}{96} + \frac{(\mathbf{A}h)^4}{1024}\right)\mathbf{A}h\end{aligned} \tag{3.13}$$

A Taylor series can be developed around $h = 0$

$$\mathbf{F} \approx \mathbf{I^{(n)}} + \mathbf{A}h + \frac{(\mathbf{A}h)^2}{2} + \frac{(\mathbf{A}h)^3}{6} + \frac{(\mathbf{A}h)^4}{24} + \frac{23(\mathbf{A}h)^5}{3072} \tag{3.14}$$

giving an error coefficient of:

$$\varepsilon = \frac{13(\mathbf{A}h)^5}{15360} \tag{3.15}$$

The stability domains of the third– and fourth–order methods are shown in Figures 3.1–3.2, respectively. Evidently, despite being part of an overall implicit algo-

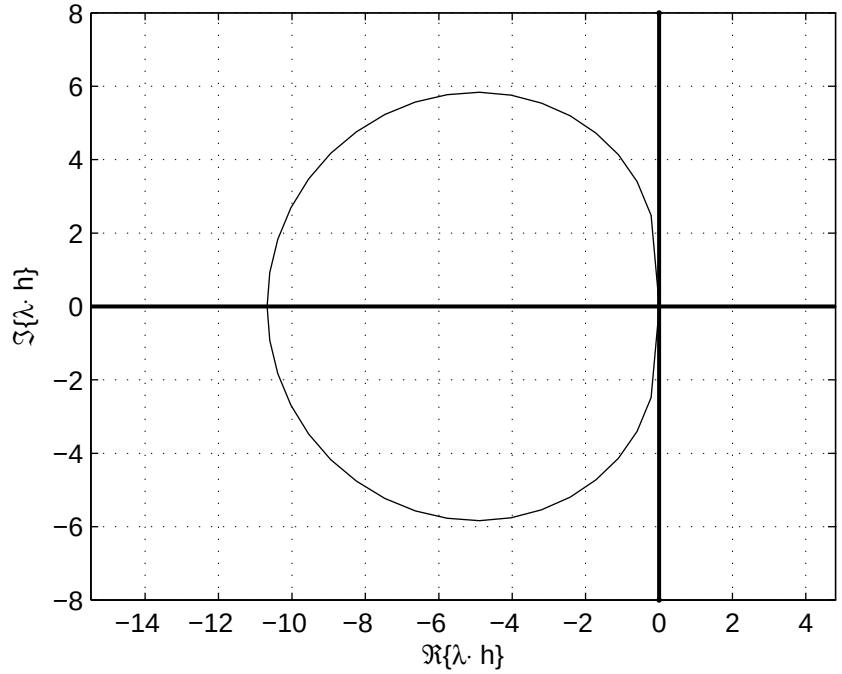

FIGURE 3.1. Stability domain of HW-SDIRK(3)

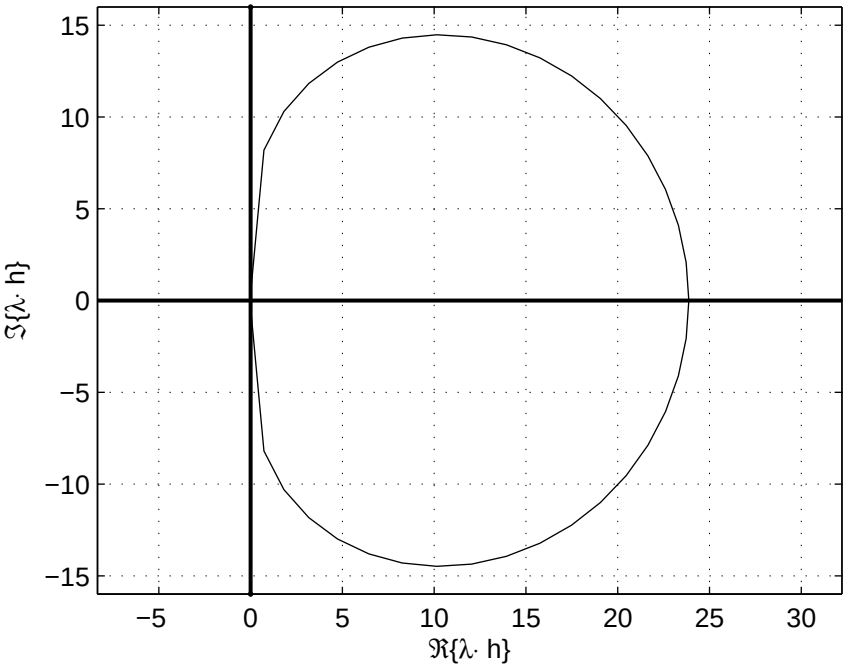

FIGURE 3.2. Stability domain of HW-SDIRK(4)

rithm, the 3rd–order accurate embedding method behaves like an explicit method as the stability domain loops in the left half of the λh–plane. In this case, the region inside the curve of Figure 3.1 is the stable region and everything outside the curve is the unstable region.

3.2 Numerical Damping

In order to judge the accuracy of an integration algorithm, a *damping plot* can be used [2]. Again using the standard linear test problem of Eq.(1.5), and choosing the initial conditions such that $t_0 = t_k$ and $\mathbf{x_0} = \mathbf{x_k}$, the solution for $t = t_{k+1}$ is:

$$\mathbf{x_{k+1}} = \exp(\mathbf{A}h) \cdot \mathbf{x_k} \tag{3.16}$$

The discrete system then has the analytical $\mathbf{F}$–matrix:

$$\mathbf{F} = \exp(\mathbf{A}h) \tag{3.17}$$

with eigenvalues:

$$\lambda_\mathbf{d} = \mathrm{eig}(\mathbf{F}) = \exp(\mathrm{eig}(\mathbf{A})h) \tag{3.18}$$

or since the eigenvalues are complex:

$$\begin{aligned} \lambda_\mathbf{d} &= \exp(\lambda_i h) \\ &= \exp((-\sigma_i + j\omega_i)h) \\ &= \exp(-\sigma_i h) \cdot \exp(j\omega_i h) \end{aligned} \tag{3.19}$$

In the continuous system, the damping σ is the distance of an eigenvalue from the imaginary axis in the λ–plane. This damping maps in the discrete system to a distance from the origin in the $\exp(\lambda \cdot h)$–plane. This can be recognized as the *z–domain*, where $z = \exp(\lambda \cdot h)$. In [2], the analytical discrete damping is then defined as $\sigma_d = \sigma_i h$. The $\mathbf{F}$–matrix is then related to the numerical discrete damping $\hat{\sigma}_d$ by:

$$\hat{\sigma}_d = \log(\max(|\,\mathrm{eig}(\mathbf{F})|)) \tag{3.20}$$

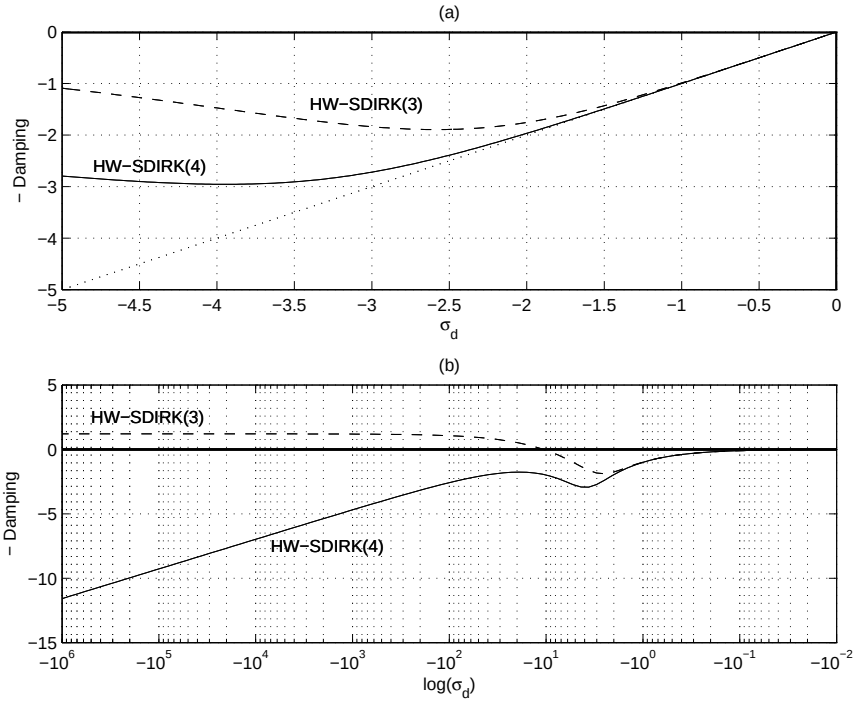

FIGURE 3.3. HW-SDIRK(3)4: a) Damping plot; b) Logarithmic damping plot

where **F** is now the **F**–matrix of the numerical solver.

The damping plot is then given by plotting both $-\sigma_d$ and $-\hat{\sigma}_d$ against $-\sigma_d$. The damping plots of of the third– and fourth–order methods are shown in Figure 3.3 with σ_d shown as the dotted line. From Figure 3.3 it can be seen that the fourth–order method is L-stable while the damping for the third–order method becomes negative.

3.3 Step–Size Control

Although HW-SDIRK already has a proper embedding method, it behaves like an explicit method. For step–size control, using such an embedding method to compute the error estimate may unnecessarily restrict the step size when solving a stiff system. While the simulation should proceed without incident, this may not be efficient as the solver may need to take more integration steps than required. Perhaps the same idea from [2] can be used to find an alternate embedding method for HW-SDIRK.

Since HW-SDIRK is not a compact algorithm, needing five stages to generate a fourth order accurate method, it should be possible to search for an alternate implicit embedding method. Solving a stiff system together with an implicit embedding method to compute the error estimate should allow for the use of larger step sizes. A single step of HW-SDIRK has the following 10 data points: $\mathbf{x_k}$, $\mathbf{x_{k+1/4}}$, $\mathbf{x_{k+3/4}}$, $\mathbf{x_{k+11/20}}$, $\mathbf{x_{k+1/2}}$, $\dot{\mathbf{x}}_{\mathbf{k+1/4}}$, $\dot{\mathbf{x}}_{\mathbf{k+3/4}}$, $\dot{\mathbf{x}}_{\mathbf{k+11/20}}$, $\dot{\mathbf{x}}_{\mathbf{k+1/2}}$, and $\dot{\mathbf{x}}_{\mathbf{k+1}}$. Looking for a 5^{th} order polynomial requires 6 of the 20 available data points giving 38760 different methods to be evaluated. The first four stages of this method are only 1^{st} order accurate, so none of the 5^{th} order polynomials are expected to be greater than 1^{st} order accurate. By blending, the approximation order is increased by one for each additional method. Therefore, at least 3 of these methods need to be blended:

$$\mathbf{x}_{\mathbf{k+1}}^{\text{blended}} = \alpha \mathbf{x}_{\mathbf{k+1}}^{1} + \beta \mathbf{x}_{\mathbf{k+1}}^{2} + (1 - \alpha - \beta)\mathbf{x}_{\mathbf{k+1}}^{3} \tag{3.21}$$

to create another 3^{rd}–order accurate method. A decent 3^{rd}–order accurate method that can be used for step–size control is the following:

$$\begin{aligned} \mathbf{x_{k+1}} = {} & c_1 \mathbf{x_{1_{k-1}}} + c_2 \mathbf{x_{3_{k-1}}} + c_3 \mathbf{x_k} + c_4 h \dot{\mathbf{x}}_{\mathbf{1_k}} + c_5 \mathbf{x_{2_k}} \\ & + c_6 h \dot{\mathbf{x}}_{\mathbf{2_k}} + c_7 \mathbf{x_{3_k}} + c_8 h \dot{\mathbf{x}}_{\mathbf{3_k}} + c_9 \mathbf{x_{4_k}} + c_{10} h \dot{\mathbf{x}}_{\mathbf{4_k}} + c_{11} h \dot{\mathbf{x}}_{\mathbf{k+1}} \end{aligned} \tag{3.22}$$

where the coefficients are:

$$\begin{aligned} c_1 &= 0.03987986285618 & c_2 &= -0.01359695108066 \\ c_3 &= -3.64539939561975 & c_4 &= -1.87066547933118 \\ c_5 &= -2.52317069157776 & c_6 &= -0.38422673936017 \\ c_7 &= 57.03714503309798 & c_8 &= 1.43334998089280 \\ c_9 &= -49.89485785767970 & c_{10} &= -2.92514906283553 \\ c_{11} &= 0.23985974910811 \end{aligned}$$

In this equation, $\mathbf{x_{1_{k-1}}}$ represents the state vector computed for the first stage of HW-SDIRK from the previous integration step, $\mathbf{x_{3_{k-1}}}$ represents the state vector

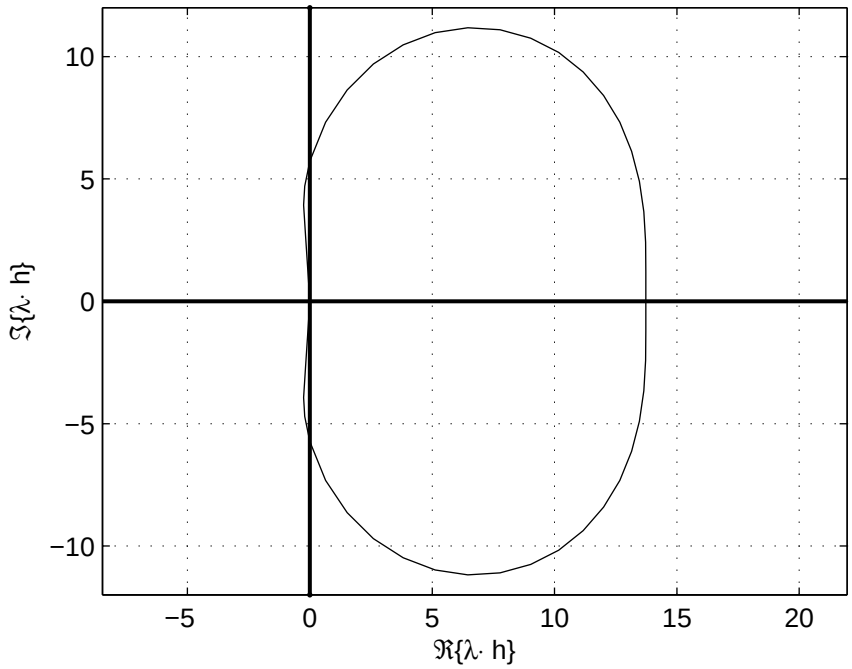

FIGURE 3.4. Stability domain of the HW-SDIRK alternate error method

computed for the third stage of the previous step, $\dot{\mathbf{x}}_{\mathbf{2_k}}$ represents the state derivative vector computed for the second stage of the current step, and so on.

The stability domain and damping plots of this method are shown in Figures 3.4–3.5, respectively. This method is not L–stable, however, it is implicit and is better suited for the purposes of step size control when solving stiff systems than the original embedding method.

3.4 Inlining

The various stages of this integration algorithm can be written as:

$$\mathbf{x}_{\mathbf{k}+\frac{\mathbf{1}}{\mathbf{4}}} = \mathbf{x}_{\mathbf{k}} + \frac{h}{4} \cdot \dot{\mathbf{x}}_{\mathbf{k}+\frac{\mathbf{1}}{\mathbf{4}}} \tag{3.23}$$

$$\mathbf{x}_{\mathbf{k}+\frac{\mathbf{3}}{\mathbf{4}}} = \mathbf{x}_{\mathbf{k}} + \frac{h}{2} \cdot \dot{\mathbf{x}}_{\mathbf{k}+\frac{\mathbf{1}}{\mathbf{4}}} + \frac{h}{4} \cdot \dot{\mathbf{x}}_{\mathbf{k}+\frac{\mathbf{3}}{\mathbf{4}}} \tag{3.24}$$

$$\mathbf{x}_{\mathbf{k}+\frac{\mathbf{11}}{\mathbf{20}}} = \mathbf{x}_{\mathbf{k}} + \frac{17h}{50} \cdot \dot{\mathbf{x}}_{\mathbf{k}+\frac{\mathbf{1}}{\mathbf{4}}} - \frac{h}{25} \cdot \dot{\mathbf{x}}_{\mathbf{k}+\frac{\mathbf{3}}{\mathbf{4}}} + \frac{h}{4} \cdot \dot{\mathbf{x}}_{\mathbf{k}+\frac{\mathbf{11}}{\mathbf{20}}} \tag{3.25}$$

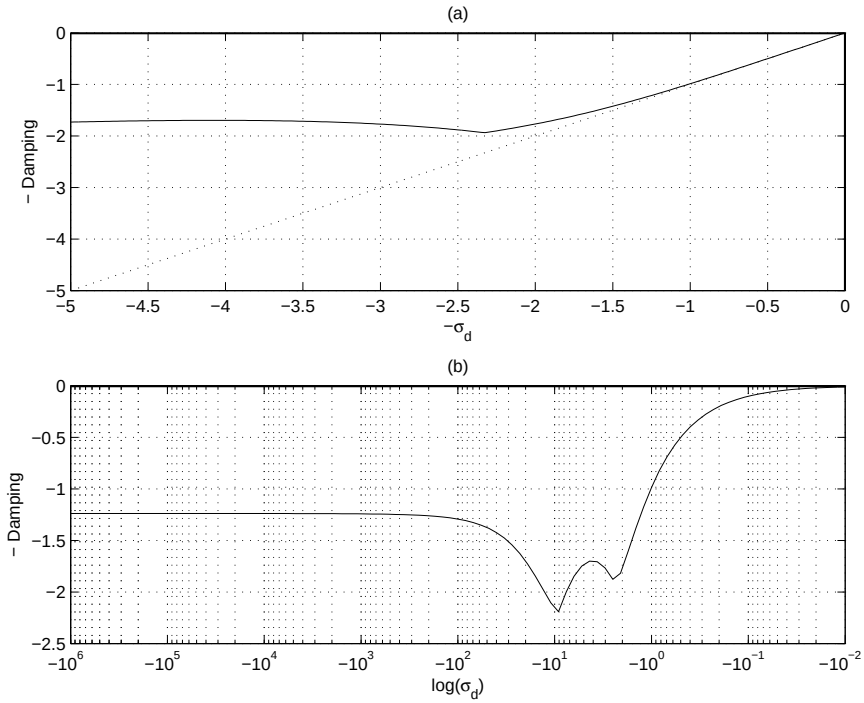

FIGURE 3.5. HW-SDIRK alternate error method: a) Damping plot; b) Logarithmic damping plot

$$\mathbf{x_{k+\frac{1}{2}}} = \mathbf{x_k} + \frac{371h}{1360}\dot{\mathbf{x}}_{\mathbf{k+\frac{1}{4}}} - \frac{137h}{2720}\dot{\mathbf{x}}_{\mathbf{k+\frac{3}{4}}} + \frac{15h}{544}\dot{\mathbf{x}}_{\mathbf{k+\frac{11}{20}}} + \frac{h}{4}\dot{\mathbf{x}}_{\mathbf{k+\frac{1}{2}}} \tag{3.26}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{25h}{24}\dot{\mathbf{x}}_{\mathbf{k+\frac{1}{4}}} - \frac{49h}{48}\dot{\mathbf{x}}_{\mathbf{k+\frac{3}{4}}} + \frac{125h}{16}\dot{\mathbf{x}}_{\mathbf{k+\frac{11}{20}}} - \frac{85h}{12}\dot{\mathbf{x}}_{\mathbf{k+\frac{1}{2}}} + \frac{h}{4}\dot{\mathbf{x}}_{\mathbf{k+1}} \tag{3.27}$$

$$\hat{\mathbf{x}}_{\mathbf{k+1}} = \mathbf{x_k} + \frac{59h}{48}\dot{\mathbf{x}}_{\mathbf{k+\frac{1}{4}}} - \frac{17h}{96}\dot{\mathbf{x}}_{\mathbf{k+\frac{3}{4}}} + \frac{225h}{32}\dot{\mathbf{x}}_{\mathbf{k+\frac{11}{20}}} - \frac{85h}{12}\dot{\mathbf{x}}_{\mathbf{k+\frac{1}{2}}} \tag{3.28}$$

As before, using the same circuit shown in Figure 2.1 the model equations need to be replicated once for each stage of the integration algorithm. For HW-SDIRK(3)4, the first two of the five sets of equations are reproduced here:

$$v_0 = f\left(t + \frac{h}{4}\right) \tag{3.29}$$

$$v_1 = R_1 \cdot j_1 \tag{3.30}$$

$$v_2 = R_2 \cdot j_2 \tag{3.31}$$

$$v_L = L \cdot dj_L \tag{3.32}$$

$$j_C = C \cdot dv_C \tag{3.33}$$

$$v_0 = v_1 + v_C \tag{3.34}$$

$$v_L = v_1 + v_2 \tag{3.35}$$

$$v_C = v_2 \tag{3.36}$$

$$j_0 = j_1 + j_L \tag{3.37}$$

$$j_1 = j_2 + j_C \tag{3.38}$$

$$u_0 = f\left(t + \frac{3h}{4}\right) \tag{3.39}$$

$$u_1 = R_1 \cdot i_1 \tag{3.40}$$

$$u_2 = R_2 \cdot i_2 \tag{3.41}$$

$$u_L = L \cdot di_L \tag{3.42}$$

$$i_C = C \cdot du_C \tag{3.43}$$

$$u_0 = u_1 + u_C \tag{3.44}$$

$$u_L = u_1 + u_2 \tag{3.45}$$

$$u_C = u_2 \tag{3.46}$$

$$i_0 = i_1 + i_L \tag{3.47}$$

$$i_1 = i_2 + i_C \tag{3.48}$$

$$j_L = \text{pre}(i_L) + \frac{h}{4} \cdot dj_L \tag{3.49}$$

$$i_L = \text{pre}(i_L) + \frac{h}{2} \cdot dj_L + \frac{h}{4} \cdot di_L \tag{3.50}$$

$$v_C = \text{pre}(u_C) + \frac{h}{4} \cdot dv_C \tag{3.51}$$

$$u_C = \text{pre}(u_C) + \frac{h}{2} \cdot dv_C + \frac{h}{4} \cdot du_C \tag{3.52}$$

$$\tag{3.53}$$

The complete set of inlined equations consists of 60 equations in 60 unknowns. As mentioned before, the procedure is similar for inlining with any integration algorithm. The structure incidence matrix for the HW-SDIRK set of inlined equations can be written as:

$$\mathbf{S_{hwsdirk}} = \begin{pmatrix} \mathbf{S} & \mathbf{Z} & \mathbf{Z} & \mathbf{Z} & \mathbf{Z} \\ \mathbf{C} & \mathbf{S} & \mathbf{Z} & \mathbf{Z} & \mathbf{Z} \\ \mathbf{C} & \mathbf{C} & \mathbf{S} & \mathbf{Z} & \mathbf{Z} \\ \mathbf{C} & \mathbf{C} & \mathbf{C} & \mathbf{S} & \mathbf{Z} \\ \mathbf{C} & \mathbf{C} & \mathbf{C} & \mathbf{C} & \mathbf{S} \end{pmatrix} \tag{3.54}$$

where $\mathbf{Z}$ is the zero matrix. The structure incidence matrix after inlining is lower block–triangular even before sorting, but this is only true for HW-SDIRK. The advantage of this is that one Newton iteration per stage can be used instead of one Newton iteration across all stages. As model complexity grows it should be obvious that this process of inlining, sorting and causalizing, tearing, and setting up the Newton iterations must be automated. The number of equations can become quite large and this process can no longer be done by hand.

Chapter 4

Lobatto IIIC(6)

Another Lobatto IIIC method has also been published that is sixth order accurate [11]. This algorithm uses four stages but also has the potential disadvantage of irrational coefficients and time steps. The Lobatto IIIC(6) algorithm is described by the Butcher tableau of Table 4.1.

