8

Differential Algebraic Equation Solvers

Preview

In the previous chapter, we have discussed symbolic algorithms for converting implicit and even higher–index DAE systems to explicit ODE form. In this chapter, we shall look at these very same problems once more from a different angle. Rather than converting implicit DAEs to explicit ODE form, we shall try to solve the DAE systems directly. Solvers that are capable of dealing with implicit DAE descriptions directly have been coined *differential algebraic equation solvers* or DAE solvers. They are the focus point of this chapter.

8.1 Introduction

Let us look once more at the homework problem [H6.2]. In that problem, we simulated a parabolic PDE in one space dimension with a nonlinear boundary condition due to radiation. Because of the nonlinear boundary condition, we required one Newton iteration in a single unknown, T_{21}, per function evaluation. However, since the ODE problem after conversion of the PDE problem using the MOL approach is stiff, we also must employ an implicit integration algorithm, such as a BDF method. Consequently, we require a second Newton iteration over many variables once every integration step. Finally, if the simulation is to be error–controlled, we may need to reject some of the integration steps after the two Newton iterations converged, in order to repeat the step with a reduced step size. The three simulation loops are illustrated in Fig.8.1.

How accurately should we perform all these iterations? Clearly, if the relative error requested for the numerical integration is to be met by the outermost loop, then the internal loops must be computed at least as accurately. On the other hand, if e.g. the iteration of the integration algorithm is still far away from convergence, why should we perform the internal iteration within the individual function evaluation very accurately already?

Clearly, the different iterations and tolerances are closely interrelated. It seems awkward that we should have to keep track of different iterations and different error tolerances that are all part of one and the same process. Maybe, we should take a step back and reconsider all these issues in the