4.1 Numerical Stability and Damping

Again using the standard linear test problem, $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$, with the ODE description of Lobatto IIIC results in the following set of equations:

$$\mathbf{k_1} = \left[\mathbf{I^{(n)}} - \frac{2\mathbf{A}h}{3} + \frac{(\mathbf{A}h)^2}{5} - \frac{(\mathbf{A}h)^3}{30} + \frac{(\mathbf{A}h)^4}{360}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \frac{2\mathbf{A}h}{3} + \frac{(\mathbf{A}h)^2}{5} - \frac{(\mathbf{A}h)^3}{30}\right)\mathbf{A}h \tag{4.1}$$

$$\mathbf{k_2} = \left[\mathbf{I^{(n)}} - \frac{2\mathbf{A}h}{3} + \frac{(\mathbf{A}h)^2}{5} - \frac{(\mathbf{A}h)^3}{30} + \frac{(\mathbf{A}h)^4}{360}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \frac{329\mathbf{A}h}{843} + \frac{61(\mathbf{A}h)^2}{1131}\right)\mathbf{A}h \tag{4.2}$$

$$\mathbf{k_3} = \left[\mathbf{I^{(n)}} - \frac{2\mathbf{A}h}{3} + \frac{(\mathbf{A}h)^2}{5} - \frac{(\mathbf{A}h)^3}{30} + \frac{(\mathbf{A}h)^4}{360}\right]^{-1}$$

Table 4.1. Lobatto IIIC(6)

0	$\frac{1}{12}$	$\frac{-\sqrt{5}}{12}$	$\frac{\sqrt{5}}{12}$	$\frac{-1}{12}$
$\frac{5-\sqrt{5}}{10}$	$\frac{1}{12}$	$\frac{1}{4}$	$\frac{10-7\sqrt{5}}{60}$	$\frac{\sqrt{5}}{60}$
$\frac{5+\sqrt{5}}{10}$	$\frac{1}{12}$	$\frac{10+7\sqrt{5}}{60}$	$\frac{1}{4}$	$\frac{-\sqrt{5}}{60}$
1	$\frac{1}{12}$	$\frac{5}{12}$	$\frac{5}{12}$	$\frac{1}{12}$
x	$\frac{1}{12}$	$\frac{5}{12}$	$\frac{5}{12}$	$\frac{1}{12}$

$$\cdot \left(\mathbf{I^{(n)}} + \frac{16\mathbf{A}h}{281} - \frac{24(\mathbf{A}h)^2}{1165} - \frac{(\mathbf{A}h)^3}{30} \right) \mathbf{A}h \tag{4.3}$$

$$\mathbf{k_4} = \left[\mathbf{I^{(n)}} - \frac{2\mathbf{A}h}{3} + \frac{(\mathbf{A}h)^2}{5} - \frac{(\mathbf{A}h)^3}{30} + \frac{(\mathbf{A}h)^4}{360} \right]^{-1} \cdot \left(\mathbf{I^{(n)}} + \frac{\mathbf{A}h}{3} + \frac{(\mathbf{A}h)^2}{30} \right) \mathbf{A}h \tag{4.4}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{h}{12} \left(\mathbf{k_1} + 5\mathbf{k_2} + 5\mathbf{k_3} + \mathbf{k_4} \right) \tag{4.5}$$

The sixth order accurate Lobatto IIIC method is characterized by the following **F**–matrix:

$$\mathbf{F} = \mathbf{I^{(n)}} + \left[\mathbf{I^{(n)}} - \frac{2\mathbf{A}h}{3} + \frac{\mathbf{A}h^2}{5} - \frac{\mathbf{A}h^3}{30} + \frac{\mathbf{A}h^4}{360} \right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \frac{(\mathbf{A}h)}{6} + \frac{(\mathbf{A}h)^2}{30} - \frac{(\mathbf{A}h)^3}{360} \right) \mathbf{A}h \tag{4.6}$$

Developing a Taylor series around $h = 0$

$$\mathbf{F} \approx \mathbf{I^{(n)}} + \mathbf{A}h + \frac{(\mathbf{A}h)^2}{2} + \frac{(\mathbf{A}h)^3}{6} + \frac{(\mathbf{A}h)^4}{24} + \frac{(\mathbf{A}h)^5}{120} + \frac{(\mathbf{A}h)^6}{720} + \frac{(\mathbf{A}h)^7}{5400} \tag{4.7}$$

and comparing this with the Taylor series of the analytical solution gives an error coefficient of:

$$\varepsilon = \frac{(\mathbf{A}h)^7}{75600} \tag{4.8}$$

4.2 Step–size control

Using the same idea as in [2], let us look for an embedding method for the sixth order accurate Lobatto IIIC algorithm, since no embedding method currently exists. A single step of Lobatto IIIC(6) has the following data points: $\mathbf{x_k}$, $\mathbf{x_{k+0}}$, $\mathbf{x}_{\mathbf{k}+\frac{5-\sqrt{5}}{10}}$, $\mathbf{x}_{\mathbf{k}+\frac{5+\sqrt{5}}{10}}$, $\dot{\mathbf{x}}_\mathbf{k}$, $\dot{\mathbf{x}}_{\mathbf{k+0}}$, $\dot{\mathbf{x}}_{\mathbf{k}+\frac{5-\sqrt{5}}{10}}$, $\dot{\mathbf{x}}_{\mathbf{k}+\frac{5+\sqrt{5}}{10}}$, and $\dot{\mathbf{x}}_{\mathbf{k+1}}$. This algorithm unfortunately has the same problem as the 4$^{\text{th}}$–order accurate method, namely, there is a zero time advance from the fourth stage of a step to the first stage of the next step. The data points $\mathbf{x_k}$ and $\mathbf{x_{k+0}}$ represent the state vector at the same time instant so no

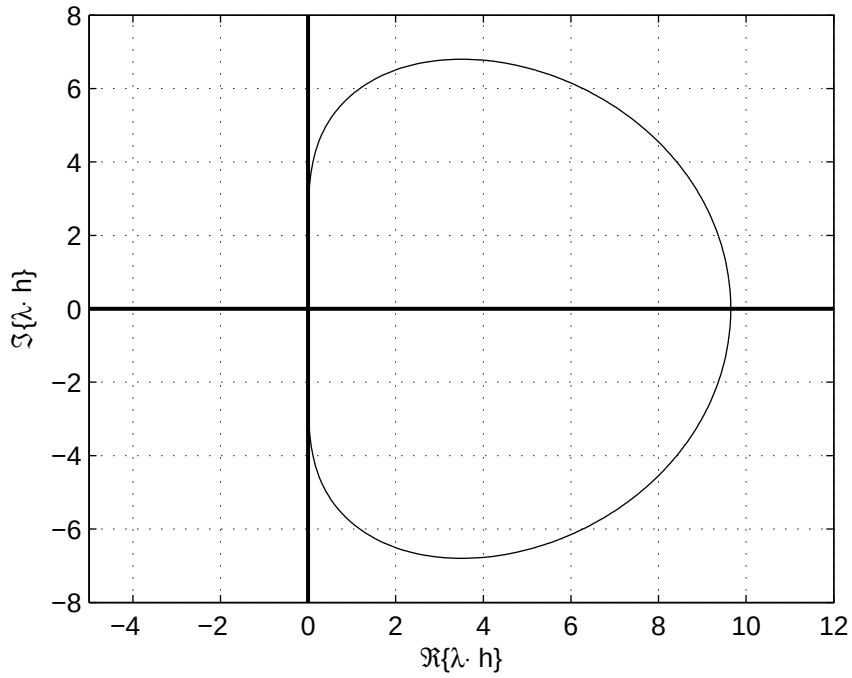

FIGURE 4.1. Stability domain of Lobatto IIIC(6)

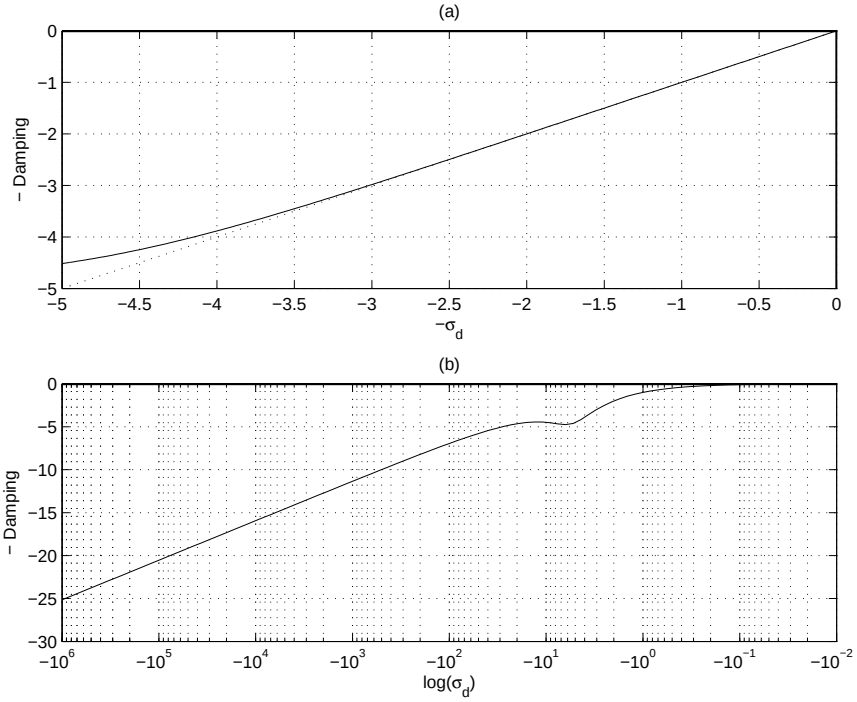

FIGURE 4.2. Lobatto IIIC(6): a) Damping plot; b) Logarithmic damping plot

single error method can use them both simultaneously. The first three stages of this algorithm are only 3rd–order accurate so none of the polynomials being searched for are expected to be of greater accuracy. To find a 5th–order accurate method, three suitable polynomials will need to be blended by Eq.(3.21). The following 5th–order accurate method can be used for step size control:

$$\begin{aligned}\mathbf{x_{k+1}} = {} & c_1\mathbf{x_{k-1}} + c_2 h\dot{\mathbf{x}}_{\mathbf{1_{k-1}}} + c_3\mathbf{x_{2_{k-1}}} + c_4 h\dot{\mathbf{x}}_{\mathbf{2_{k-1}}} + c_5\mathbf{x_k} \\ & + c_6\mathbf{x_{1_k}} + c_7 h\dot{\mathbf{x}}_{\mathbf{1_k}} + c_8\mathbf{x_{2_k}} + c_9 h\dot{\mathbf{x}}_{\mathbf{2_k}} + c_{10} h\dot{\mathbf{x}}_{\mathbf{3_k}} + c_{11} h\dot{\mathbf{x}}_{\mathbf{k+1}}\end{aligned} \tag{4.9}$$

where the coefficients are:

$$\begin{aligned} c_1 &= 0.02061173185679 & c_2 &= 0.00133204868429 \\ c_3 &= -0.00704709981316 & c_4 &= 0.00908420357588 \\ c_5 &= 0.83597287802675 & c_6 &= 0.03506343063678 \\ c_7 &= 0.07743334426486 & c_8 &= 0.11539905929277 \\ c_9 &= 0.39352639145014 & c_{10} &= 0.41959151135295 \\ c_{11} &= 0.08264938766485 \end{aligned}$$

Similarly in this equation, $\dot{\mathbf{x}}_{\mathbf{1_{k-1}}}$ represents the state derivative vector computed for the first stage of Lobatto IIIC(6) from the previous integration step, $\mathbf{x_{2_{k-1}}}$ represents the state vector computed for the second stage of the previous integration step, $\dot{\mathbf{x}}_{\mathbf{2_k}}$ represents the state derivative vector computed for the second stage of the current integration step, and so on.

This method has a nice stability domain shown in Figure 4.3 and damping characteristics shown in Figure 4.4. This error method is not L–stable but has a large *asymptotic region* and should be well suited for step–size control when solving stiff systems. In Figure 4.4a, the asymptotic region is the area near the origin where $\hat{\sigma}_d$ follows σ_d until about $\sigma_d = 3.4$ [2].

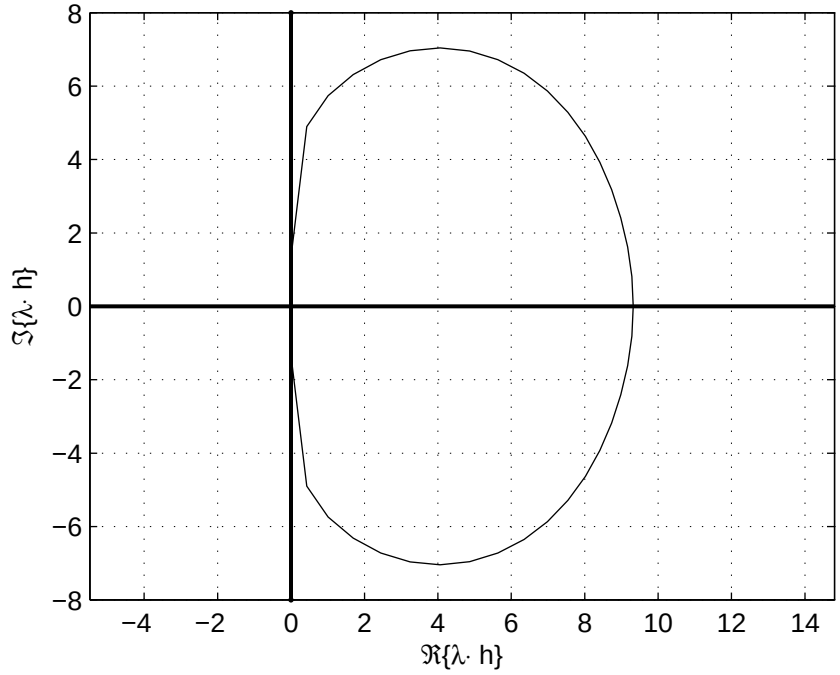

FIGURE 4.3. Stability domain of Lobatto IIIC(6) error method

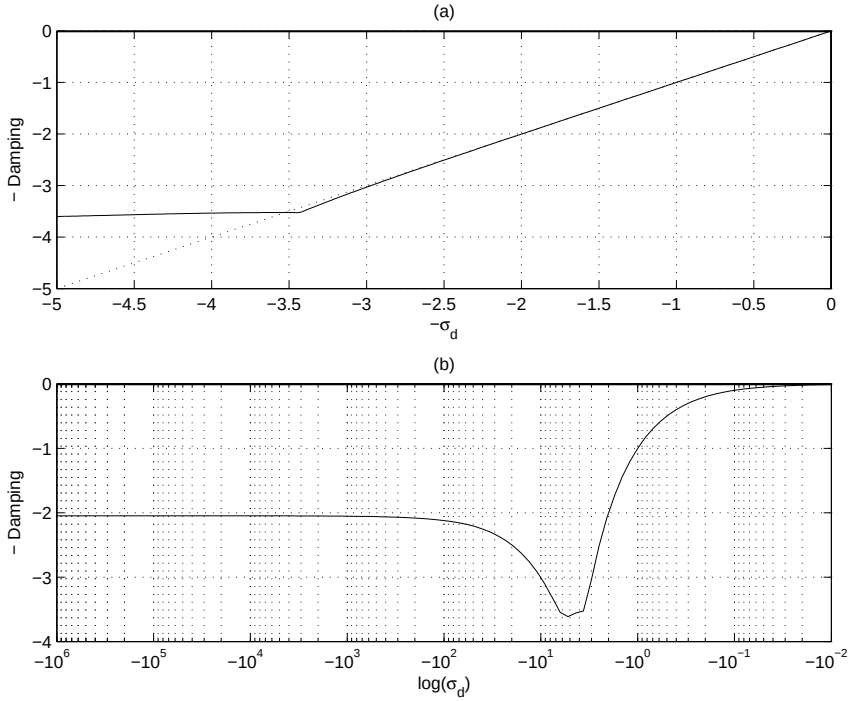

FIGURE 4.4. Lobatto IIIC(6) error method: a) Damping plot; b) Logarithmic damping plot

Chapter 5

Numerical Experiments

In this chapter, the different integration algorithms and respective error methods will be evaluated using selected test ODEs found in Appendix A. These ODEs were selected, along with considerations from [17], from a larger set suggested by [12]. For reference, the solution of these ODEs have been plotted in Figures 5.1– 5.8. These solutions were all found using the MATLAB command `ode15s`. This command invokes a stiff system solver based on Numerical Differential Formulae (NDF) which are closely related to the BDF techniques [18].

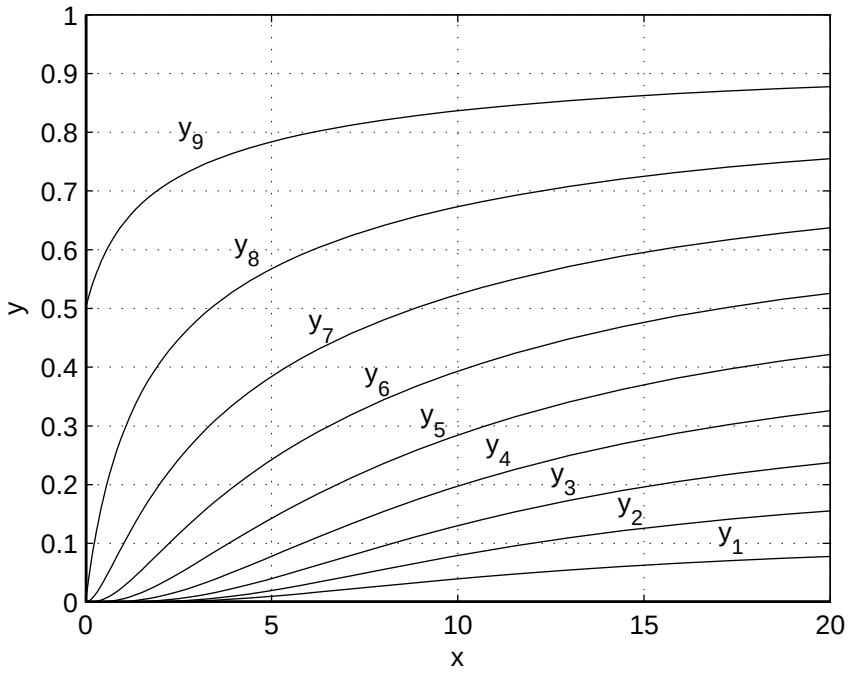

Figure 5.1. ODE set A solution

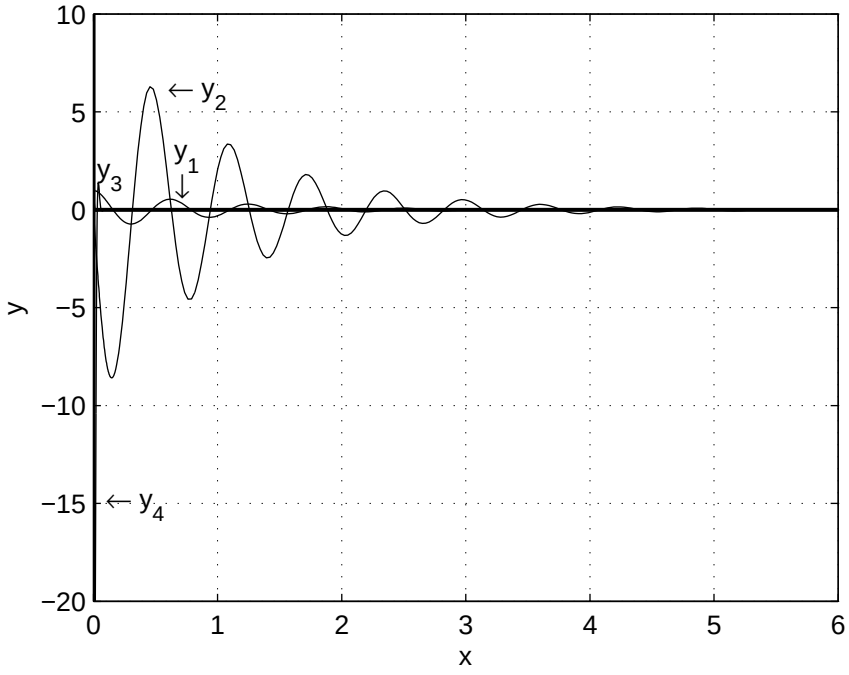

FIGURE 5.2. ODE set B solution

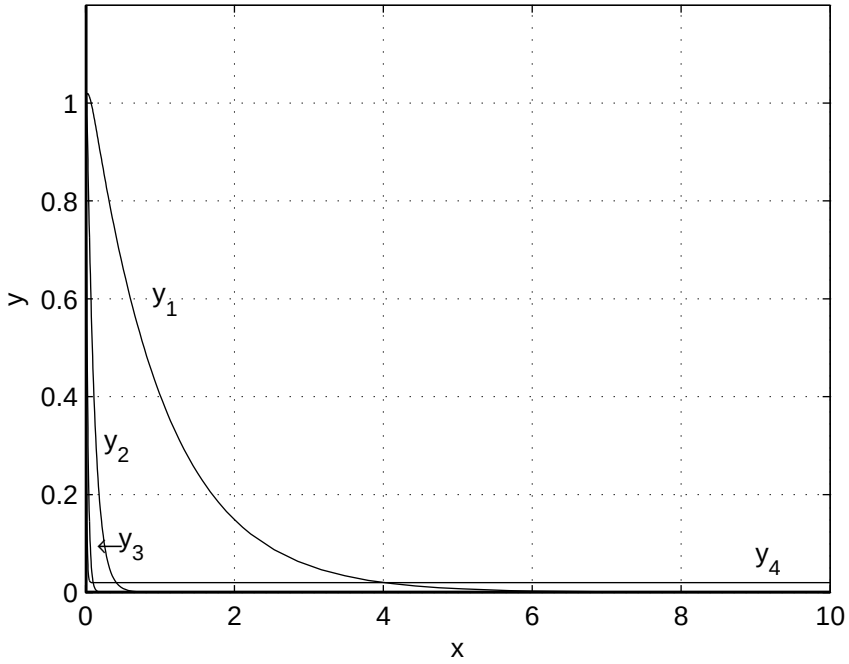

FIGURE 5.3. ODE set C solution

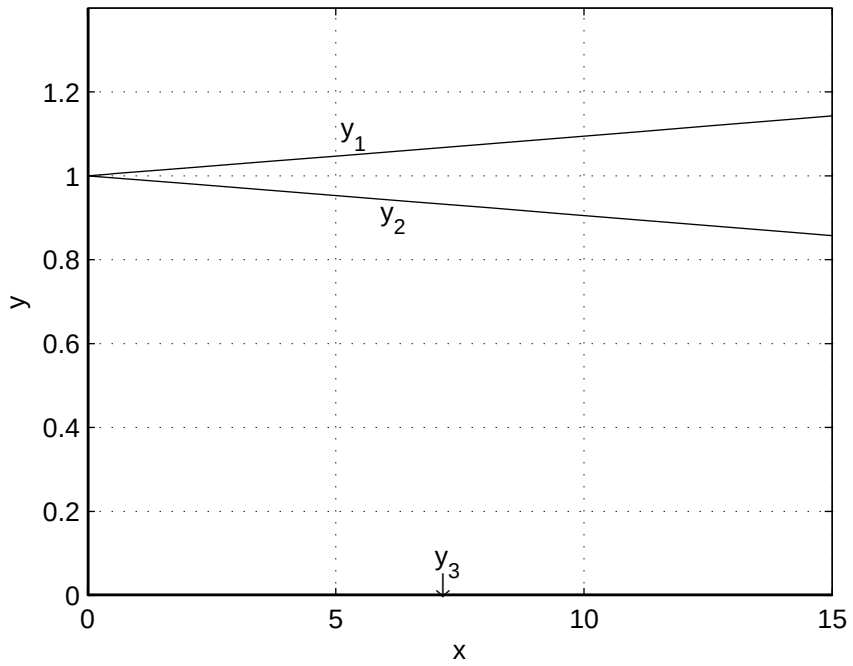

FIGURE 5.4. ODE set D solution

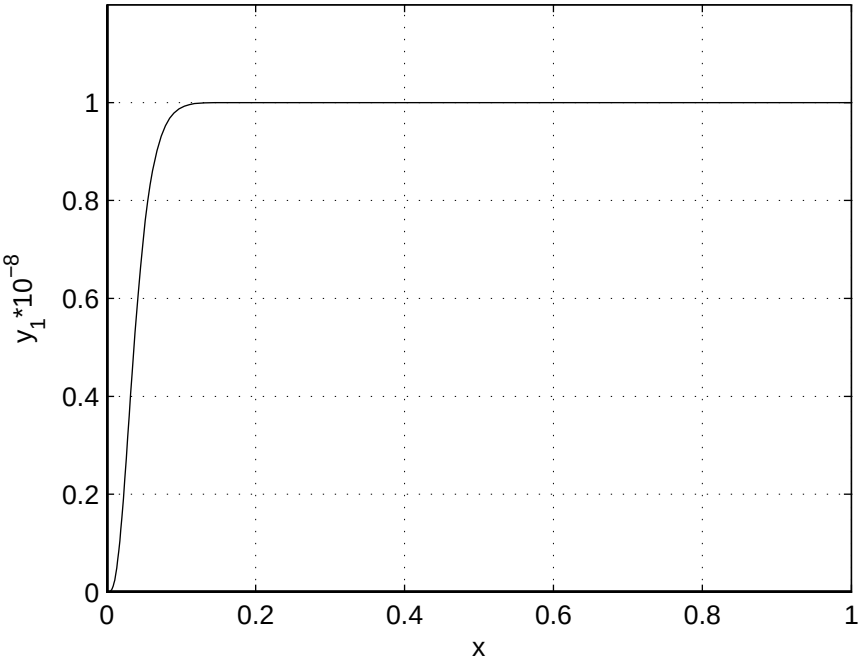

FIGURE 5.5. ODE set E y_1 solution

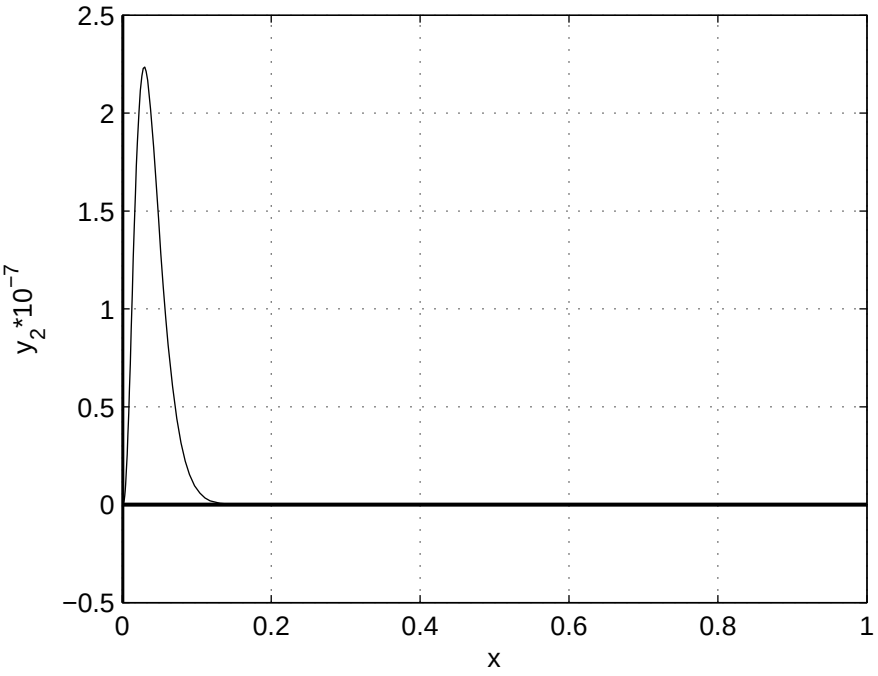

FIGURE 5.6. ODE set E y_2 solution

5.1 Implementation

The methods presented in the previous chapters have been implemented using the Modelica language [21]. One software package that implements Modelica is called Dymola [22]. Due to the large number of difference equations resulting after inlining, sorting and tearing operations can no longer be done by hand. Since Modelica is an acausal language, Dymola can perform automated sorting and tearing. An example Rad3 implementation in pseudo–Modelica code is shown in Listing 5.1. As an example, the implementation of ODE set A inlined with Rad5 used can be found in Appendix B. The Modelica implementations are similar for the various algorithms. After inlining Dymola can then just loop over the model equations to find a solution.

LISTING 5.1. Inlining Rad3 in pseudo-Modelica

```
model inlinerad3
  parameter Real <parameters: tol, initial step size>
  constant Real  <Time instants>
  constant Real  <Radau IIA(3) coefficients>
  constant Real  <Step size control coefficients>

  //variable for states and state derivatives
  Real <stage1 variables with initial conditions>
  Real <stage2 variables with initial conditions>
  Real <variable history>
algorithm
  when time >= pre(NextSampling) then
    compute stage1 state derivatives
  end when;
  when time >= pre(NextSampling2) then
    compute stage2 state derivatives
    compute stage1 states
    compute stage2 states
    compute error, new step size
    store state history
  end when;
end inlinerad3;
```

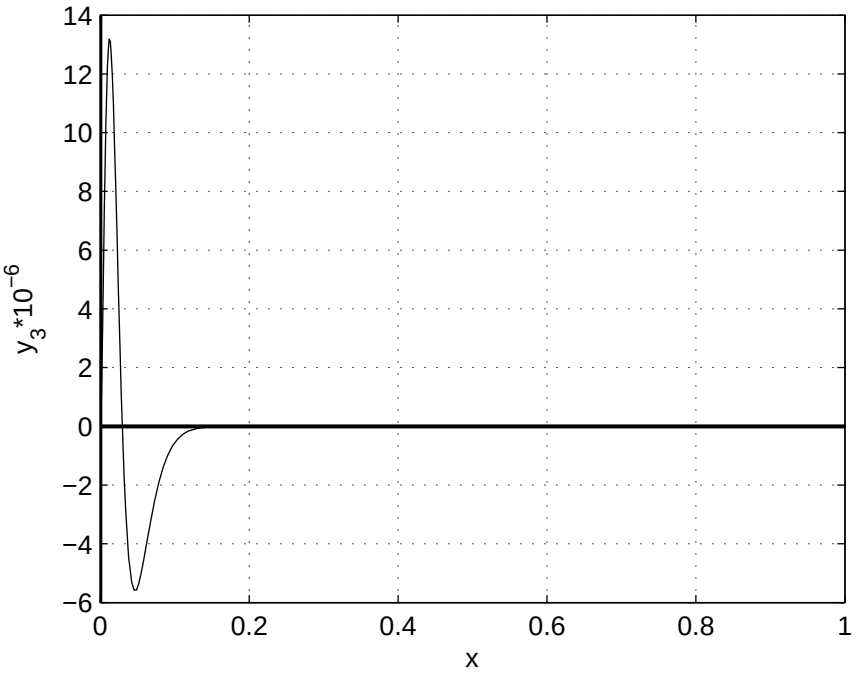

FIGURE 5.7. ODE set E y_3 solution

5.2 Simulation Results

Shown in Figures 5.9–5.14 are the simulation results for ODE set A. Each of the various algorithms with associated error estimate have been inlined with this ODE set. For this set, the Radau algorithms reach a step size of about 8×10^{-4} before constantly changing. One drawback to this implementation is that rejected steps are not repeated but propagated to the next step as the integration continues. Despite having data with a large error propagated, the Rad3 algorithm is able to continue. For the Lobatto family, the solutions are not correct when compared with the accurately computed solution of Figure 5.1 but the error estimate stays fixed and the step size remains small. The solution with the original embedding method of HW-SDIRK determines a step size of about 8×10^{-4} and stays unchanged until the end of the simulation. Using the alternate embedding method causes a slightly larger step size, on average, of $8.7\times$

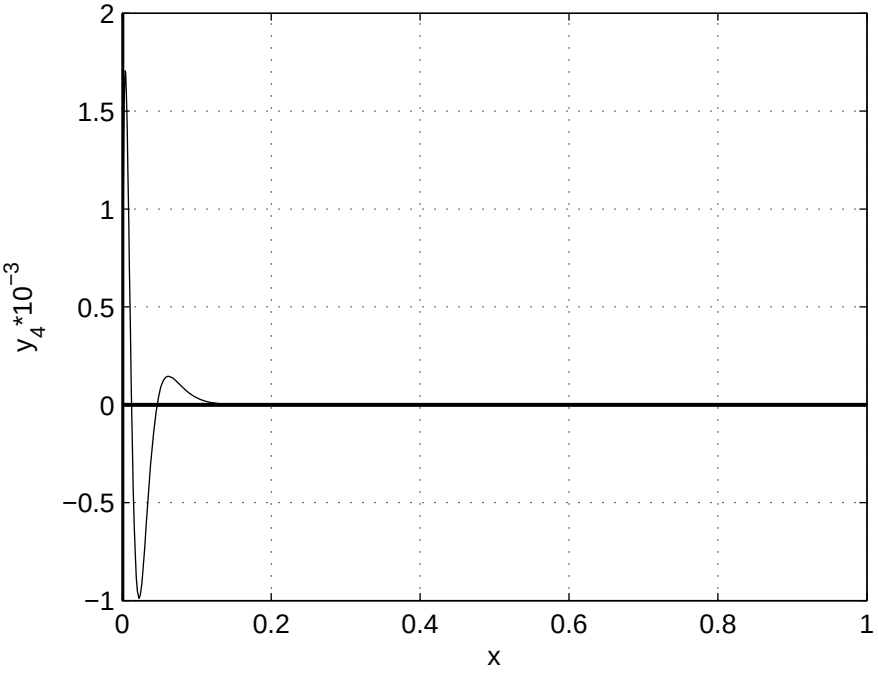

FIGURE 5.8. ODE set E y_4 solution

10^{-4} to be selected. The alternate error method allows more aggressive step sizes to be chosen but in this implementation has a hard time keeping the error near the specified tolerance. The errors in the resulting trajectory of y_9 are apparent when comparing Figure 5.14 with that of the accurately computed solution shown in Figure 5.1. The CPU-time needed using an Intel Pentium–M[1] processor running at 1.8 GHz and the total number of steps computed for the various algorithms are summarized in Table 5.1. The data given for `ode15s` is only for reference and cannot be used for direct comparison. One difference is that the implicit Runge–Kutta algorithms are single–step methods while `ode15s` is a multi–step method. Another difference is that `ode15s` offers a mature and optimized implementation while additional development and optimization needs to be performed on the still immature implementation of IRKs and associated error methods presented.

[1]with SpeedStep disabled

The solutions for ODE set B for the various algorithms are shown in Figures 5.15–5.20. Again the trajectories produced by the Lobatto family are incorrect. Both error estimates for the Radau solutions stay near the specified tolerance of 10^{-5} and the step size shrinks before the error grows too large. Rad3 completes the simulation using 13,737 steps while Rad5 needs 35,811 steps. Using the included embedding method of HW-SDIRK, one of the trajectories is oscillating even though the estimated error stays around 10^{-3} after the initial spike. This could suggest that the explicit method is not stable when solving this problem. The errors produced by HW-SDIRK with the alternate error estimate are similar to those produced by the Radau algorithms in that these three algorithms keep the error near the specified tolerance of 10^{-3}. With the alternate embedding method the step size for HW-SDIRK is chosen more appropriately.

ODE set C solutions are shown in Figures 5.21–5.26. Yet again the trajectories produced by the Lobatto algorithms are incorrect. Both Radau solutions run into a bit of trouble as the step size grows too large. The problem is caused by the restriction that the step size not change too dramatically. By the time the step size drops sufficiently, too many steps have occurred and the data being propagated during the subsequent steps is completely incorrect such that a different problem is now being solved. HW-SDIRK with the included embedding method has similar problems but is able to recover without completely changing the trajectories. For this ODE set, HWSDIRK with the alternate error method is the winner when looking only at the solution trajectories. Evidently, the error estimate became equal in magnitude to the specified tolerance, so the step size remained small and unchanged for a longer period than in the other algorithms.

Interesting results, shown in Figures 5.27–5.32, are produced for ODE set D. When inlined with Rad3, the errors cause the step size to shrink but in this implementation Rad3 and the error estimate cannot recover from bad data being propagated. The errors become so bad that the integration terminates itself. On the other hand, Rad5

is able to solve this system while the estimated error changes wildly between 10^{-4} and 10^{-6}. The Rad5 algorithm tries to keep the step size around 5×10^{-4}. Surprisingly, without working for any of the other test problems, the Lobatto algorithms are able to solve this problem. Perhaps this is only because there is little change, in this case less than ± 0.2, from the initial value to the final values of y_1 and y_2. Both Lobatto algorithms try to keep the step size around 6.4×10^{-4}.

As noted in [17], this problem is badly scaled, so the trajectories are plotted individually and shown in Figures 5.33–5.44. Inlining doesn't seem to mind this ill–posed problem and even for $tol = 10^{-6}$ the resulting trajectories still resemble the accurately computed trajectories of Figures 5.5–5.8. The relative magnitudes for each of the solutions are $|y_1| \leq 10^{-8}$, $|y_2| \leq 3 \times 10^{-7}$, $|y_3| \leq 2 \times 10^{-5}$, $|y_4| \leq 10^{-3}$. Again, the Lobatto algorithms incorrectly compute the trajectories, but this time HW-SDIRK with the alternate error method allows for a step size larger than that of Rad5 as seen in Figures 5.35 and 5.44. Even though a bigger step was taken, Rad5 completes this problem using 933 steps whereas HW-SDIRK and the alternate error method needs 9,514 steps. For improved trajectory results, the specified tolerance could be increased to at least 10^{-9}, one order of magnitude smaller than the smallest component, and rejected integration steps must not be propagated to the next step. In this test case, it is also evident that limiting the step size to $h/2$ causes problems since the integration algorithms cannot keep up with the dynamics of the system.

TABLE 5.1. Cost Summary

ODE set	Algorithm	CPU Time (sec)	Total Steps
A	ode15s	0.04	86
	Rad3	1.29	30110
	Rad5	4.68	63250
	Lob4	4.43	55970
	Lob6	8.62	68034
	HW-SDIRK	5.03	25354
	HW-SDIRK w/alt. error	5	22893
B	ode15s	0.12	346
	Rad3	0.551	13737
	Rad5	2.32	35811
	Lob4	12	175817
	Lob6	6.78	77156
	HW-SDIRK	0.24	1104
	HW-SDIRK w/alt. error	48.6	396890
C	ode15s	0.04	124
	Rad3	1.84	45264
	Rad5	0.751	11350
	Lob4	2.08	30441
	Lob6	6.25	68364
	HW-SDIRK	0.16	627
	HW-SDIRK w/alt. error	8.09	73311
D	ode15s	0.04	24
	Rad3	0.2	1861
	Rad5	2.05	30501
	Lob4	0.981	15910
	Lob6	1.58	16818
	HW-SDIRK	3.37	32252
	HW-SDIRK w/alt. error	9.85	93748
E	ode15s	0.05	66
	Rad3	0.08	665
	Rad5	0.15	933
	Lob4	0.341	3051
	Lob6	0.701	6102
	HW-SDIRK	0.17	442
	HW-SDIRK w/alt. error	1.11	9514

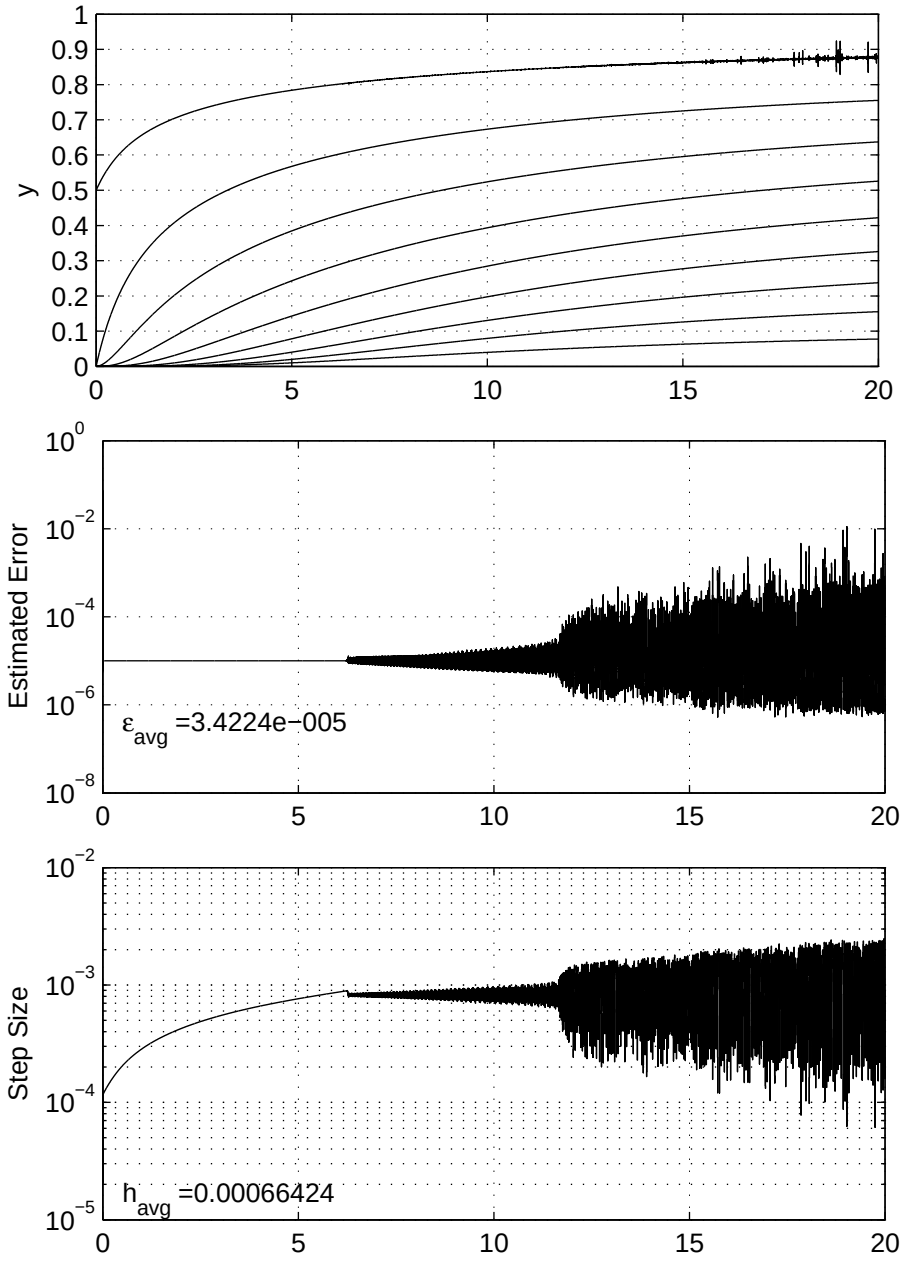

FIGURE 5.9. ODE set A inlined with Rad3

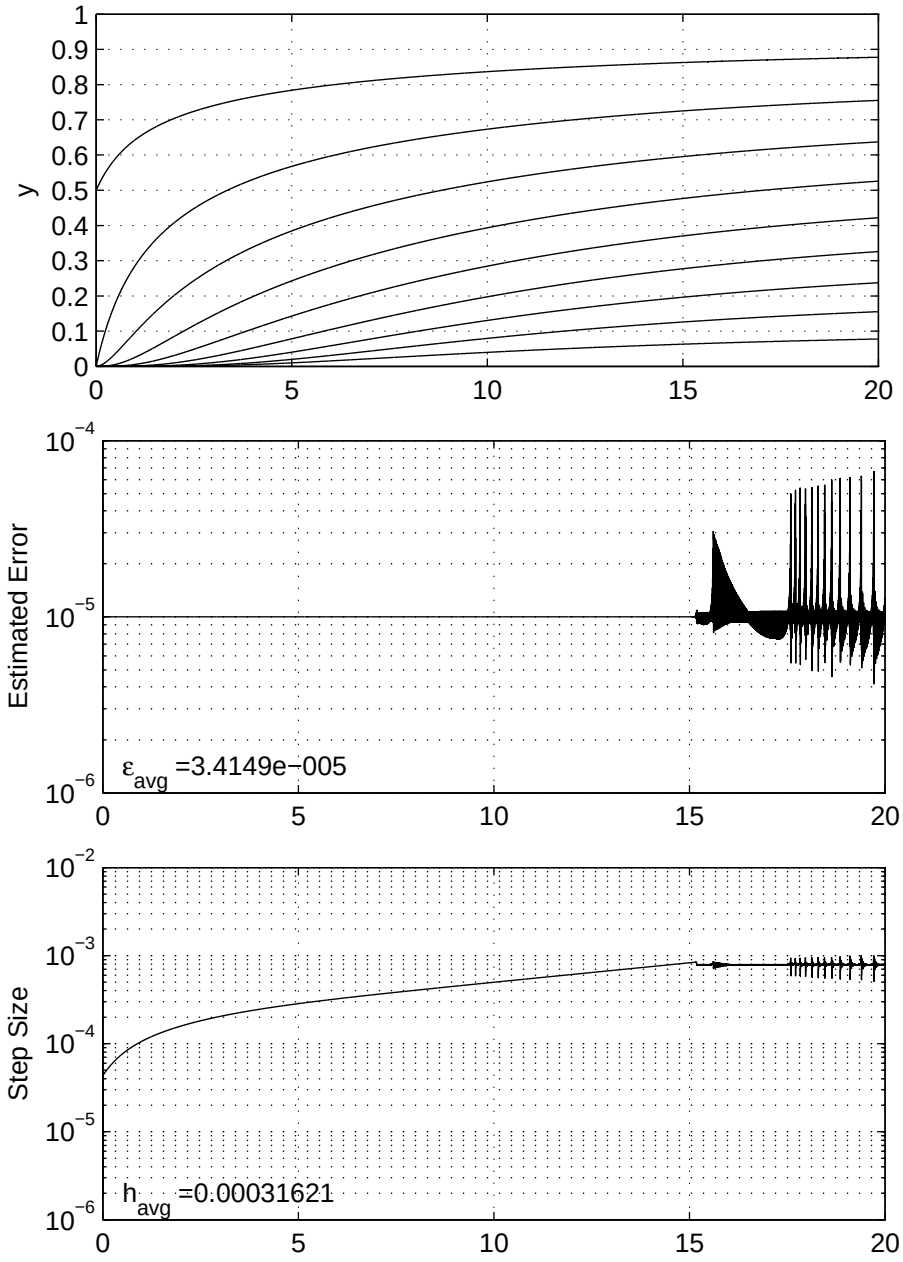

FIGURE 5.10. ODE set A inlined with Rad5

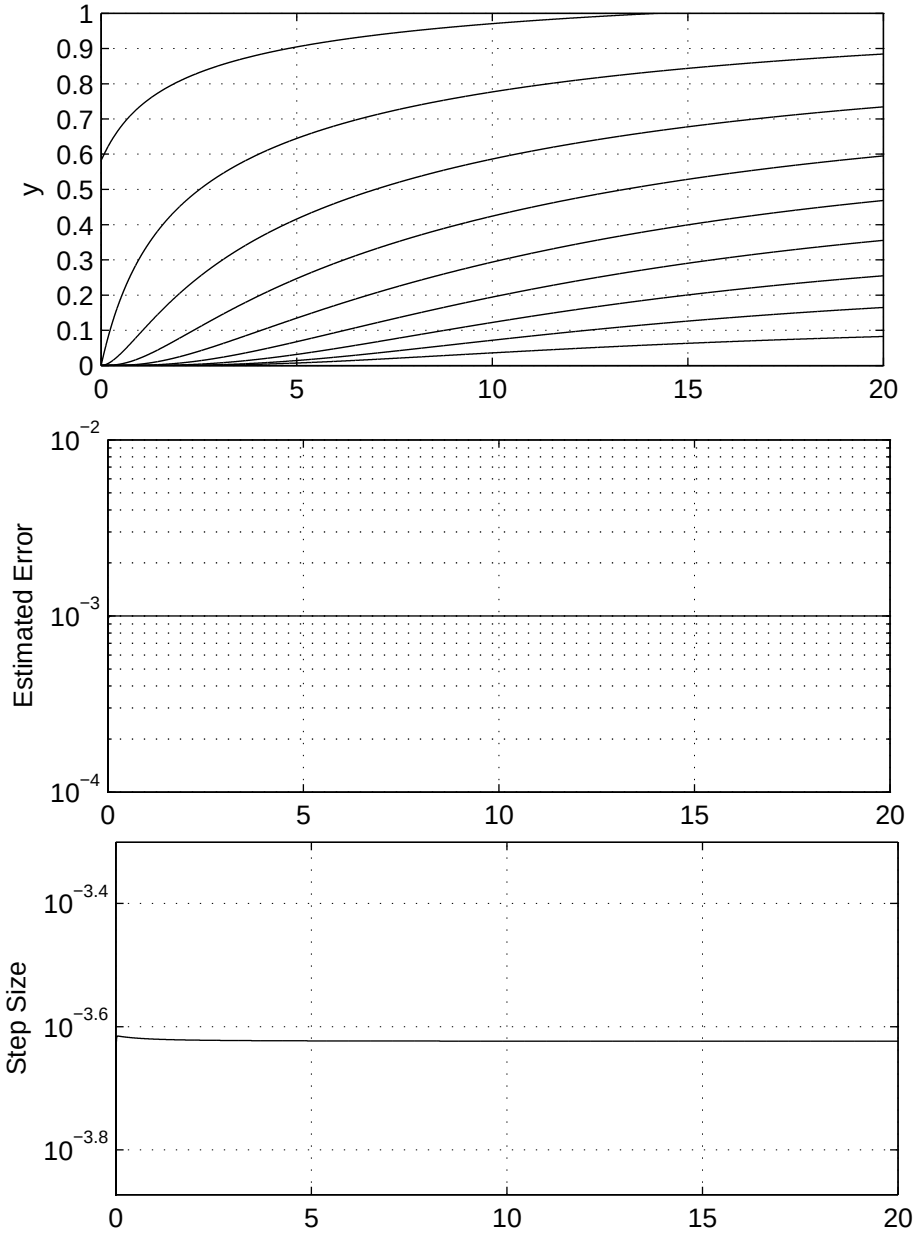

FIGURE 5.11. ODE set A inlined with Lob4

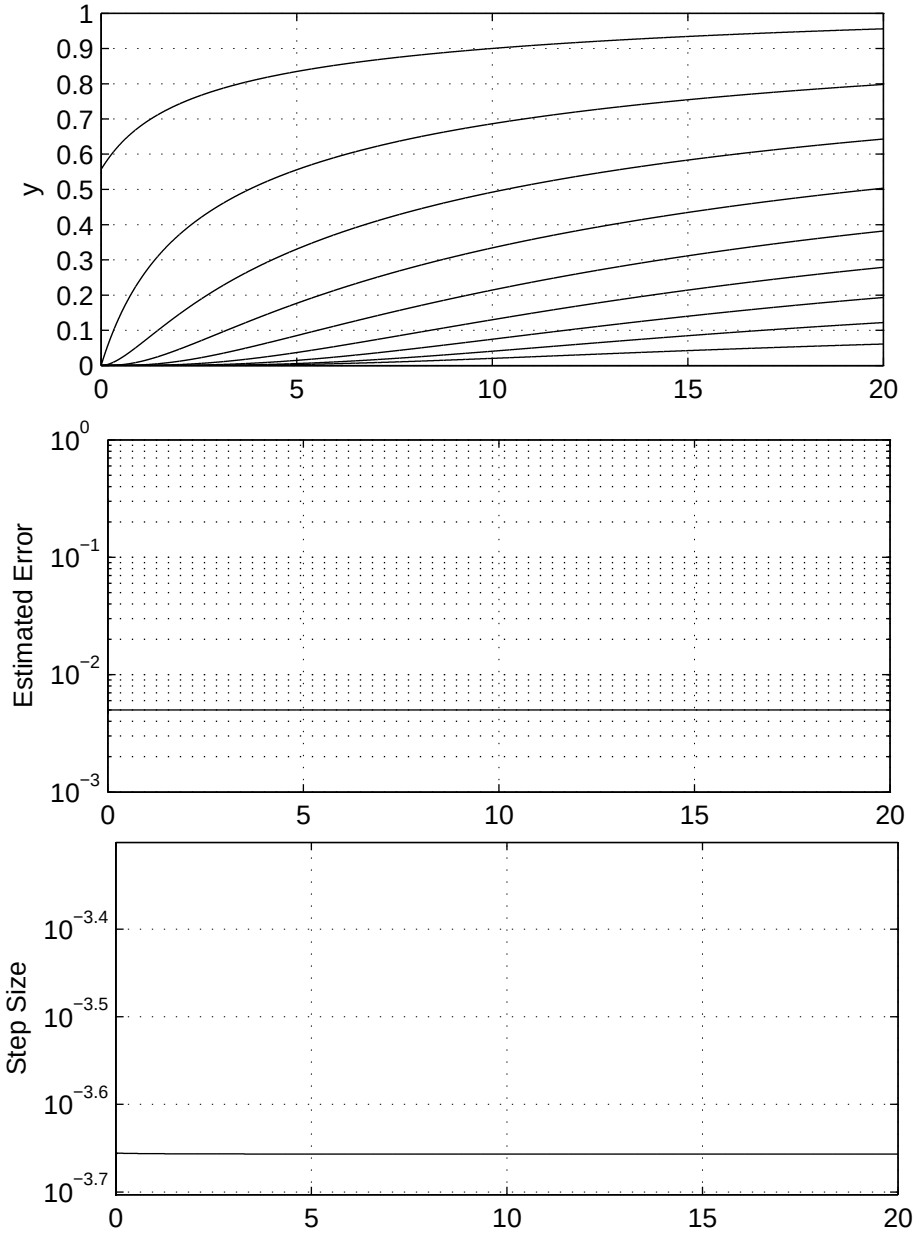

FIGURE 5.12. ODE set A inlined with Lob6

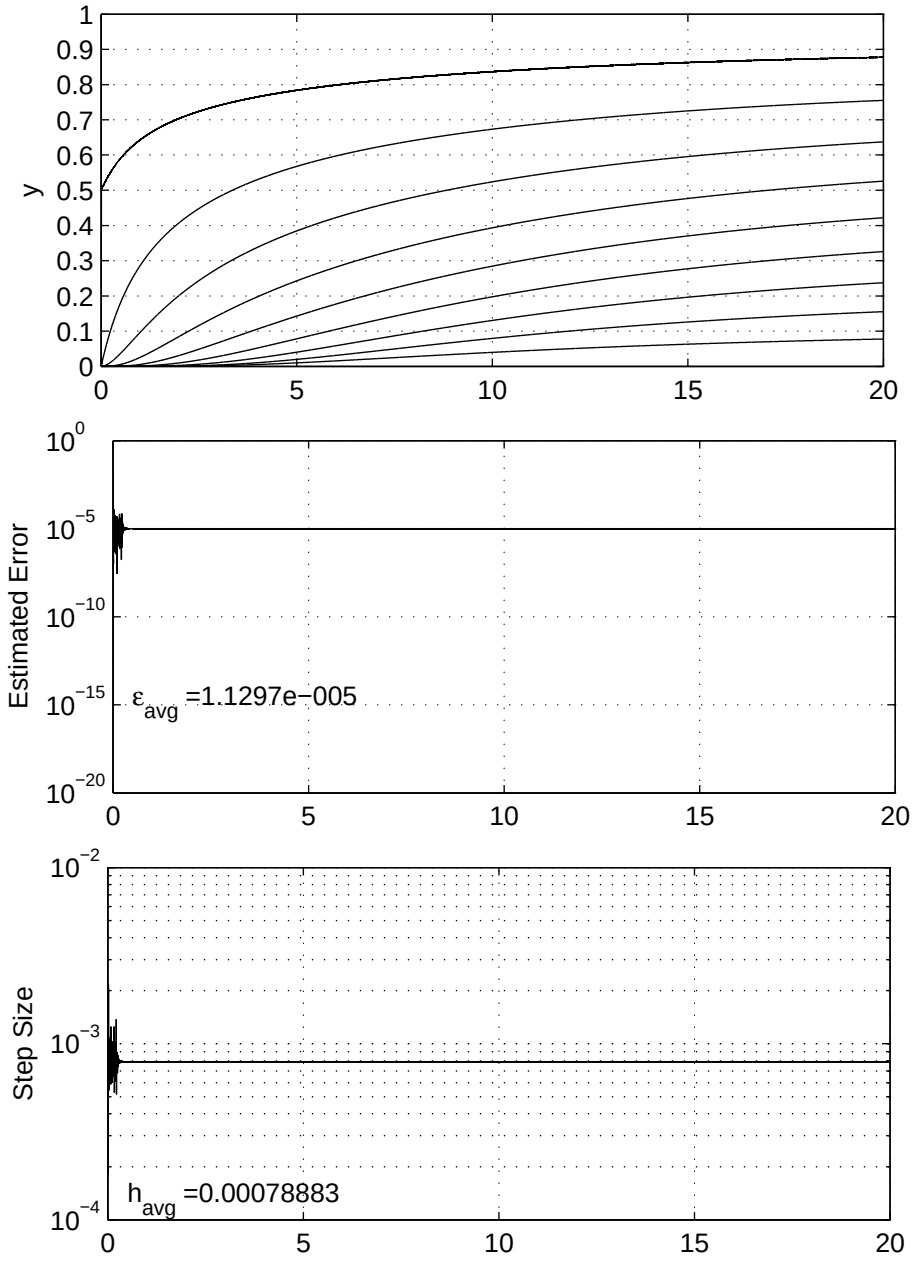

FIGURE 5.13. ODE set A inlined with HW-SDIRK

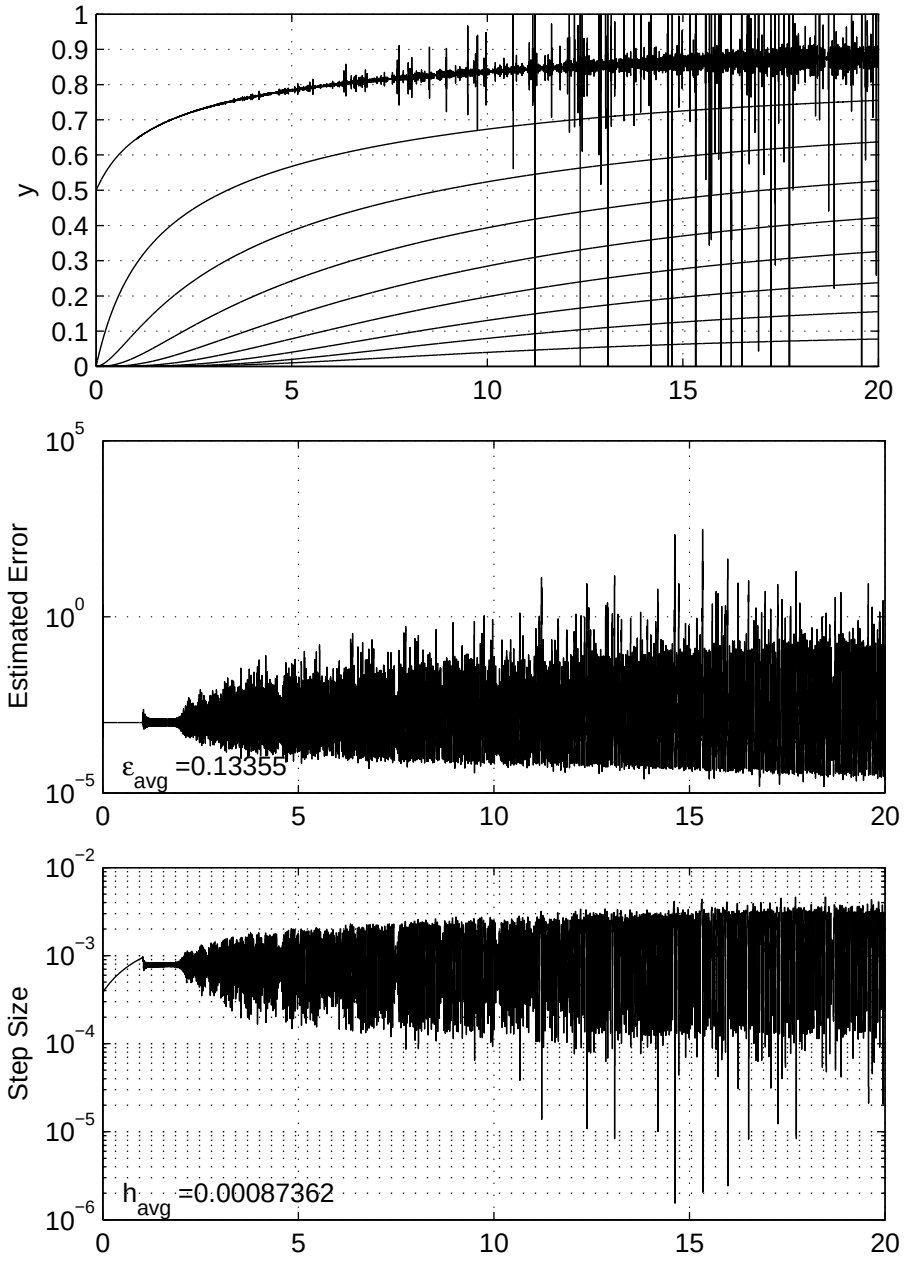

FIGURE 5.14. ODE set A inlined with HW-SDIRK and alternate error method

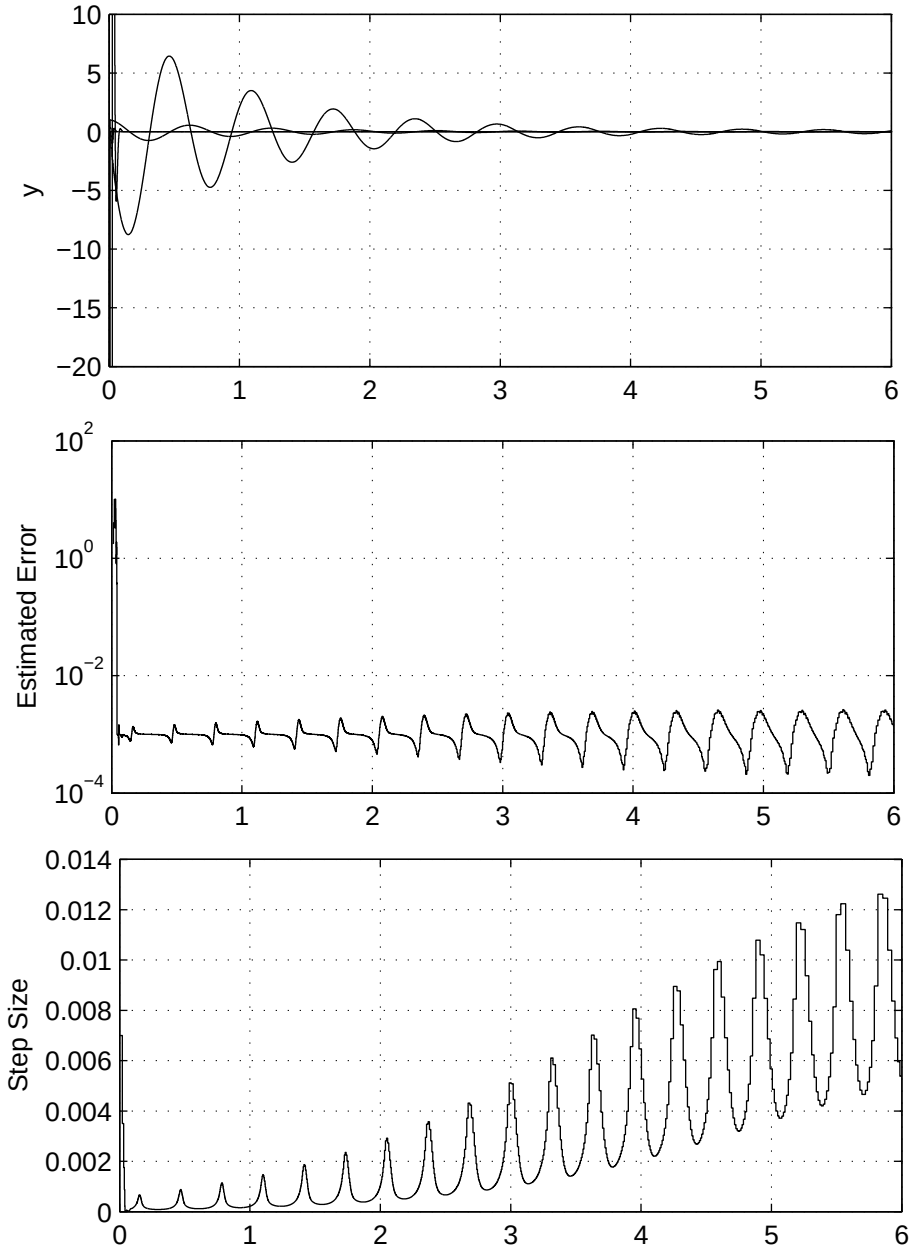

FIGURE 5.15. ODE set B inlined with Rad3

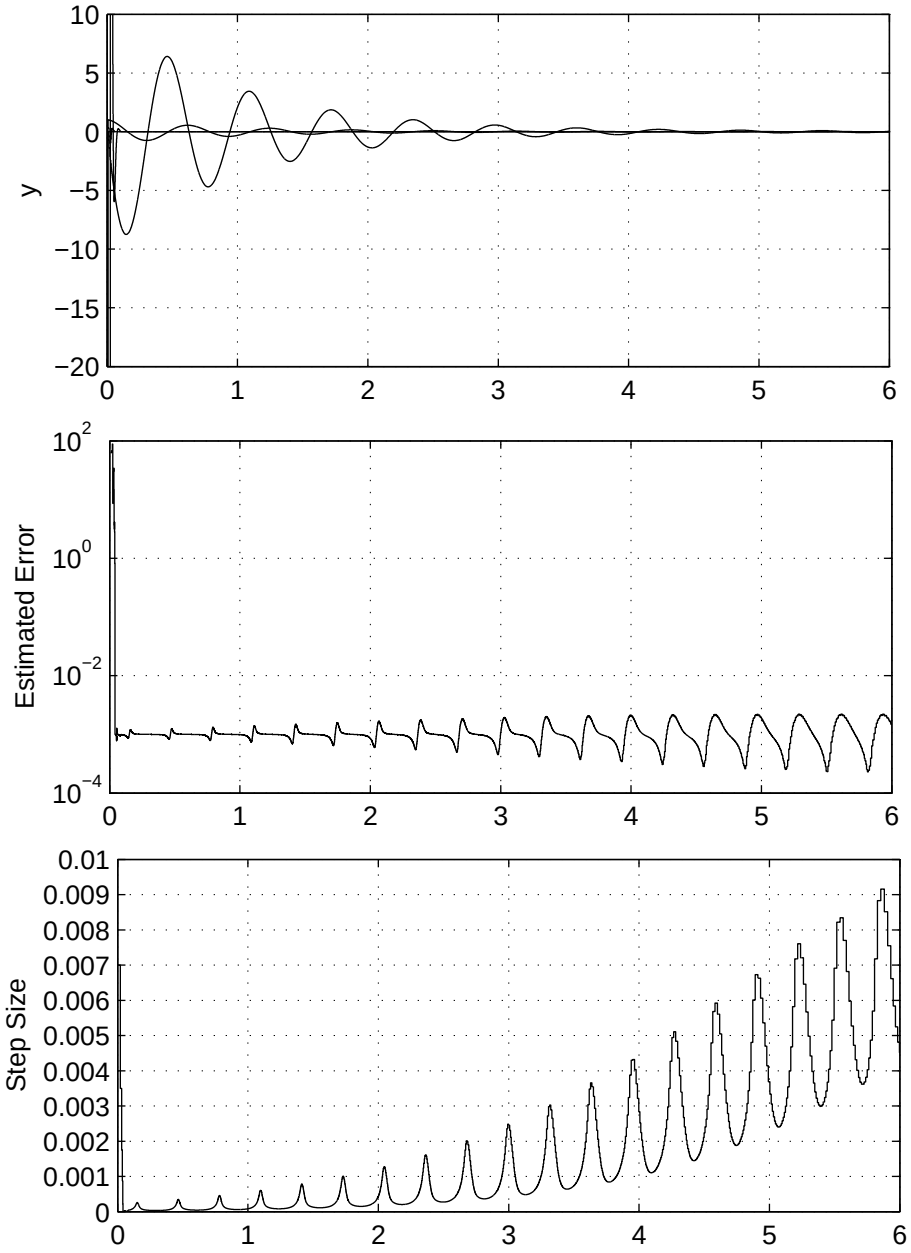

FIGURE 5.16. ODE set B inlined with Rad5

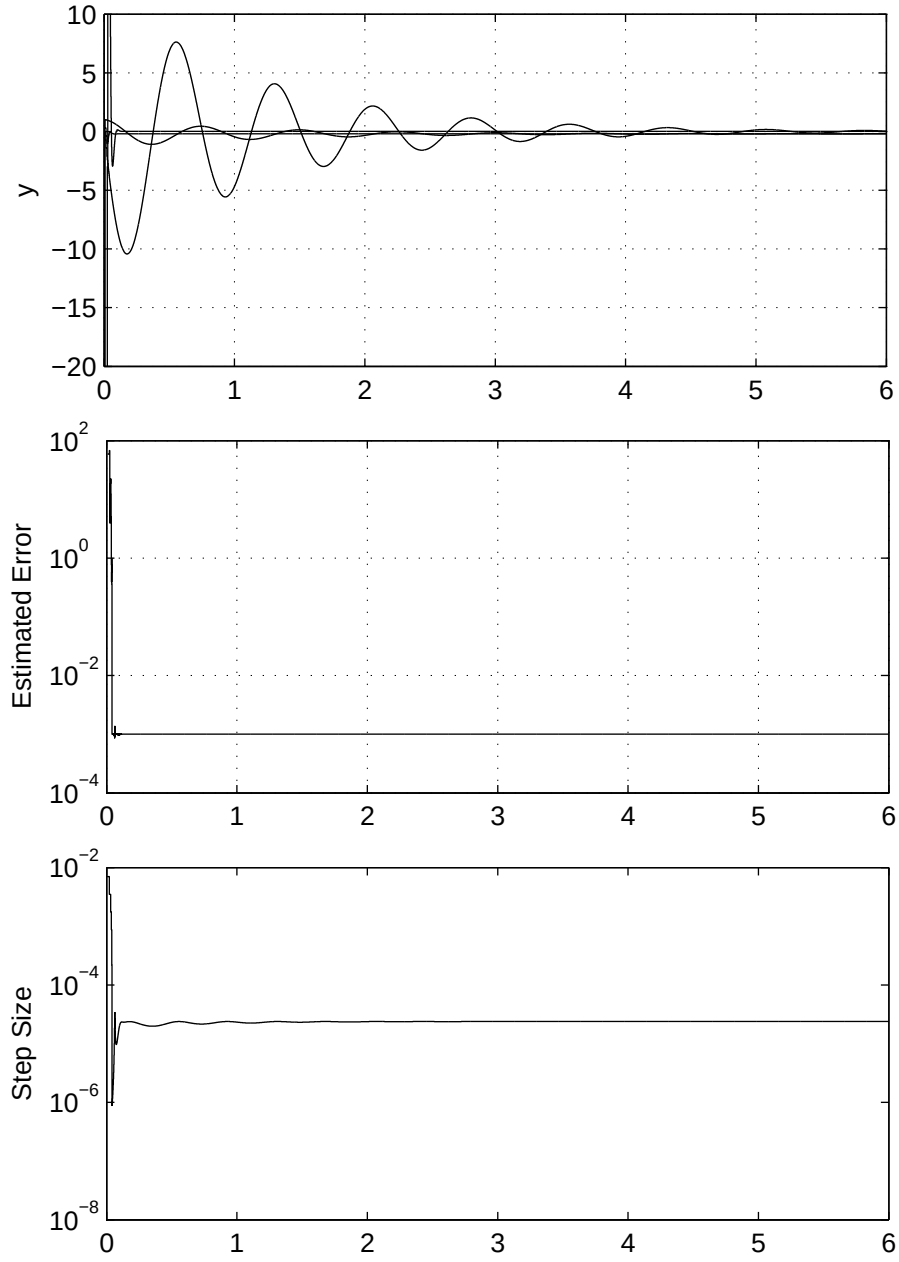

FIGURE 5.17. ODE set B inlined with Lob4

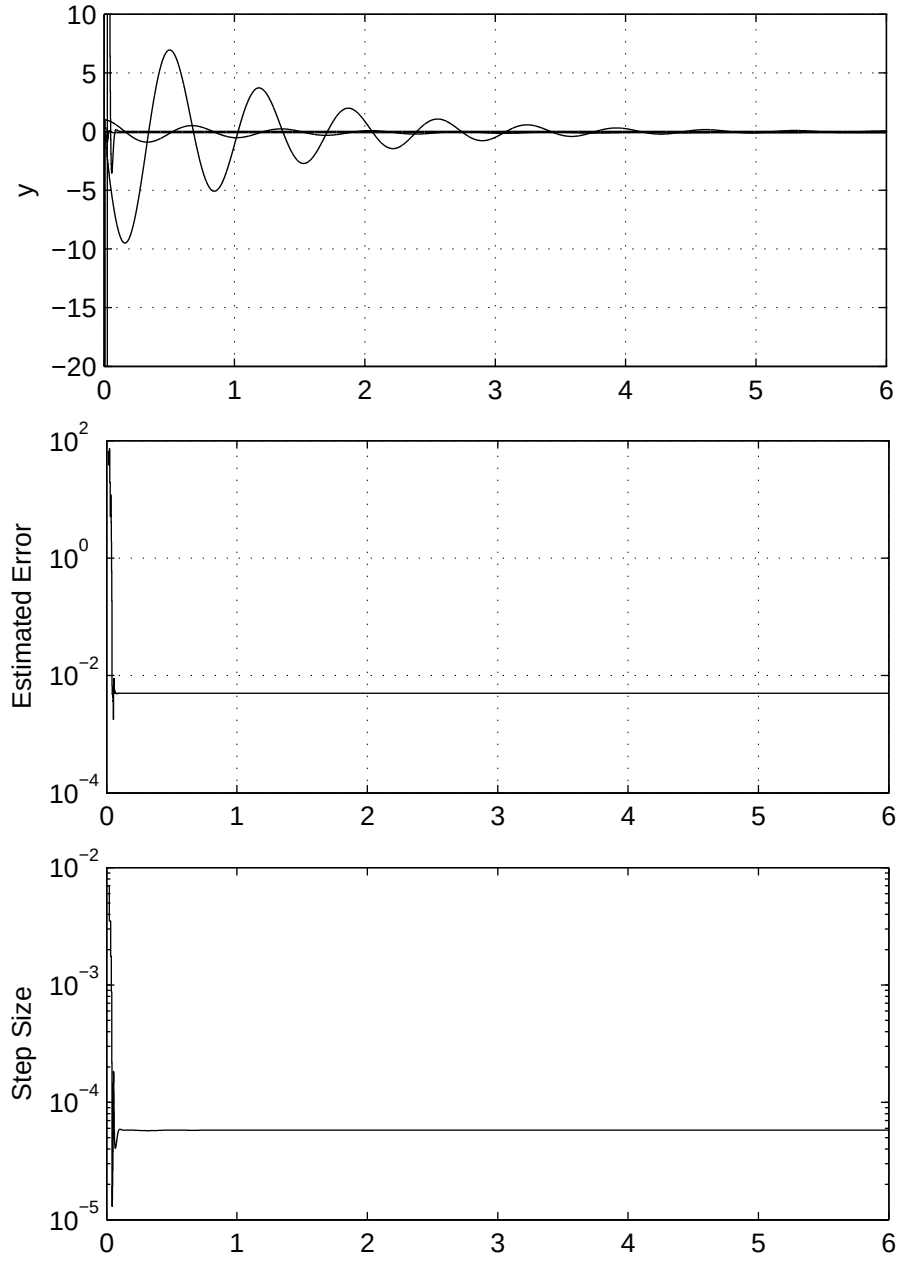

FIGURE 5.18. ODE set B inlined with Lob6

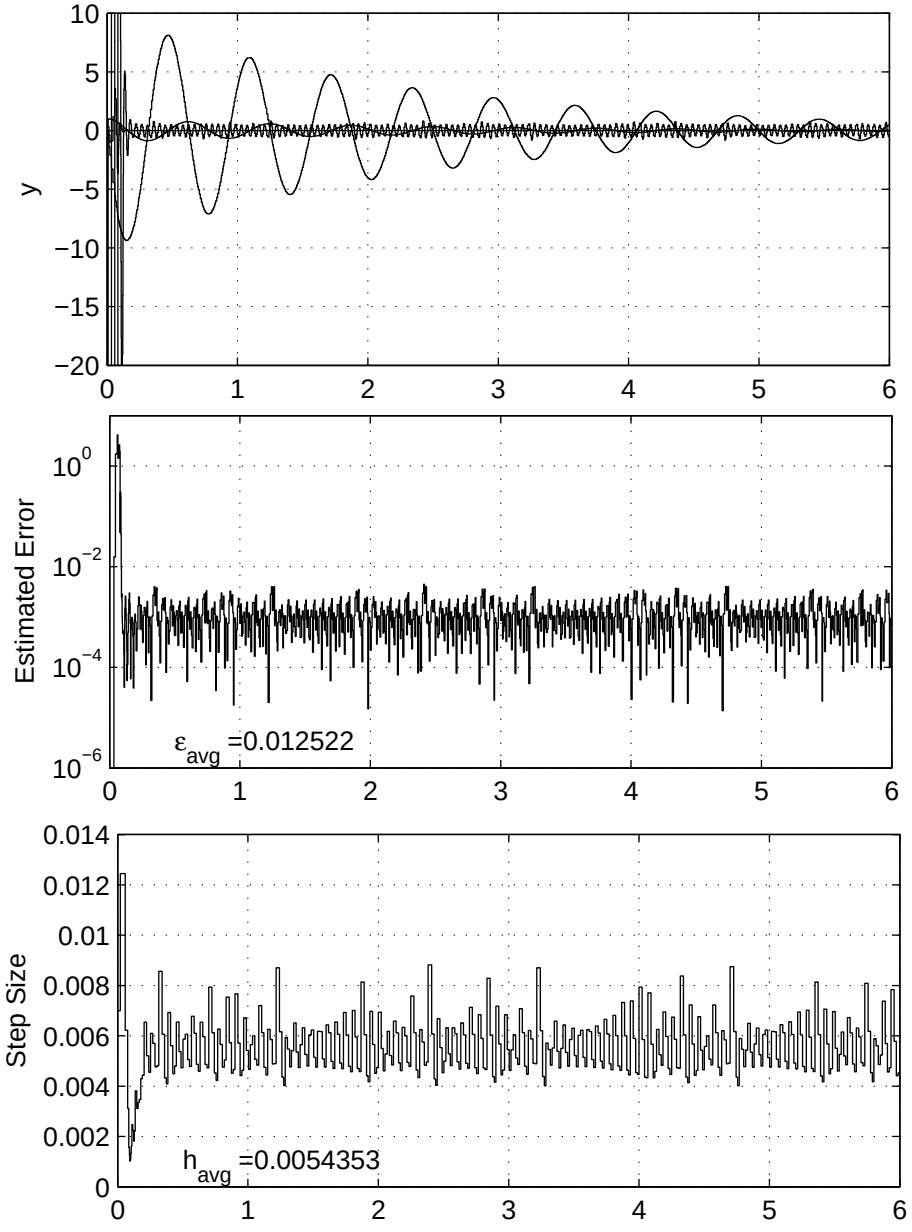

FIGURE 5.19. ODE set B inlined with HW-SDIRK

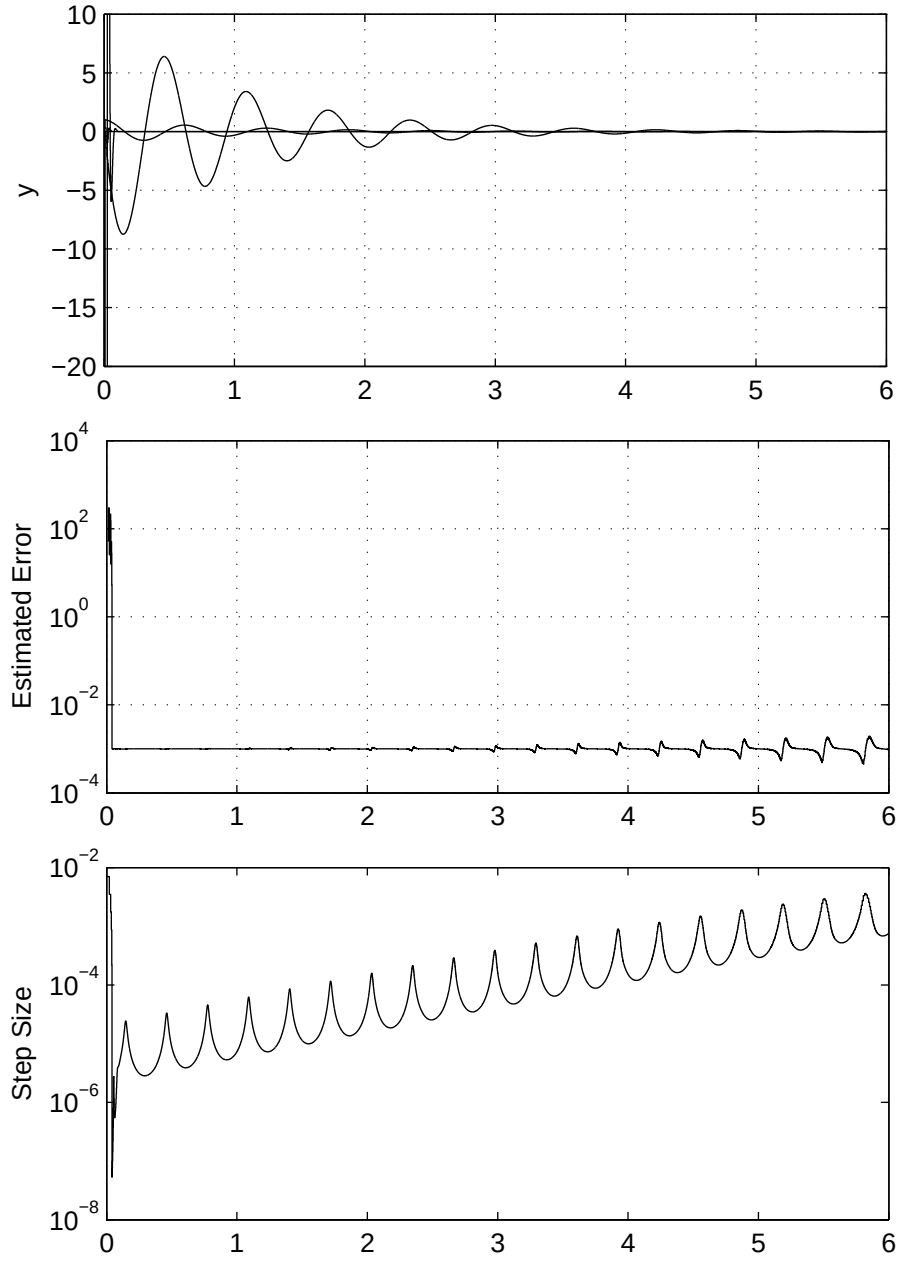

FIGURE 5.20. ODE set B inlined with HW-SDIRK and alternate error method

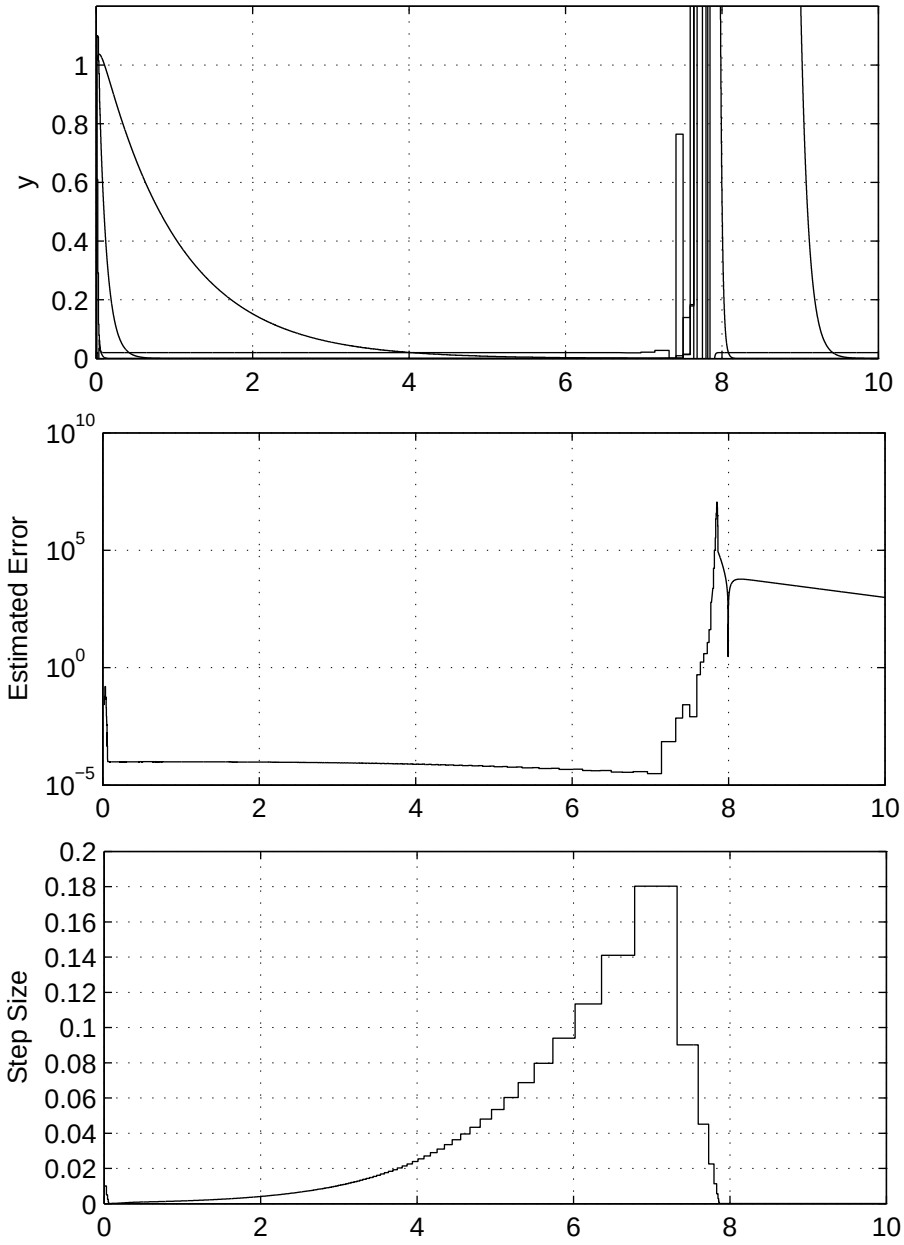

FIGURE 5.21. ODE set C inlined with Rad3

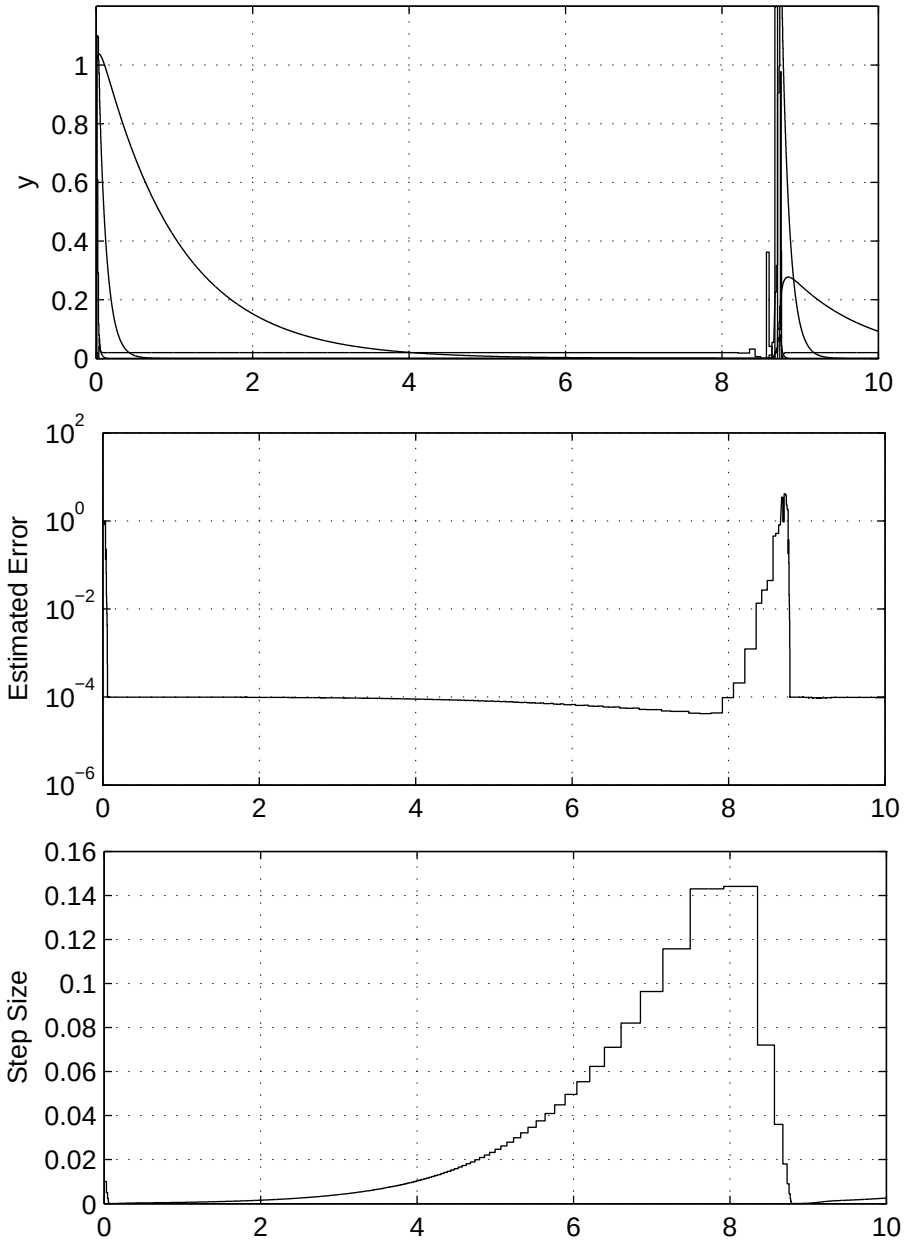

FIGURE 5.22. ODE set C inlined with Rad5

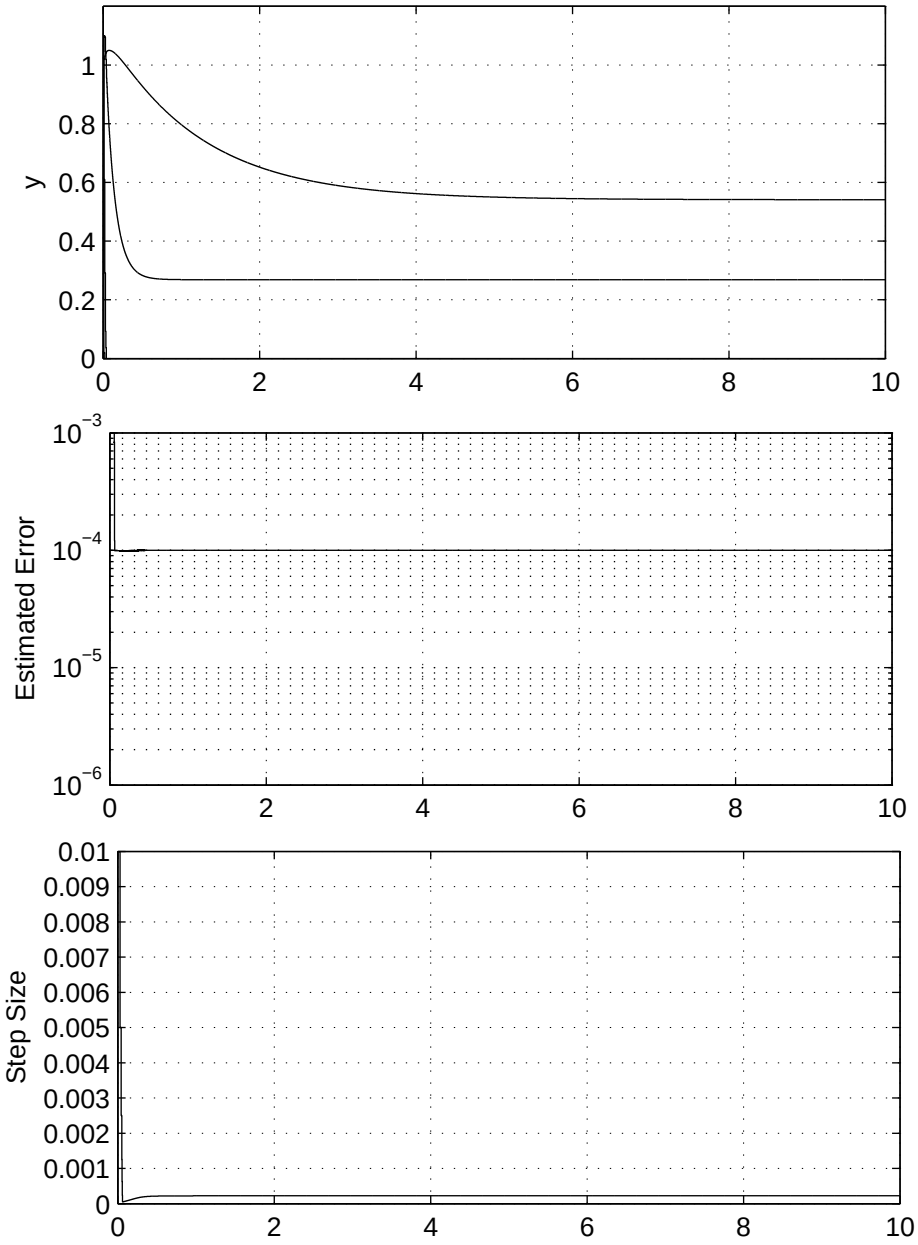

FIGURE 5.23. ODE set C inlined with Lob4

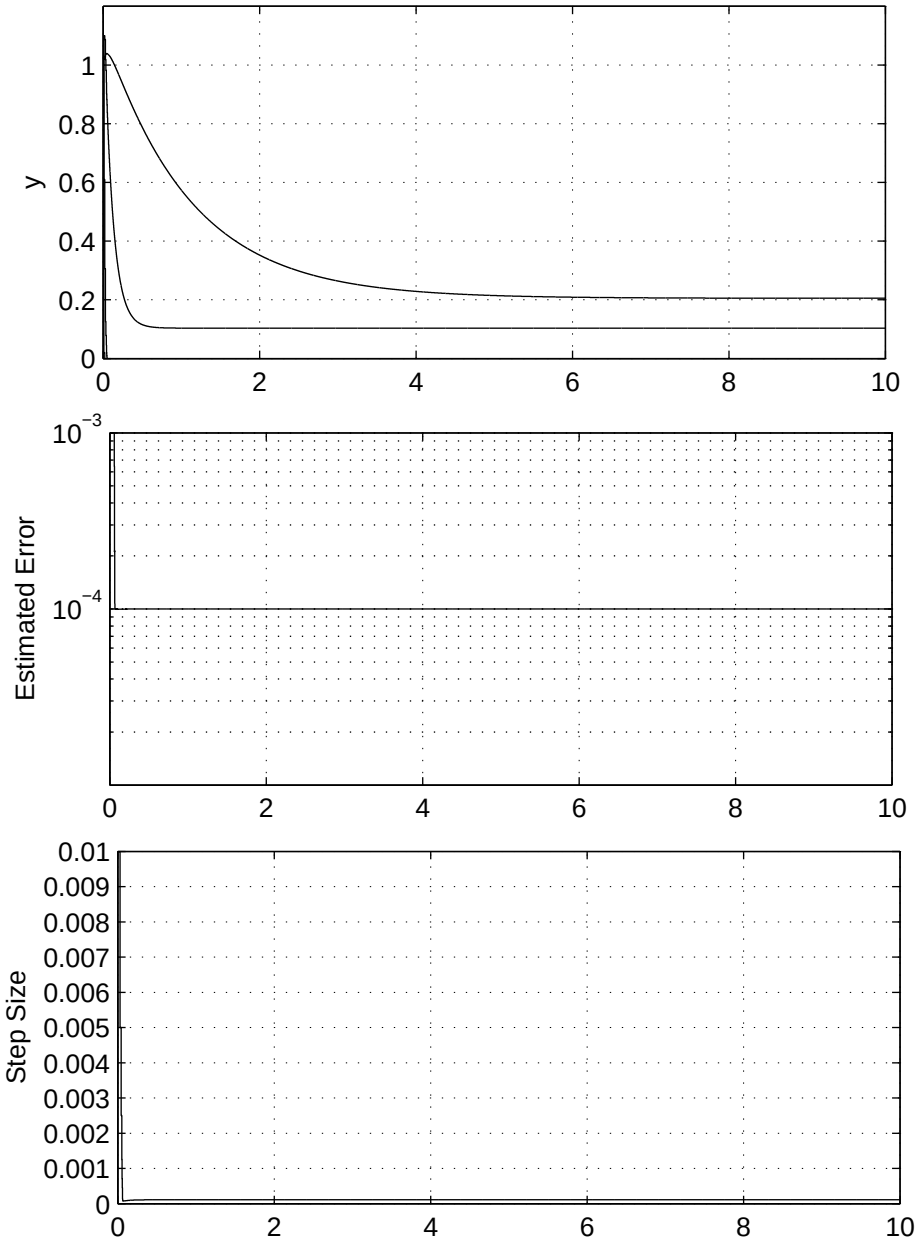

FIGURE 5.24. ODE set C inlined with Lob6

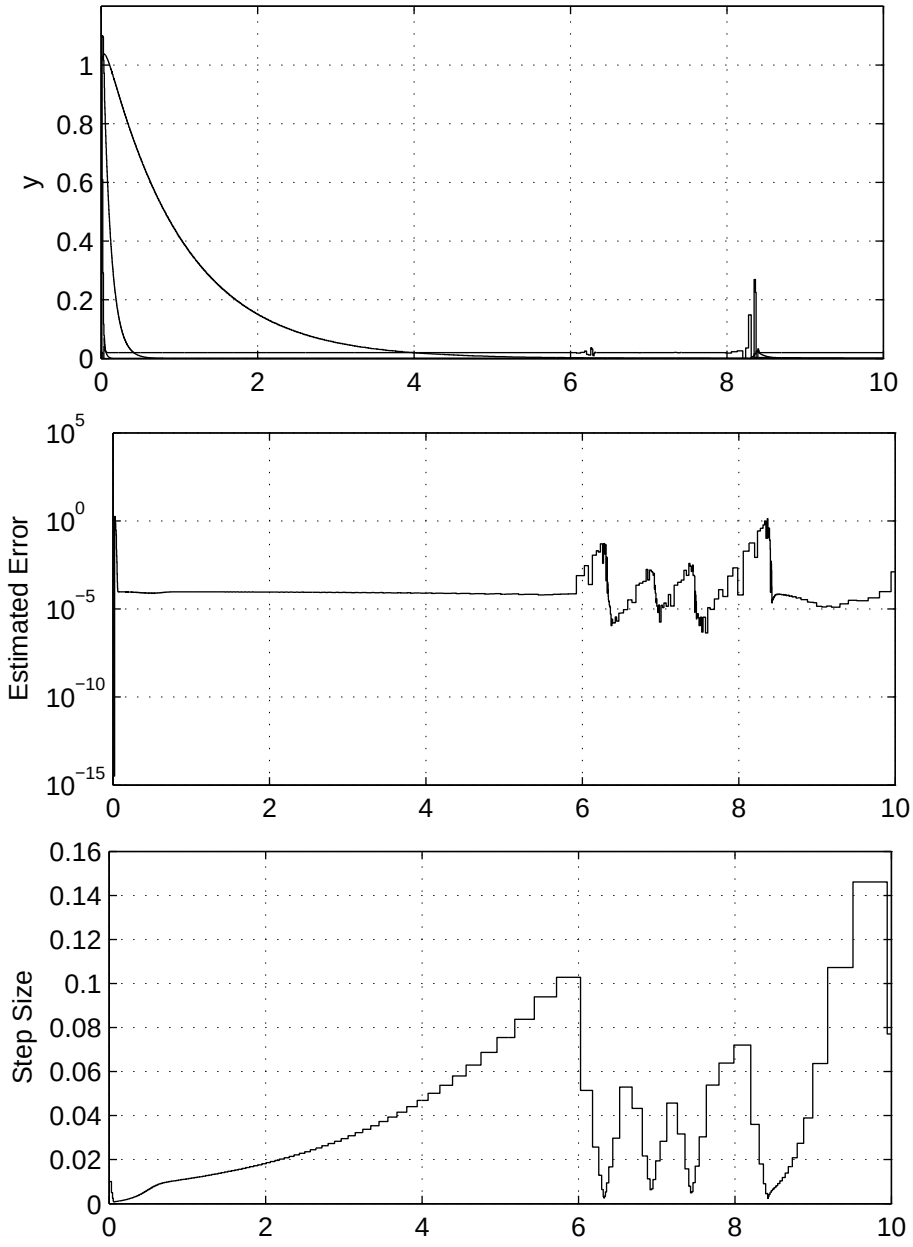

FIGURE 5.25. ODE set C inlined with HW-SDIRK

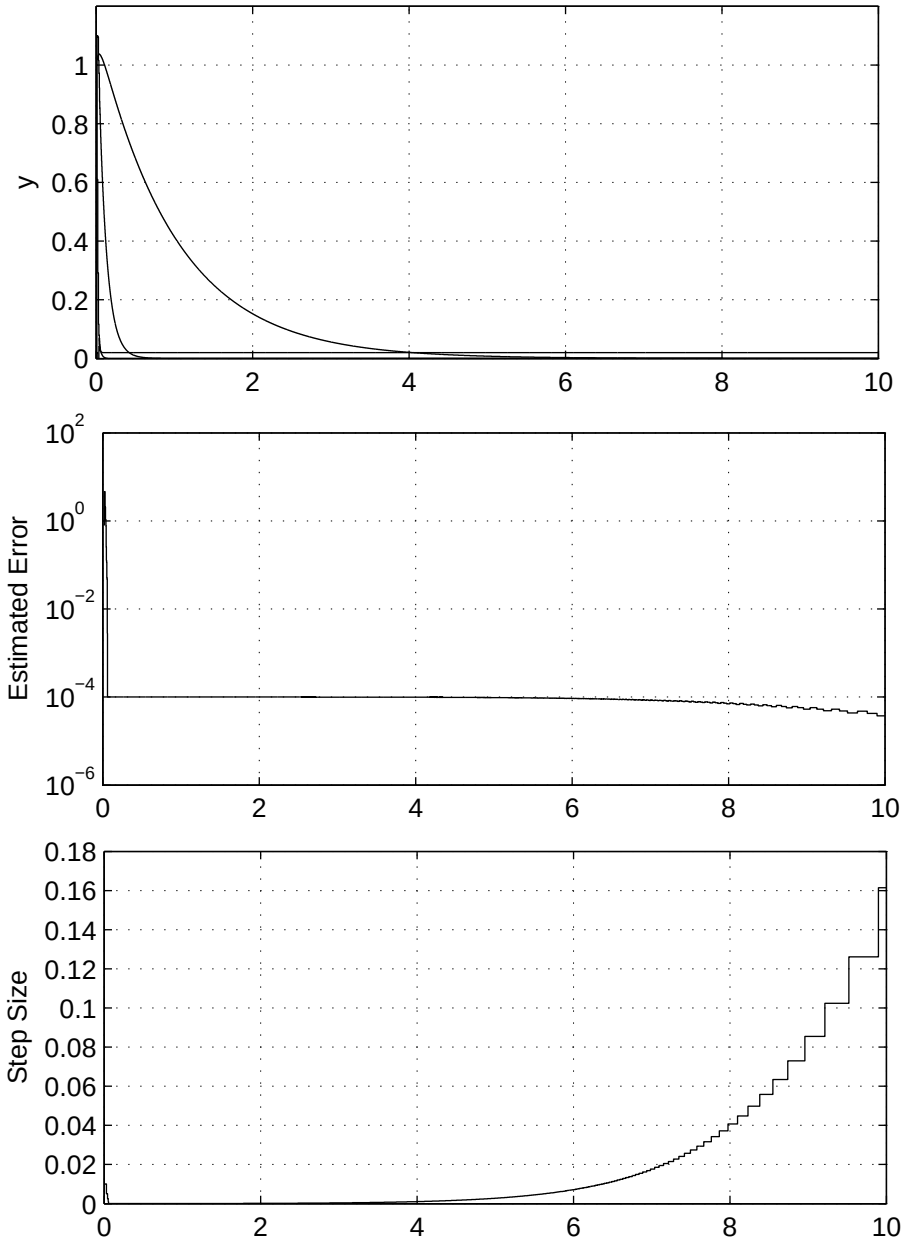

FIGURE 5.26. ODE set C inlined with HW-SDIRK and alternate error method

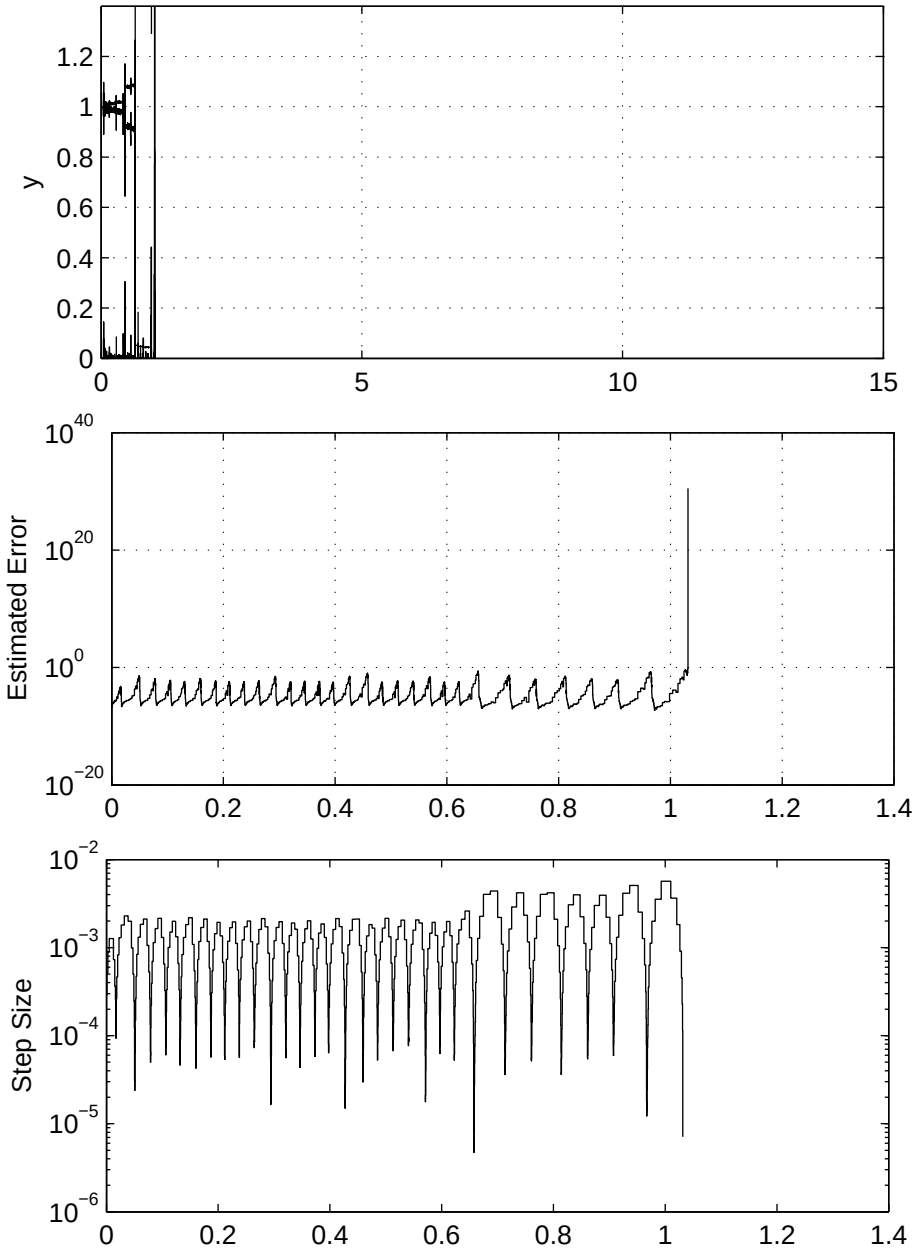

FIGURE 5.27. ODE set D inlined with Rad3

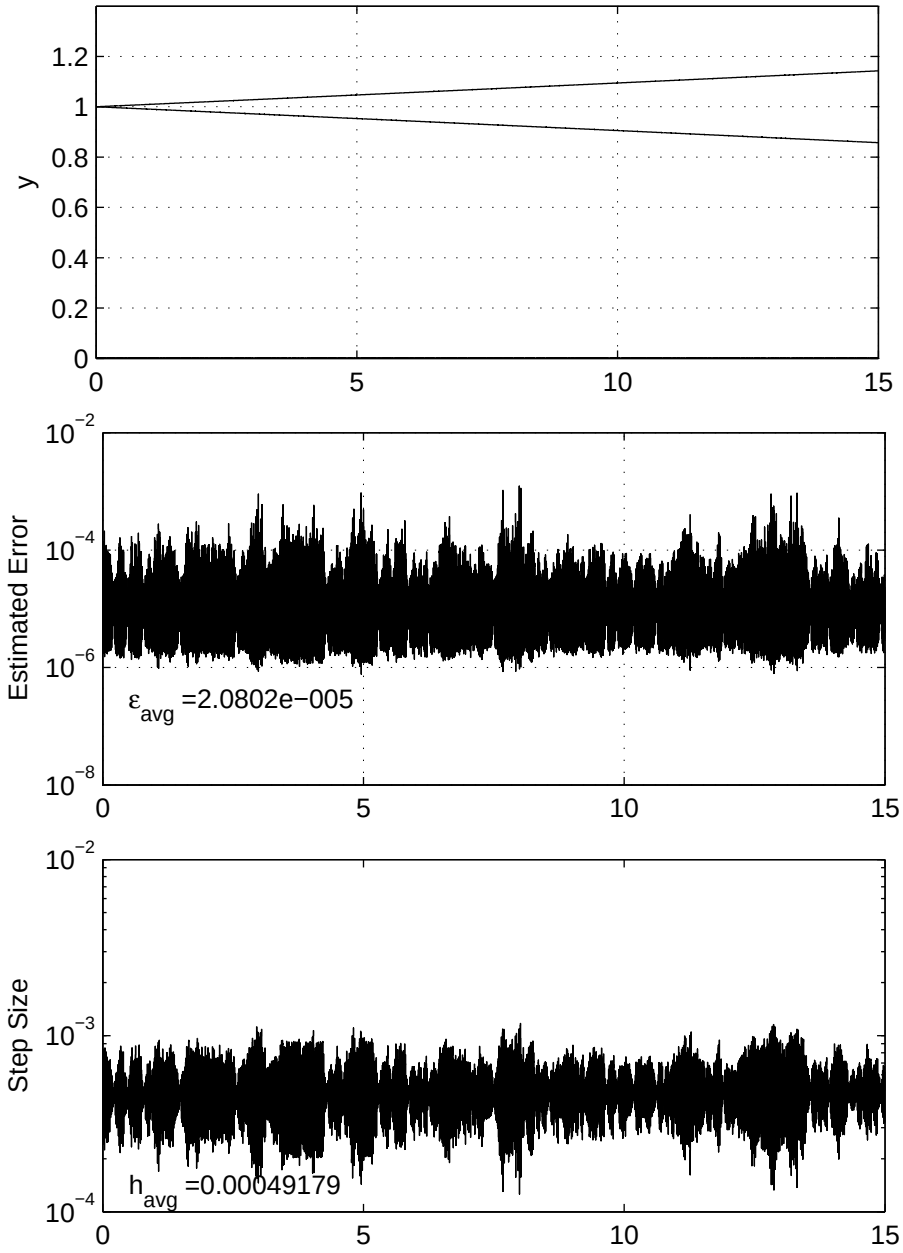

FIGURE 5.28. ODE set D inlined with Rad5

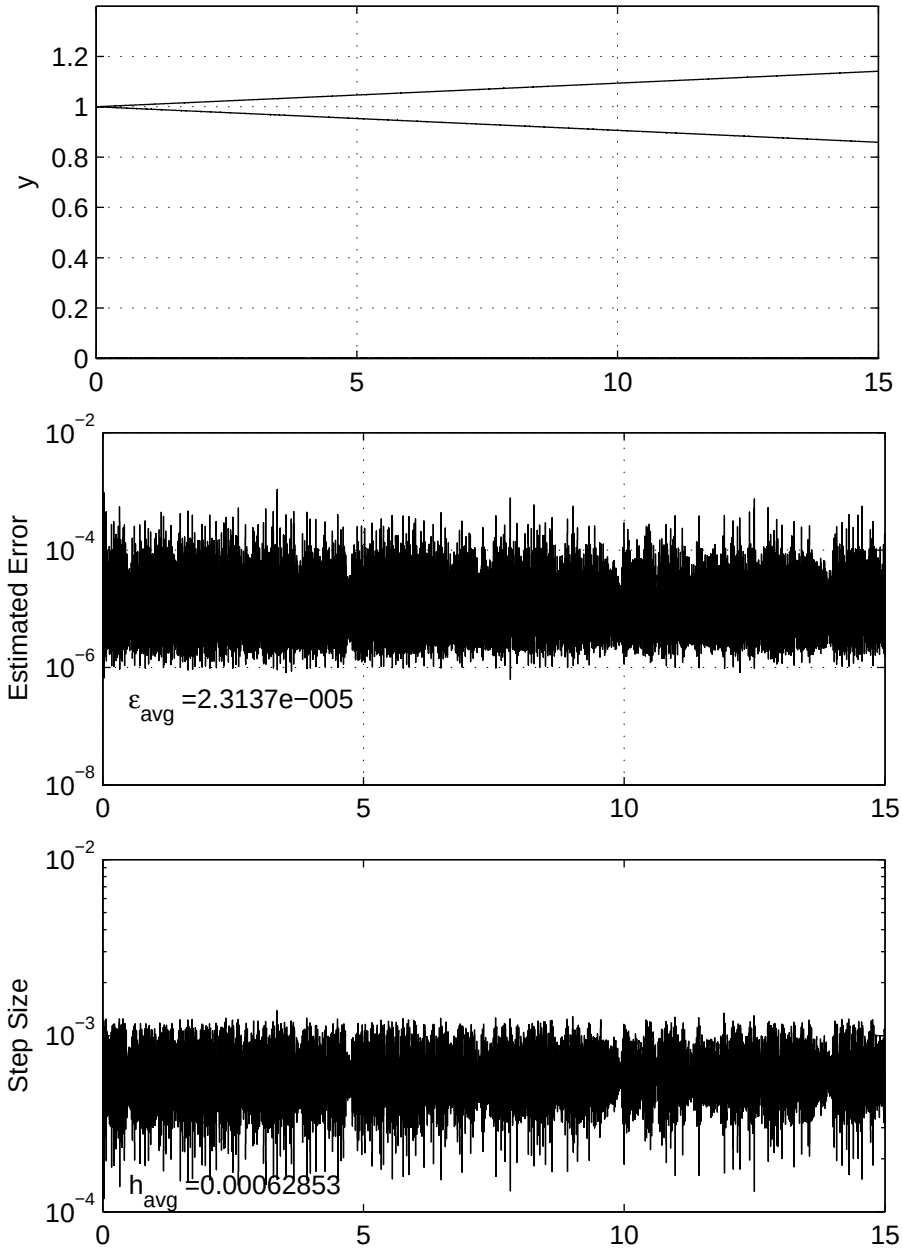

FIGURE 5.29. ODE set D inlined with Lob4

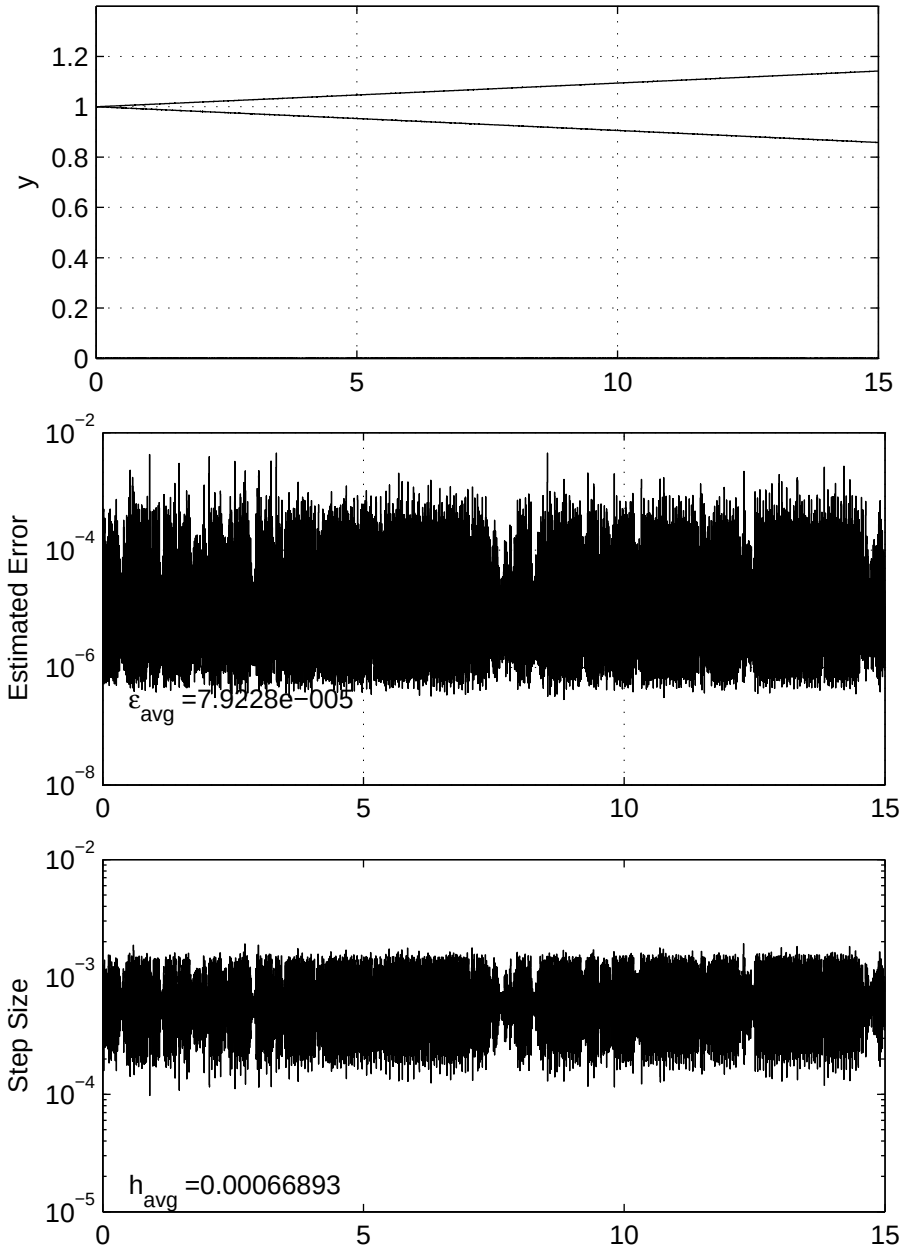

FIGURE 5.30. ODE set D inlined with Lob6

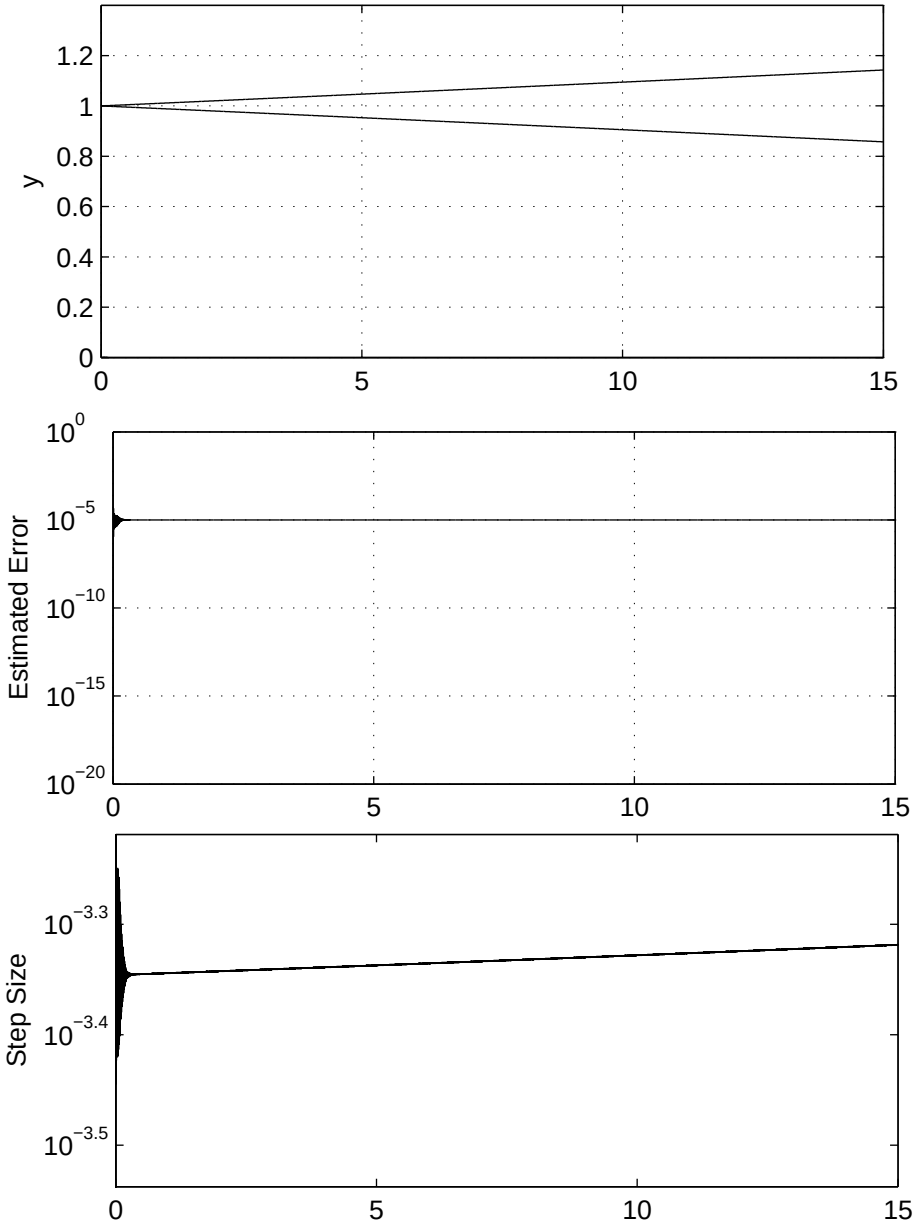

FIGURE 5.31. ODE set D inlined with HW-SDIRK

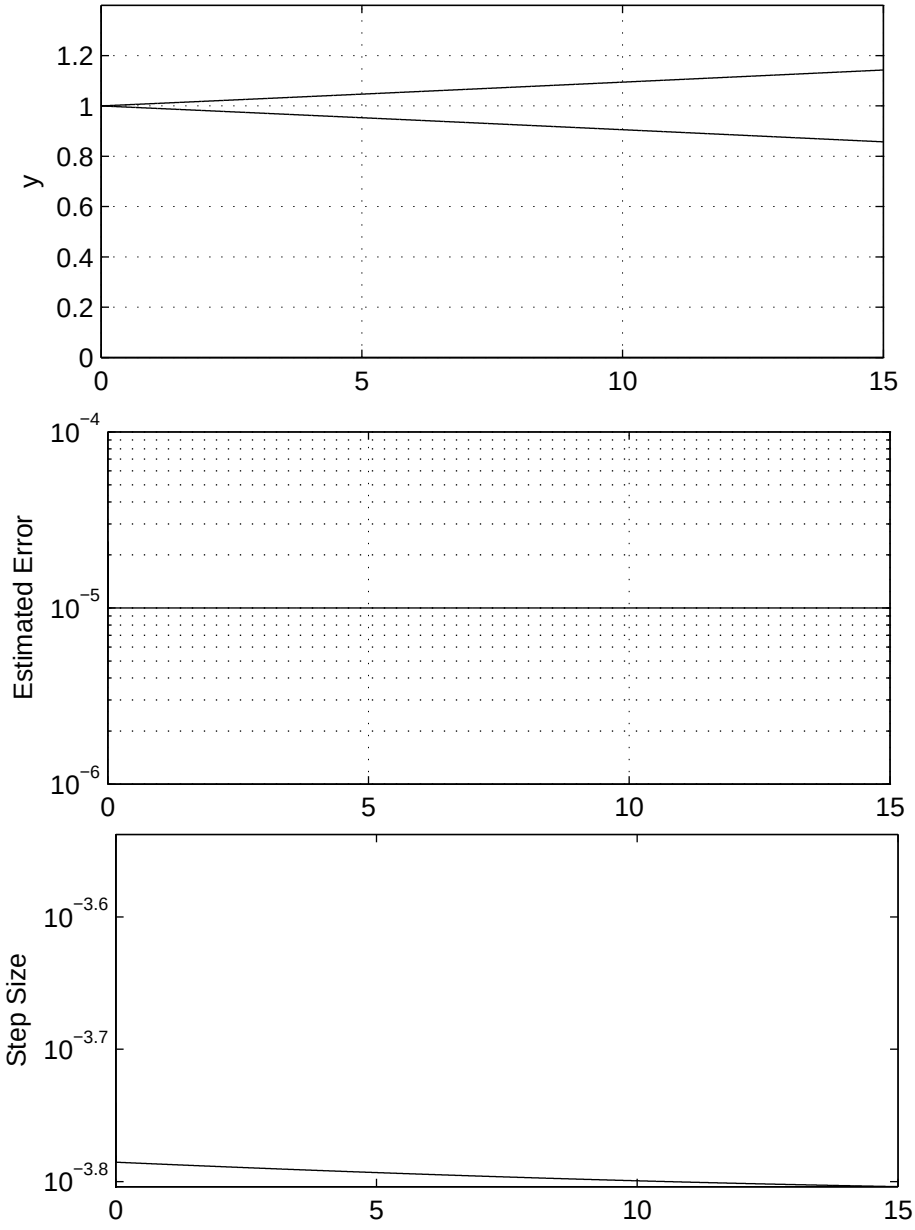

FIGURE 5.32. ODE set D inlined with HW-SDIRK and alternate error method

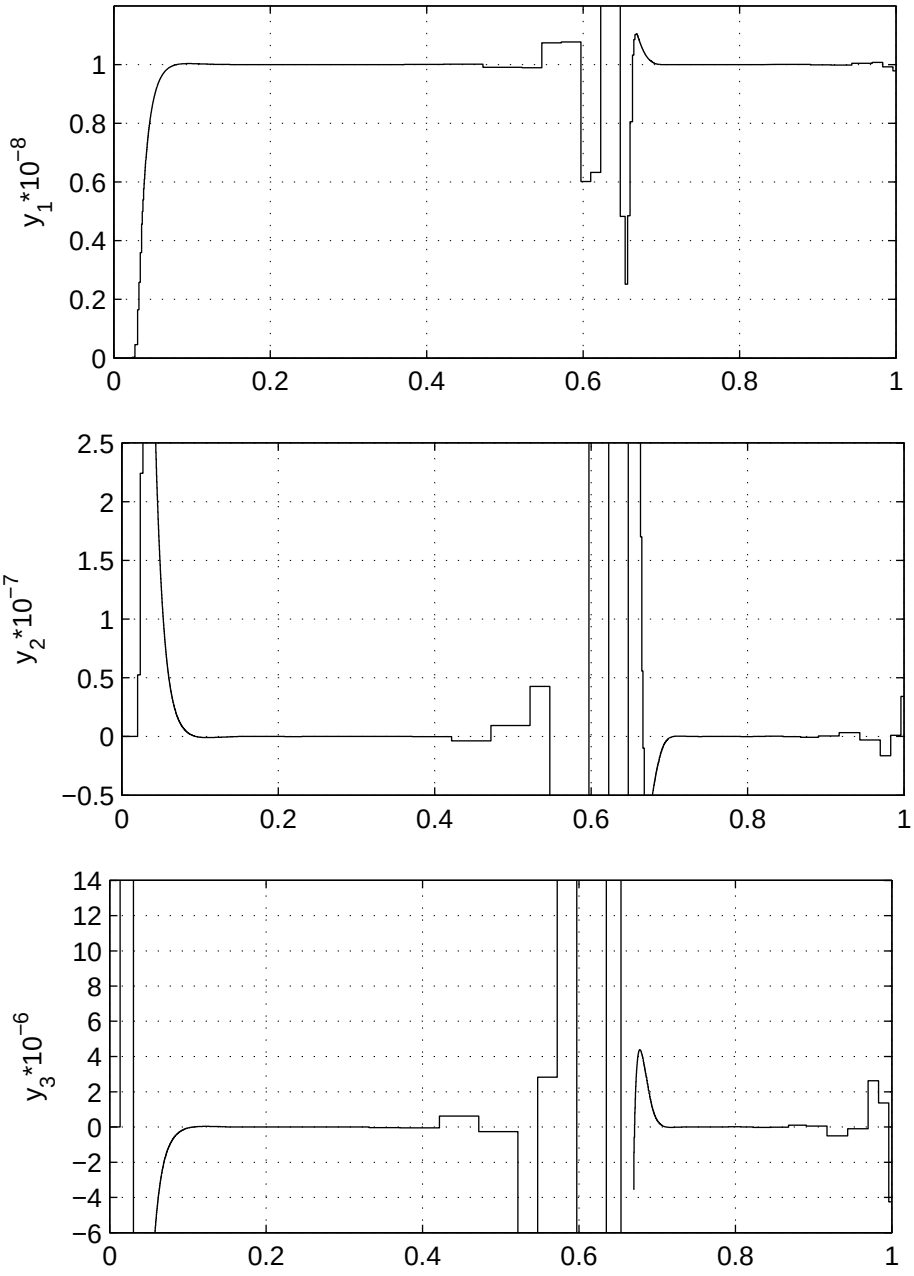

FIGURE 5.33. ODE set E inlined with Rad3

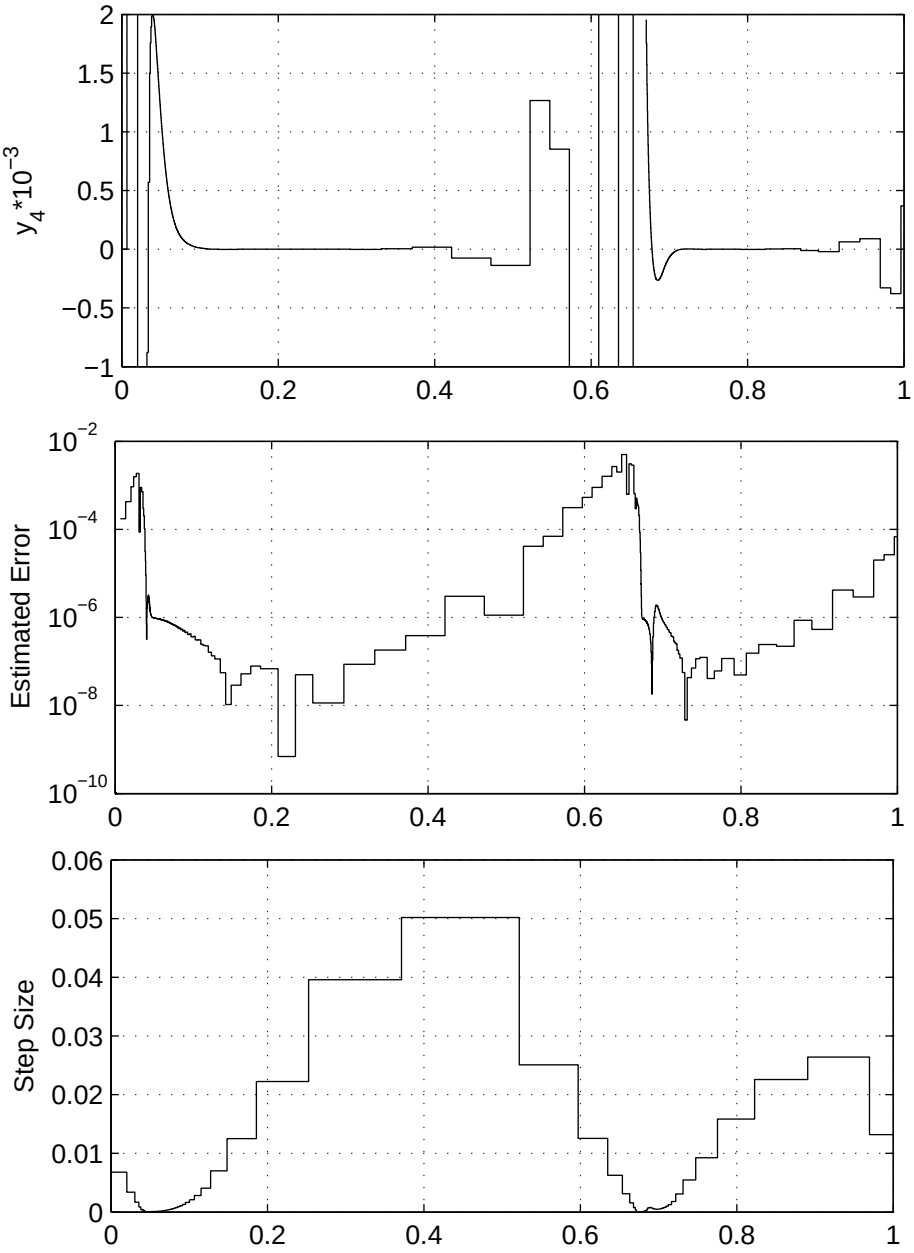

FIGURE 5.34. cont) ODE set E inlined with Rad3

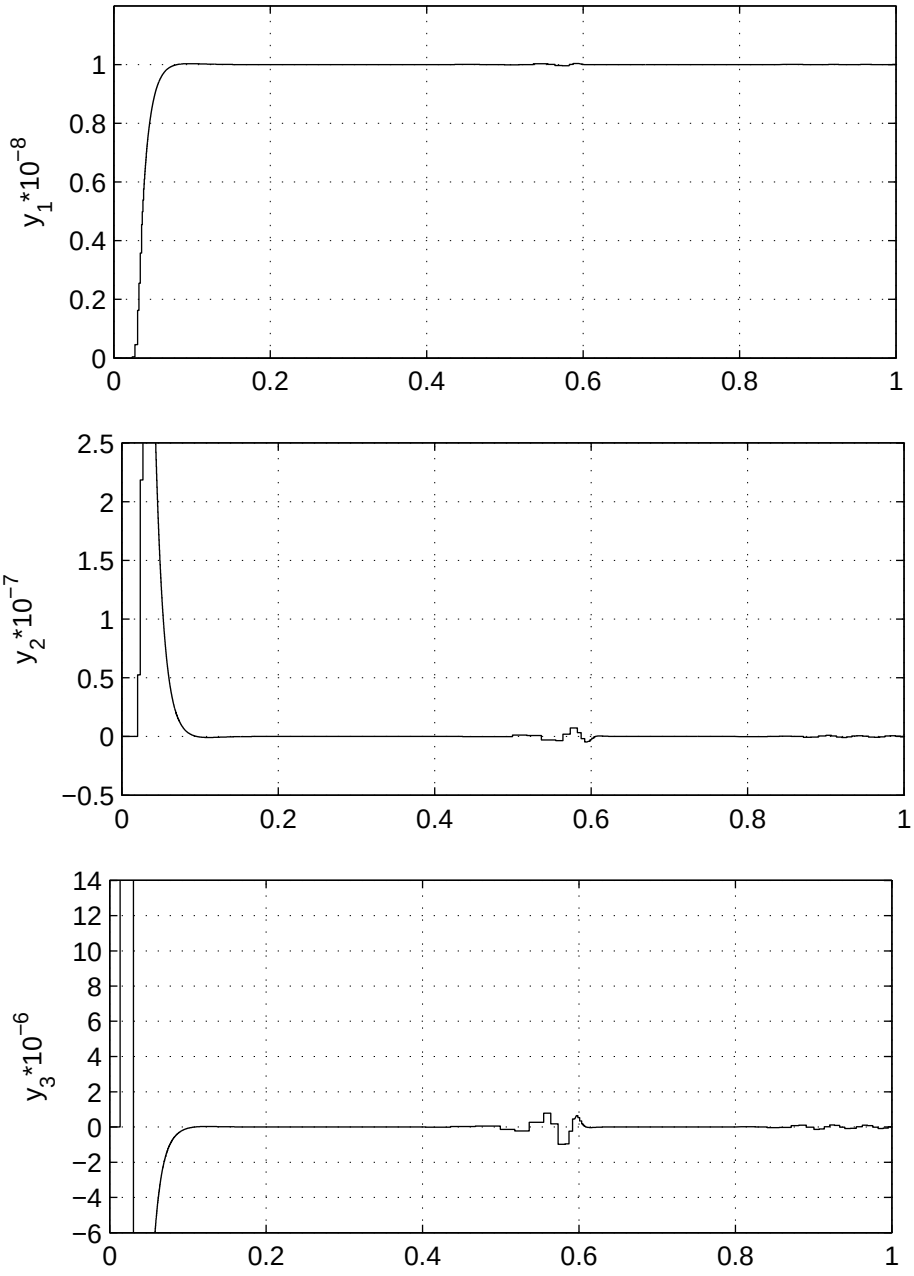

FIGURE 5.35. ODE set E inlined with Rad5

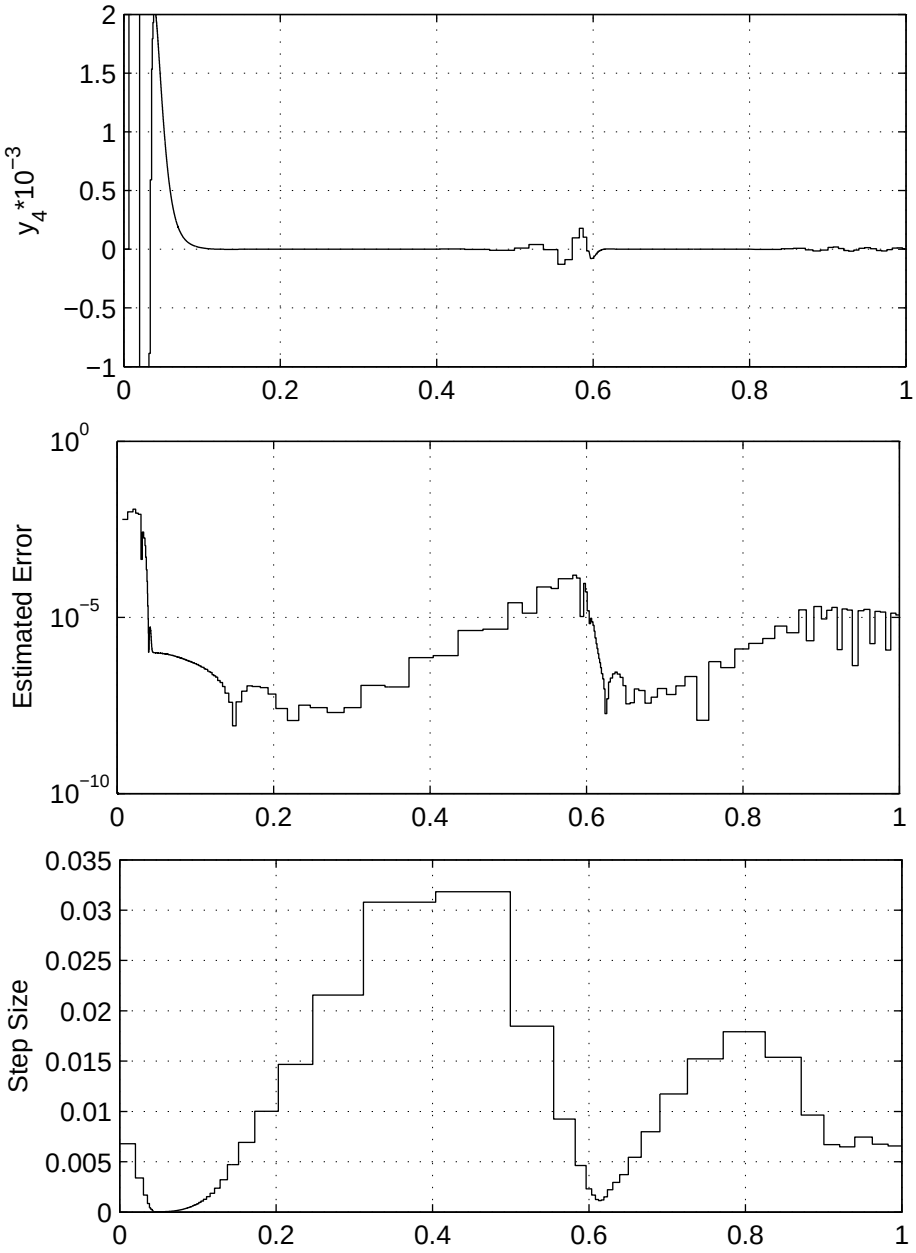

FIGURE 5.36. cont) ODE set E inlined with Rad5

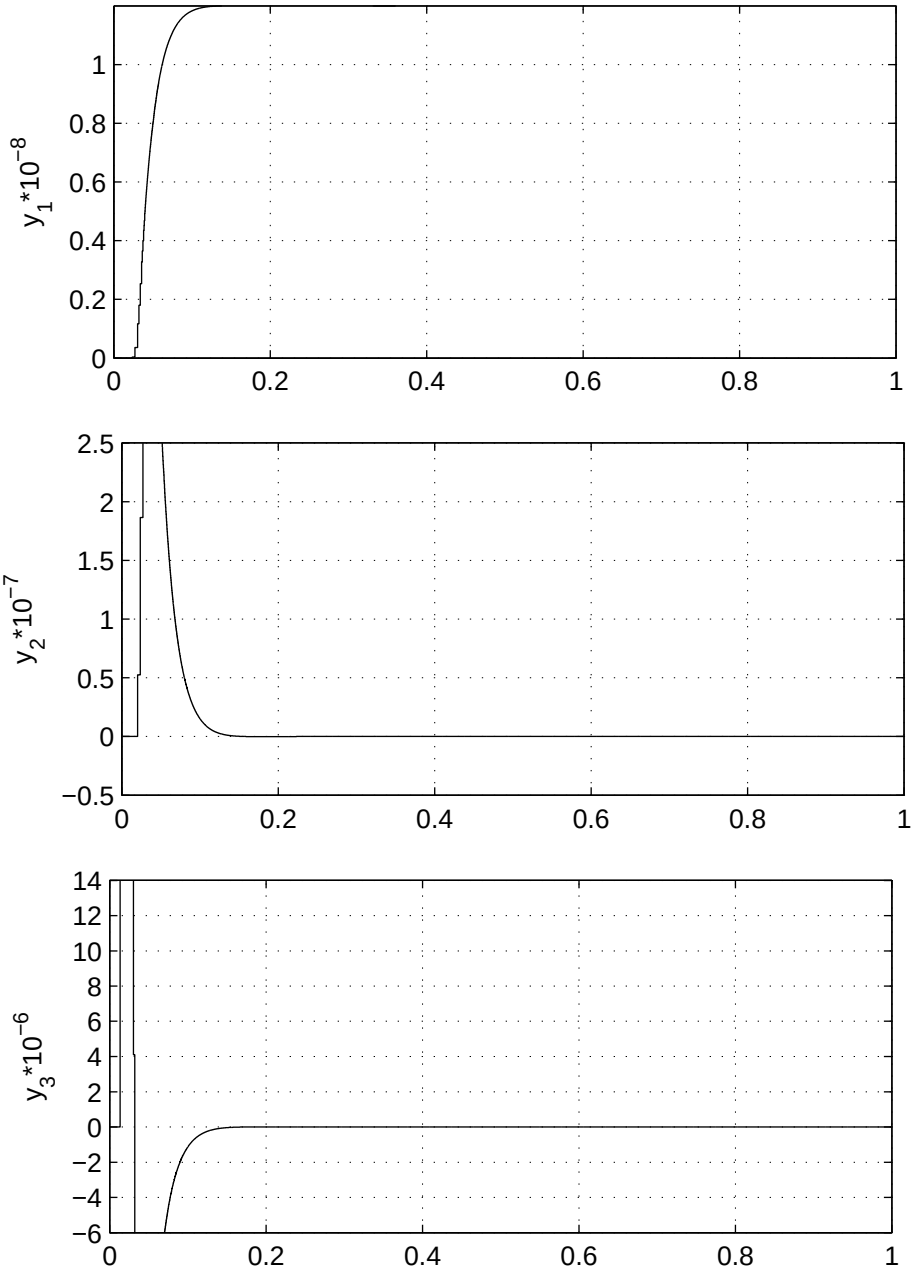

FIGURE 5.37. ODE set E inlined with Lob4

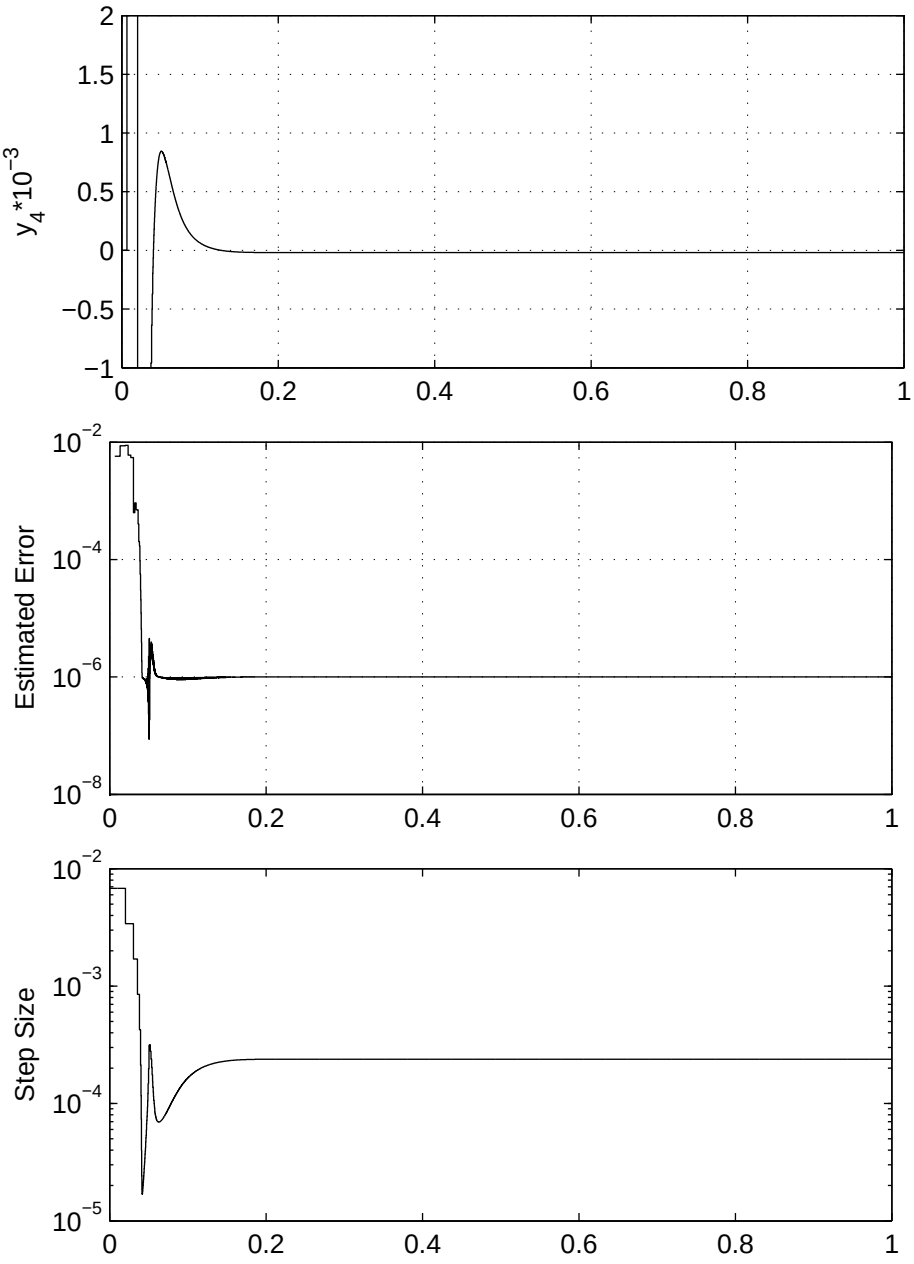

FIGURE 5.38. cont) ODE set E inlined with Lob4

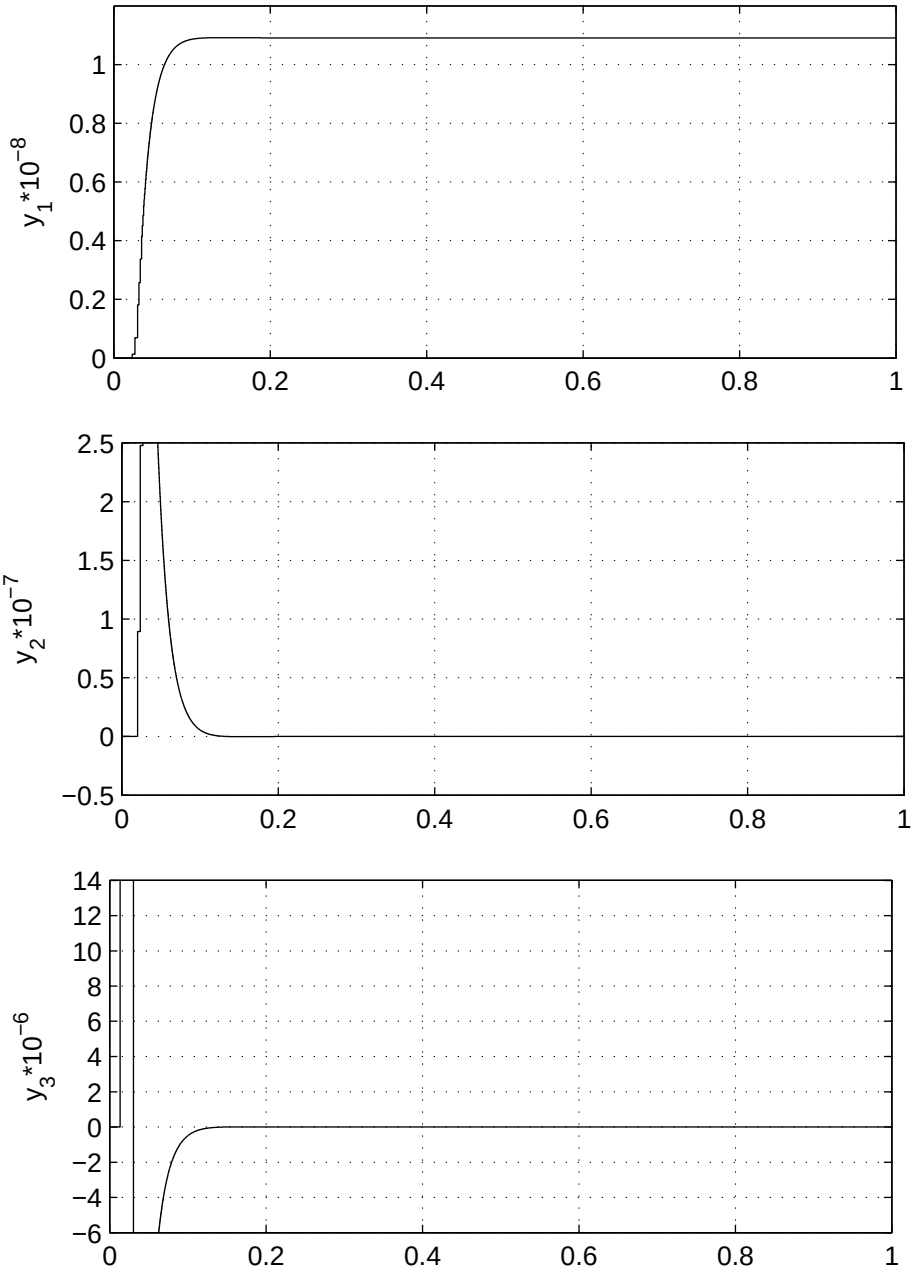

FIGURE 5.39. ODE set E inlined with Lob6

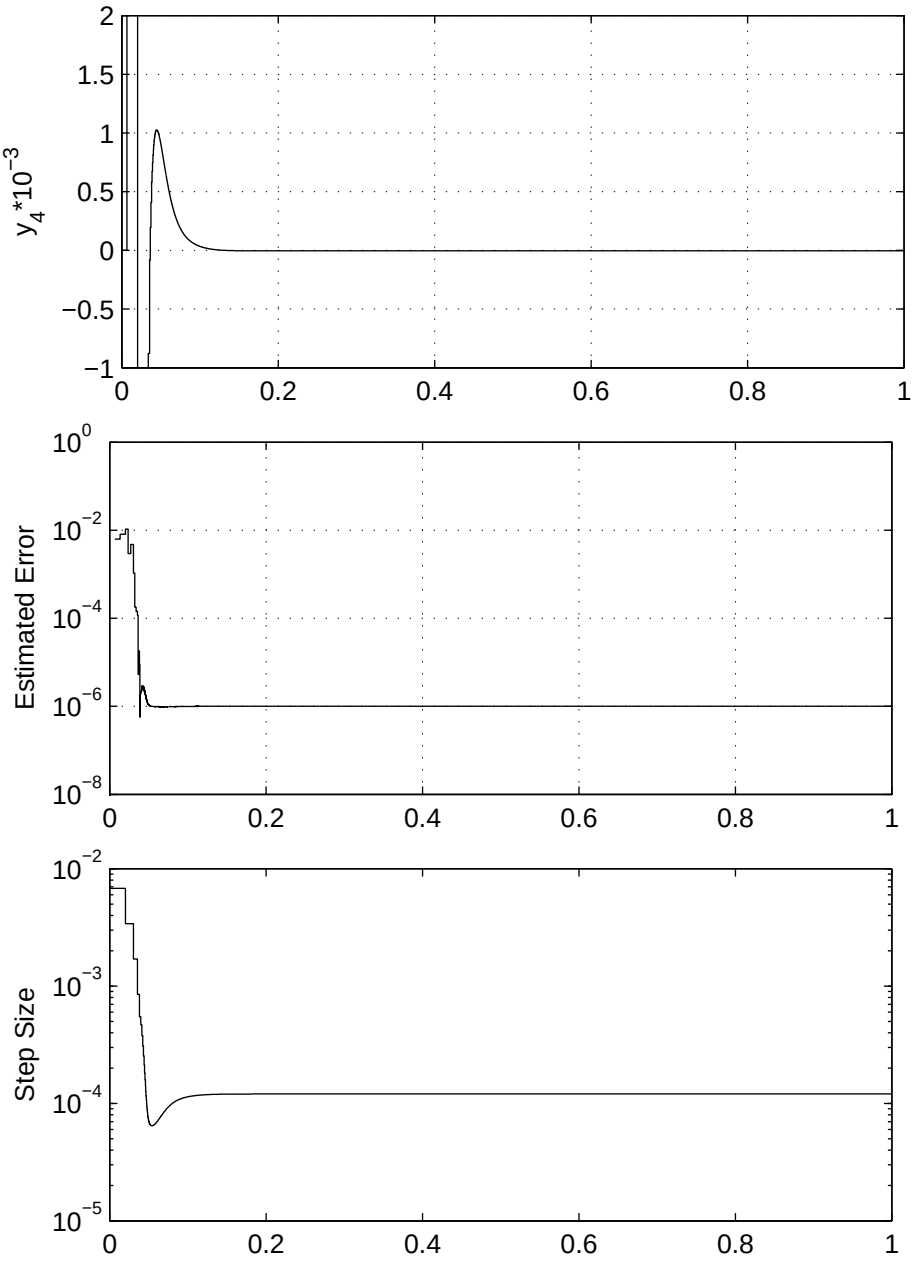

FIGURE 5.40. cont) ODE set E inlined with Lob6

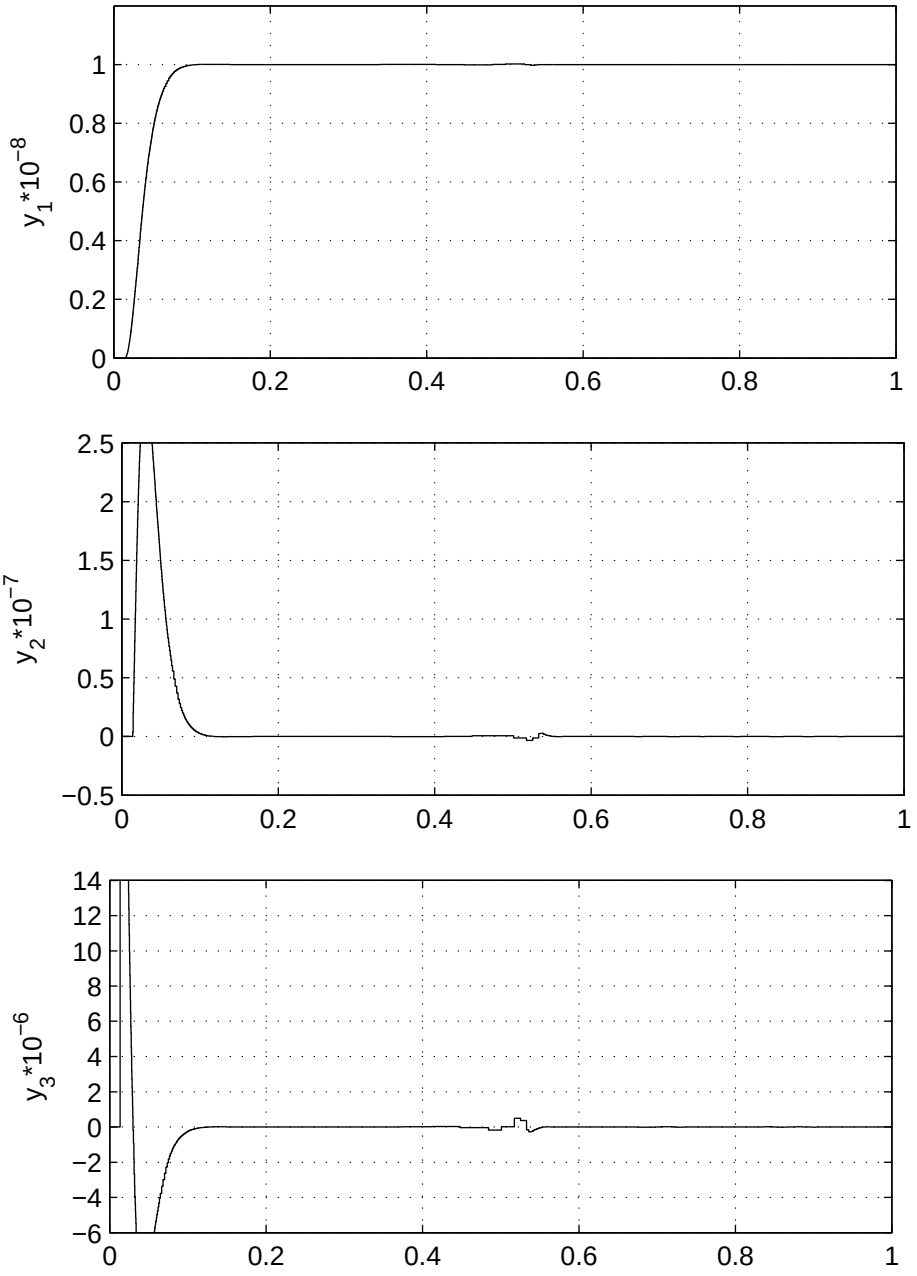

FIGURE 5.41. ODE set E inlined with HW-SDIRK

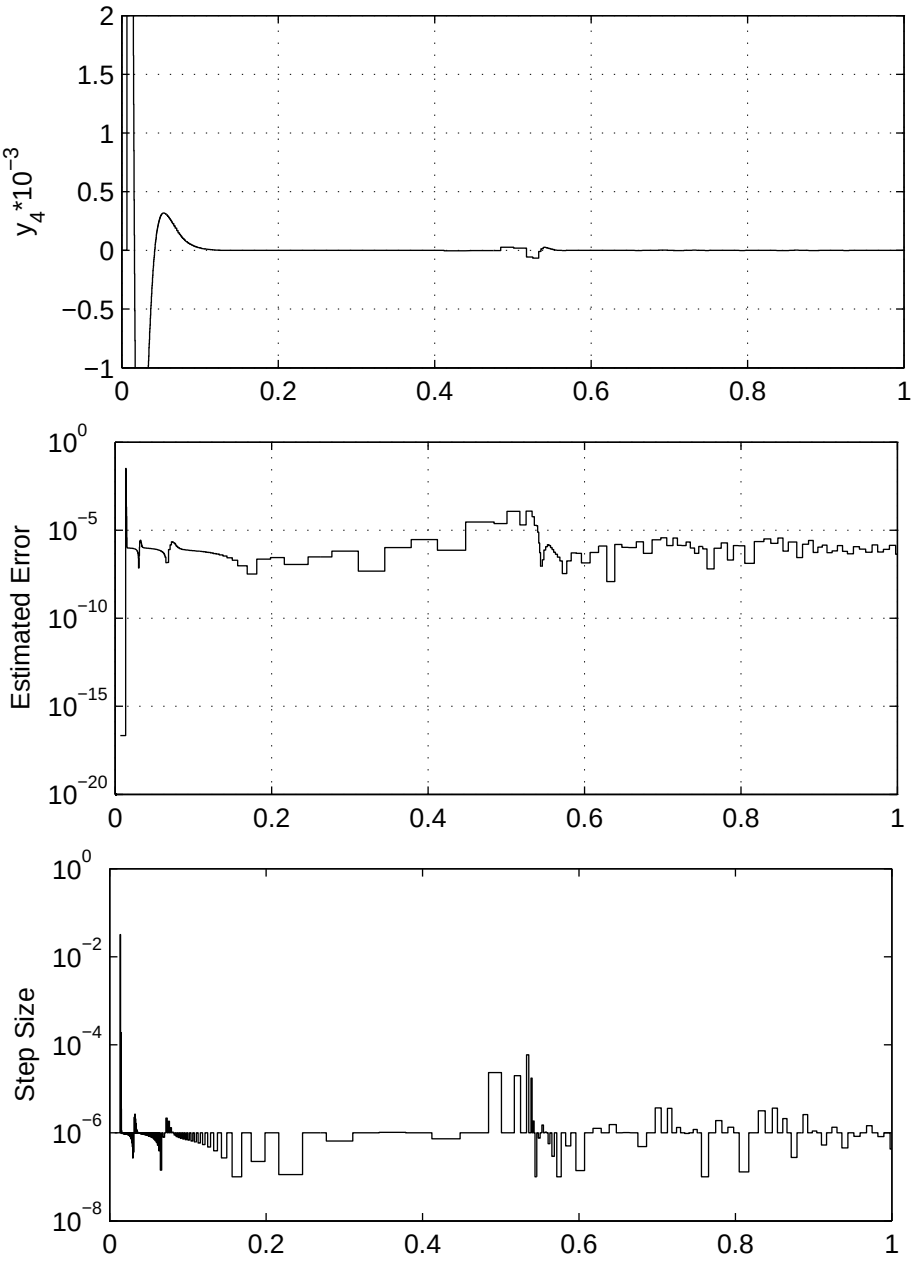

FIGURE 5.42. cont) ODE set E inlined with HW-SDIRK

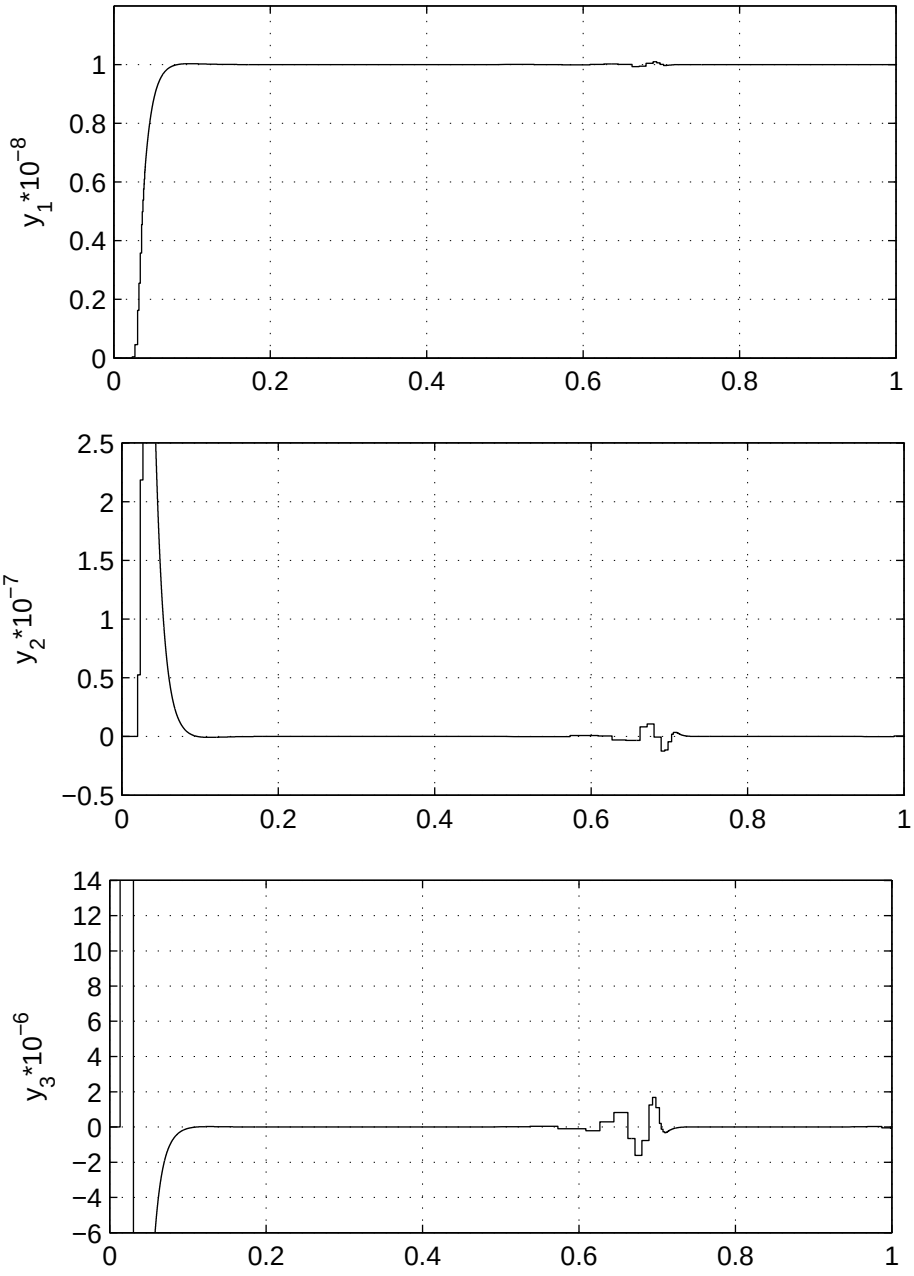

FIGURE 5.43. ODE set E inlined with HW-SDIRK and alternate error method

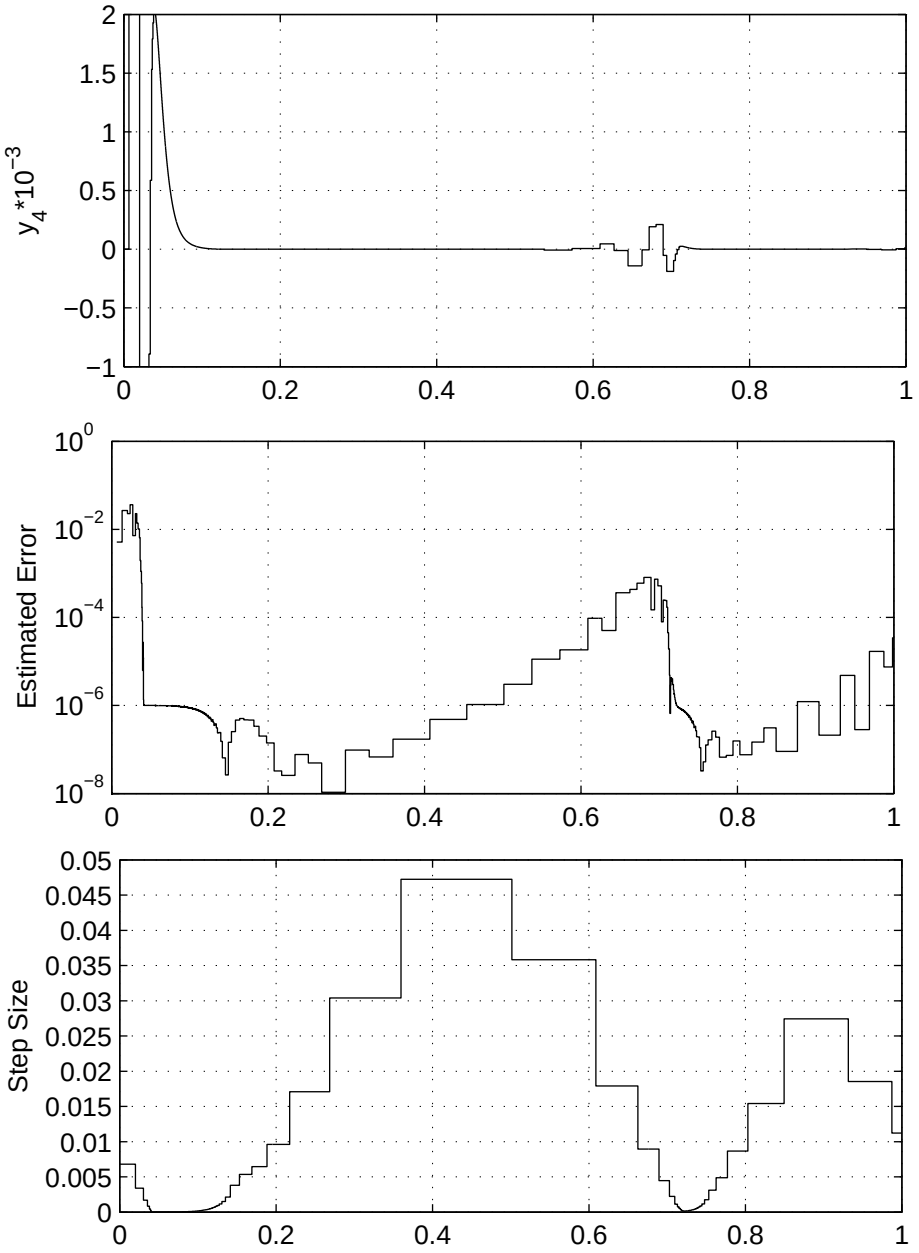

FIGURE 5.44. cont) ODE set E inlined with HW-SDIRK and alternate error method

Chapter 6

Conclusions

Two new error methods that can be used for step size control of HW-SDIRK and Lobatto IIIC(6) have been presented. Both methods have good damping and stability properties and are well suited for use when solving stiff systems. The Lobatto IIIC family seems to produce dubious results but both algorithms and associated error methods perform consistently with each other.

Together with inlining, implicit Runge–Kutta algorithms can be implemented with relative ease. Symbolic manipulations further add to the efficiency as tearing can be used to reduce the number of iteration variables. The problem of step size control with implicit Runge–Kutta algorithms is effectively solved by using data from a previous step. With embedding methods the computational load of the error estimate is now shared with the main integration algorithm. The biggest advantage with step size control and implicit Runge–Kutta algorithms is that with larger step sizes fewer integration steps are necessary. Thus, fewer iterations will be performed and the overall computational cost will be reduced.

The Radau IIA family performs well and has good resistance to bad initial (or propagated) data from taking too large of a step. The Lobatto IIIC family seems to produce correct results only for ODE set D, so evidently this family may not lead to a good general purpose stiff system solver.

A different step size control algorithm, such as a predictive control [23], may also be used to improve step size selection. Depending on the application this may further avoid the occurrence of rejected steps. The restriction to limit the step sizes from drastically changing is not needed since implicit Runge–Kutta algorithms are single–step algorithms. This was seen in the results for ODE set C and E as the

integration algorithms could not keep up with the dynamics of the systems being solved. By allowing the step size to drastically change, the Radau IIA algorithms may be suitable for use in real–time applications. In this application, much like the in the numerical experiments performed, there isn't time to reject and recompute a step.

Appendix A

Test Problems

In this section are the 5 test problems selected from [12] with considerations from [17]. The complete set of 25 test problems were separated into five different classes and have been widely used to test the performance of stiff system solvers. The initial step sizes for each of the problems are given by h_0.

A.1 Class A – Linear with real eigenvalues

(A2; circuit theory)

$$y_1' = -1800y_1 + 900y_2 \qquad y_1(0) = 0 \tag{A.1}$$

$$y_i' = y_{i-1} - 2y_i + y_{i+1} \qquad y_i(0) = 0 \qquad i = 2 \ldots 8 \tag{A.2}$$

$$y_9' = 1000y_8 - 2000y_9 + 1000 \qquad y_9(0) = 0 \tag{A.3}$$

$$h_0 = 5 \times 10^{-4}$$

A.2 Class B – Linear with non–real eigenvalues

(B1)

$$y_1' = -y_1 + y_2 \qquad y_1(0) = 1 \tag{A.4}$$

$$y_2' = -100y_1 - y_2 \qquad y_2(0) = 0 \tag{A.5}$$

$$y_3' = -100y_3 + y_4 \qquad y_3(0) = 1 \tag{A.6}$$

$$y_4' = -10000y_3 - 100y_4 \qquad y_4(0) = 0 \tag{A.7}$$

$$h_0 = 7 \times 10^{-3}$$

A.3 Class C – Non-linear coupling

(C1)

$$y_1' = -y_1 + y_2^2 + y_3^2 + y_4^2 \qquad y_1(0) = 1 \tag{A.8}$$

$$y_2' = -10y_2 + 10(y_3^2 + y_4^2) \qquad y_2(0) = 1 \tag{A.9}$$

$$y_3' = -40y_3 + 40y_4^2 \qquad y_3(0) = 1 \tag{A.10}$$

$$y_4' = -100y_4 + 2 \qquad y_4(0) = 1 \tag{A.11}$$

$$h_0 = 10^{-2}$$

A.4 Class D – Non-linear with real eigenvalues

(D4; chemistry)

$$y_1' = -0.013y_1 - 1000y_1y_3 \qquad y_1(0) = 1 \tag{A.12}$$

$$y_2' = -2500y_2y_3 \qquad y_2(0) = 1 \tag{A.13}$$

$$y_3' = +0.013y_1 - 1000y_1y_3 - 2500y_2y_3 \qquad y_3(0) = 0 \tag{A.14}$$

$$h_0 = 2.9 \times 10^{-4}$$

A.5 Class E – Non-linear with non-real eigenvalues

(E1; control theory)

$$y_1' = y_2 \qquad y_1(0) = 0 \tag{A.15}$$

$$y_2' = y_3 \qquad y_2(0) = 0 \tag{A.16}$$

$$y_3' = y_4 \qquad y_3(0) = 0 \tag{A.17}$$

$$y_4' = \left(y_1^2 - \sin(y_1) - K^4\right) y_1 + \left(\frac{y_2y_3}{y_1^2+1} - 4K^3\right) y_2 + (1 - 6K^2)y_3 + (10e^{-y_4^2} - 4K)y_4 + 1 \qquad y_4(0) = 0 \tag{A.18}$$

$$h_0 = 6.8 \times 10^{-3} \qquad K = 100$$

Appendix B

Sample Implementation

Shown in the following listing is a Modelica implementation of ODE set A inlined with Radau IIA(5) together with the embedding method. This is the same model that was used for simulation. Although Modelica supports arrays, an implementation using arrays doesn't behave as expected in version 5.0 of Dymola.

Listing B.1. ODE set A inlined with Rad5 in Modelica

```
model odeArad5
  parameter Real tol=1.0e-5 "Tolerated error";
  parameter Real h0=5.0e-4 "Initial step size";
  constant Real ti1 = (4 - sqrt(6))/10 "Time instant 1";
  constant Real ti2 = (4 + sqrt(6))/10 "Time instant 2";
  constant Real ti3 = 1 "Time instant 3";

  //Butcher tableau A-matrix coefficients
  constant Real a11 = (88 - 7*sqrt(6))/360;
  constant Real a12 = (296 - 169*sqrt(6))/1800;
  constant Real a13 = (-2 + 3*sqrt(6))/225;
  constant Real a21 = (296 + 169*sqrt(6))/1800;
  constant Real a22 = (88 + 7*sqrt(6))/360;
  constant Real a23 = (-2 - 3*sqrt(6))/225;
  constant Real a31 = (16 - sqrt(6))/36;
  constant Real a32 = (16 + sqrt(6))/36;
  constant Real a33 = 1/9;

  //Rad5 error method error coefficients
  constant Real c1 = -0.00517140382204;
  constant Real c2 = -0.00094714677404;
  constant Real c3 = -0.04060469717694;
  constant Real c4 = -0.01364429384901;
  constant Real c5 = +1.41786808325433;
  constant Real c6 = -0.17475783086782;
  constant Real c7 = +0.48299282769491;
  constant Real c8 = -0.19733415138754;
  constant Real c9 = +0.55942205973218;
```

```
constant Real c10 = +0.10695524944855;

//State variables with initial conditions
Real y1(start=0);
Real y2(start=0);
Real y3(start=0);
Real y4(start=0);
Real y5(start=0);
Real y6(start=0);
Real y7(start=0);
Real y8(start=0);
Real y9(start=0);
Real y12(start=0);
Real y22(start=0);
Real y32(start=0);
Real y42(start=0);
Real y52(start=0);
Real y62(start=0);
Real y72(start=0);
Real y82(start=0);
Real y92(start=0);
output Real y13(start=0);
output Real y23(start=0);
output Real y33(start=0);
output Real y43(start=0);
output Real y53(start=0);
output Real y63(start=0);
output Real y73(start=0);
output Real y83(start=0);
output Real y93(start=0);
Real y13e(start=0);
Real y23e(start=0);
Real y33e(start=0);
Real y43e(start=0);
Real y53e(start=0);
Real y63e(start=0);
Real y73e(start=0);
Real y83e(start=0);
Real y93e(start=0);

Real y1dot(start=0);
```

```
Real y2dot(start=0);
Real y3dot(start=0);
Real y4dot(start=0);
Real y5dot(start=0);
Real y6dot(start=0);
Real y7dot(start=0);
Real y8dot(start=0);
Real y9dot(start=0);
Real y12dot(start=0);
Real y22dot(start=0);
Real y32dot(start=0);
Real y42dot(start=0);
Real y52dot(start=0);
Real y62dot(start=0);
Real y72dot(start=0);
Real y82dot(start=0);
Real y92dot(start=0);
Real y13dot(start=0);
Real y23dot(start=0);
Real y33dot(start=0);
Real y43dot(start=0);
Real y53dot(start=0);
Real y63dot(start=0);
Real y73dot(start=0);
Real y83dot(start=0);
Real y93dot(start=0);

//State history with initial conditions
Real y1old(start=0);
Real y2old(start=0);
Real y3old(start=0);
Real y4old(start=0);
Real y5old(start=0);
Real y6old(start=0);
Real y7old(start=0);
Real y8old(start=0);
Real y9old(start=0);
Real y12old(start=0);
Real y22old(start=0);
Real y32old(start=0);
Real y42old(start=0);
```

```
Real y52old(start=0);
Real y62old(start=0);
Real y72old(start=0);
Real y82old(start=0);
Real y92old(start=0);
Real y13old(start=0);
Real y23old(start=0);
Real y33old(start=0);
Real y43old(start=0);
Real y53old(start=0);
Real y63old(start=0);
Real y73old(start=0);
Real y83old(start=0);
Real y93old(start=0);

Real y12old2(start=0);
Real y22old2(start=0);
Real y32old2(start=0);
Real y42old2(start=0);
Real y52old2(start=0);
Real y62old2(start=0);
Real y72old2(start=0);
Real y82old2(start=0);
Real y92old2(start=0);
Real y13old2(start=0);
Real y23old2(start=0);
Real y33old2(start=0);
Real y43old2(start=0);
Real y53old2(start=0);
Real y63old2(start=0);
Real y73old2(start=0);
Real y83old2(start=0);
Real y93old2(start=0);

Real y1dotold(start=0);
Real y2dotold(start=0);
Real y3dotold(start=0);
Real y4dotold(start=0);
Real y5dotold(start=0);
Real y6dotold(start=0);
Real y7dotold(start=0);
```

```
  Real y8dotold(start=0);
  Real y9dotold(start=0);
  Real y12dotold(start=0);
  Real y22dotold(start=0);
  Real y32dotold(start=0);
  Real y42dotold(start=0);
  Real y52dotold(start=0);
  Real y62dotold(start=0);
  Real y72dotold(start=0);
  Real y82dotold(start=0);
  Real y92dotold(start=0);

  Real y1dotold2(start=0);
  Real y2dotold2(start=0);
  Real y3dotold2(start=0);
  Real y4dotold2(start=0);
  Real y5dotold2(start=0);
  Real y6dotold2(start=0);
  Real y7dotold2(start=0);
  Real y8dotold2(start=0);
  Real y9dotold2(start=0);
  Real y12dotold2(start=0);
  Real y22dotold2(start=0);
  Real y32dotold2(start=0);
  Real y42dotold2(start=0);
  Real y52dotold2(start=0);
  Real y62dotold2(start=0);
  Real y72dotold2(start=0);
  Real y82dotold2(start=0);
  Real y92dotold2(start=0);

  output Real e(start=0);
  output Real er(start=0);
  output Real h(start=h0);
  Integer count(start=0);
  discrete Real NextSampling(start=ti1*h0);
  discrete Real NextSampling2(start=ti2*h0);
  discrete Real NextSampling3(start=ti3*h0);
algorithm
  //Compute first stage state derivatives
  when time >= pre(NextSampling) then
```

```
  y1dot := -1800*y1 + 900*y2;
  y2dot := y1 - 2*y2 + y3;
  y3dot := y2 - 2*y3 + y4;
  y4dot := y3 - 2*y4 + y5;
  y5dot := y4 - 2*y5 + y6;
  y6dot := y5 - 2*y6 + y7;
  y7dot := y6 - 2*y7 + y8;
  y8dot := y7 - 2*y8 + y9;
  y9dot := 1000*y8 - 2000*y9 + 1000;
end when;

//Compute second stage state derivatives
when time >= pre(NextSampling2) then
  y12dot := -1800*y12 + 900*y22;
  y22dot := y12 - 2*y22 + y32;
  y32dot := y22 - 2*y32 + y42;
  y42dot := y32 - 2*y42 + y52;
  y52dot := y42 - 2*y52 + y62;
  y62dot := y52 - 2*y62 + y72;
  y72dot := y62 - 2*y72 + y82;
  y82dot := y72 - 2*y82 + y92;
  y92dot := 1000*y82 - 2000*y92 + 1000;
end when;

//Compute last stage state derivatives and states for time+h
when time >= pre(NextSampling3) then
  y13dot := -1800*y13 + 900*y23;
  y23dot := y13 - 2*y23 + y33;
  y33dot := y23 - 2*y33 + y43;
  y43dot := y33 - 2*y43 + y53;
  y53dot := y43 - 2*y53 + y63;
  y63dot := y53 - 2*y63 + y73;
  y73dot := y63 - 2*y73 + y83;
  y83dot := y73 - 2*y83 + y93;
  y93dot := 1000*y83 - 2000*y93 + 1000;

  //States
  y1 := y13old + a11*h*y1dot + a12*h*y12dot + a13*h*y13dot;
  y2 := y23old + a11*h*y2dot + a12*h*y22dot + a13*h*y23dot;
  y3 := y33old + a11*h*y3dot + a12*h*y32dot + a13*h*y33dot;
  y4 := y43old + a11*h*y4dot + a12*h*y42dot + a13*h*y43dot;
```

```
y5 := y53old + a11*h*y5dot + a12*h*y52dot + a13*h*y53dot;
y6 := y63old + a11*h*y6dot + a12*h*y62dot + a13*h*y63dot;
y7 := y73old + a11*h*y7dot + a12*h*y72dot + a13*h*y73dot;
y8 := y83old + a11*h*y8dot + a12*h*y82dot + a13*h*y83dot;
y9 := y93old + a11*h*y9dot + a12*h*y92dot + a13*h*y93dot;

y12 := y13old + a21*h*y1dot + a22*h*y12dot + a23*h*y13dot;
y22 := y23old + a21*h*y2dot + a22*h*y22dot + a23*h*y23dot;
y32 := y33old + a21*h*y3dot + a22*h*y32dot + a23*h*y33dot;
y42 := y43old + a21*h*y4dot + a22*h*y42dot + a23*h*y43dot;
y52 := y53old + a21*h*y5dot + a22*h*y52dot + a23*h*y53dot;
y62 := y63old + a21*h*y6dot + a22*h*y62dot + a23*h*y63dot;
y72 := y73old + a21*h*y7dot + a22*h*y72dot + a23*h*y73dot;
y82 := y83old + a21*h*y8dot + a22*h*y82dot + a23*h*y83dot;
y92 := y93old + a21*h*y9dot + a22*h*y92dot + a23*h*y93dot;

y13 := y13old + a31*h*y1dot + a32*h*y12dot + a33*h*y13dot;
y23 := y23old + a31*h*y2dot + a32*h*y22dot + a33*h*y23dot;
y33 := y33old + a31*h*y3dot + a32*h*y32dot + a33*h*y33dot;
y43 := y43old + a31*h*y4dot + a32*h*y42dot + a33*h*y43dot;
y53 := y53old + a31*h*y5dot + a32*h*y52dot + a33*h*y53dot;
y63 := y63old + a31*h*y6dot + a32*h*y62dot + a33*h*y63dot;
y73 := y73old + a31*h*y7dot + a32*h*y72dot + a33*h*y73dot;
y83 := y83old + a31*h*y8dot + a32*h*y82dot + a33*h*y83dot;
y93 := y93old + a31*h*y9dot + a32*h*y92dot + a33*h*y93dot;

//Compute error estimate
y13e := c1*y13old2 + c2*h*y1dotold2 + c3*y12old2 +
        c4*h*y12dotold2 + c5*y13old + c6*y1old +
        c7*h*y1dotold + c8*y12old + c9*h*y12dotold +
        c10*h*y13dot;
y23e := c1*y23old2 + c2*h*y2dotold2 + c3*y22old2 +
        c4*h*y22dotold2 + c5*y23old + c6*y2old +
        c7*h*y2dotold + c8*y22old + c9*h*y22dotold +
        c10*h*y23dot;
y33e := c1*y33old2 + c2*h*y3dotold2 + c3*y32old2 +
        c4*h*y32dotold2 + c5*y33old + c6*y3old +
        c7*h*y3dotold + c8*y32old + c9*h*y32dotold +
        c10*h*y33dot;
y43e := c1*y43old2 + c2*h*y4dotold2 + c3*y42old2 +
        c4*h*y42dotold2 + c5*y43old + c6*y4old +
```

```
        c7*h*y4dotold + c8*y42old + c9*h*y42dotold +
        c10*h*y43dot;
y53e := c1*y53old2 + c2*h*y5dotold2 + c3*y52old2 +
        c4*h*y52dotold2 + c5*y53old + c6*y5old +
        c7*h*y5dotold + c8*y52old + c9*h*y52dotold +
        c10*h*y53dot;
y63e := c1*y63old2 + c2*h*y6dotold2 + c3*y62old2 +
        c4*h*y62dotold2 + c5*y63old + c6*y6old +
        c7*h*y6dotold + c8*y62old + c9*h*y62dotold +
        c10*h*y63dot;
y73e := c1*y73old2 + c2*h*y7dotold2 + c3*y72old2 +
        c4*h*y72dotold2 + c5*y73old + c6*y7old +
        c7*h*y7dotold + c8*y72old + c9*h*y72dotold +
        c10*h*y73dot;
y83e := c1*y83old2 + c2*h*y8dotold2 + c3*y82old2 +
        c4*h*y82dotold2 + c5*y83old + c6*y8old +
        c7*h*y8dotold + c8*y82old + c9*h*y82dotold +
        c10*h*y83dot;
y93e := c1*y93old2 + c2*h*y9dotold2 + c3*y92old2 +
        c4*h*y92dotold2 + c5*y93old + c6*y9old +
        c7*h*y9dotold + c8*y92old + c9*h*y92dotold +
        c10*h*y93dot;

e := sqrt((y13e - y13)^2 + (y23e - y23)^2 + (y33e - y33)^2 +
     (y43e - y43)^2 + (y53e - y53)^2 + (y63e - y63)^2 +
     (y73e - y73)^2 + (y83e - y83)^2 + ( y93e - y93)^2);
er := if e > tol/10 then e else tol/10;

//Keep step size constant for 2 steps
if count == 2 then
  h := max(h/2, min(2*h, (tol/er)^(1/6)*h));
  count := 1;
else
  count := count + 1;
end if;

//Next integration time instants
NextSampling := time + h*ti1;
NextSampling2 := time + h*ti2;
NextSampling3 := time + h*ti3;
```

```
//Remember state and state derivative history
y12old2 := y12old;
y22old2 := y22old;
y32old2 := y32old;
y42old2 := y42old;
y52old2 := y52old;
y62old2 := y62old;
y72old2 := y72old;
y82old2 := y82old;
y92old2 := y92old;

y13old2 := y13old;
y23old2 := y23old;
y33old2 := y33old;
y43old2 := y43old;
y53old2 := y53old;
y63old2 := y63old;
y73old2 := y73old;
y83old2 := y83old;
y93old2 := y93old;

y1dotold2 := y1dotold;
y2dotold2 := y2dotold;
y3dotold2 := y3dotold;
y4dotold2 := y4dotold;
y5dotold2 := y5dotold;
y6dotold2 := y6dotold;
y7dotold2 := y7dotold;
y8dotold2 := y8dotold;
y9dotold2 := y9dotold;

y12dotold2 := y12dotold;
y22dotold2 := y22dotold;
y32dotold2 := y32dotold;
y42dotold2 := y42dotold;
y52dotold2 := y52dotold;
y62dotold2 := y62dotold;
y72dotold2 := y72dotold;
y82dotold2 := y82dotold;
y92dotold2 := y92dotold;
```

```
y1old := y1;
y2old := y2;
y3old := y3;
y4old := y4;
y5old := y5;
y6old := y6;
y7old := y7;
y8old := y8;
y9old := y9;

y12old := y12;
y22old := y22;
y32old := y32;
y42old := y42;
y52old := y52;
y62old := y62;
y72old := y72;
y82old := y82;
y92old := y92;

y13old := y13;
y23old := y23;
y33old := y33;
y43old := y43;
y53old := y53;
y63old := y63;
y73old := y73;
y83old := y83;
y93old := y93;

y1dotold := y1dot;
y2dotold := y2dot;
y3dotold := y3dot;
y4dotold := y4dot;
y5dotold := y5dot;
y6dotold := y6dot;
y7dotold := y7dot;
y8dotold := y8dot;
y9dotold := y9dot;

y12dotold := y12dot;
```

```
    y22dotold := y22dot;
    y32dotold := y32dot;
    y42dotold := y42dot;
    y52dotold := y52dot;
    y62dotold := y62dot;
    y72dotold := y72dot;
    y82dotold := y82dot;
    y92dotold := y92dot;

  end when;

end odeArad5;
```

References

[1] M. Minsky, "Models, Minds, Machines," *Proceedings IFIP Congress,* pp. 45–49, 1965.

[2] F. E. Cellier and E. Kofman, *Continuous System Simulation,* Springer, New York, 2005. In preparation.

[3] G. D. Byrne and A. C. Hindmarsh, "Stiff ODE Solvers: A Review of Current and Coming Attractions," *J. Comput. Phys. 70,* pp. 1-62, 1987.

[4] L. F. Shampine, "Measuring Stiffness," *Appl. Numer. Math. 1*, pp. 107–119, 1985.

[5] F. E. Cellier,*Continuous System Modeling,* Springer, New York. 1991.

[6] J. C. Butcher, *The Numerical Analysis of Ordinary Differential Equations: RungeKutta and General Linear Methods.* John Wiley, Chichester, United Kingdom, 1987.

[7] H. Elmqvist, M. Otter, and F. E. Cellier, "Inline Integration: A New Mixed Symbolic/Numeric Approach for solving Differential–Algebraic Equation Systems," *Proc. SCS European Simulation Multiconference,* pp. xxiii–xxxiv, Prague, Czech Republic, 1995.

[8] H. Elmqvist and M. Otter, "Methods for Tearing Systems of Equations in Object–Oriented Modeling," *Proc. SCS European Simulation Multiconference*, pp. 326–332, Barcelona, Spain, 1994

[9] C. W. Gear, "Simultaneous Numerical Solution of Differential–Algerbraic Equations," *IEEE Trans. Circuit Theory,* CT–18(1):89–95, 1971.

[10] E. Hairer, S. P. Nørsett, and G. Wanner, *Solving Ordinary Differential Equations I: Nonstiff Problems,* Springer, Berlin, 2nd rev. ed., 2000.

[11] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential–Algebraic Problems,* Springer, Berlin, 2nd rev. ed., 2002.

[12] W. H. Enright, T. E. Hull, and B. Lindberg, "Comparing Numerical Methods for Stiff Systems of ODEs," *BIT 15,* pp. 10-48, 1975.

[13] W. H. Enright, and T. E. Hull, "Comparing Numerical Methods for the Solution of Stiff Systems of ODEs Arising in Chemistry." *In: Numerical Methods for*

Differential Systems, Recent Developments in Algorithms, Software and Applications, Ed. L. Lapidus and W. E. Schiesser, New York: Academic Press, pp. 45-66, 1976.

[14] T. E. Hull, W. H. Enright, B. M. Fellen, and A. E. Sedgwick, "Comparing Numerical Methods for Ordinary Differential Equations," *SIAM J. Numer. Anal. 9*, pp. 603-637, 1972.

[15] T. E. Hull, W. H. Enright, B. M. Fellen, and A. E. Sedgwick, "Erratum to 'Comparing Numerical Methods for Ordinary Differential Equations'," *SIAM J. Numer. Anal. 11*, pp. 681, 1974.

[16] L. F. Shampine, "Ill–Conditioned Matrices and the Integration of Stiff ODEs," *J. Comput. Appl. Math. 48*, pp. 279-292, 1993.

[17] L. F. Shampine, "Evaluation of a Test Set for Stiff ODE Solvers," *ACM T. Math. Software 7,* pp. 409-420, 1981.

[18] L. F. Shampine and M. W. Reichelt, "The MATLAB ODE Suite," *SIAM Journal on Scientific Computing 18,* pp 1-22, 1997.

[19] L. Lapidus and J. H. Seinfeld, *Numerical Solution of Ordinary Differential Equations,* Academic Press, New York, 1971.

[20] W. L. Miranker, *Numerical Methods for Stiff Equations and Singular Perturbation Problems,* D. Reidel Publishing, Holland, 1981.

[21] Modelica Association,*Modelica – A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification*, Modelica Association, 2002.

[22] Dynasim AB,*Dymola – Dynamic Modeling Laboratory, Users Manual*, Dynasim AB, 2002.

[23] K. Gustafsson, "Control-Theoretic Techniques for Selection in Implicit Runge-Kutta," *ACM T. Math. Software 20,* pp. 496-517, 1994.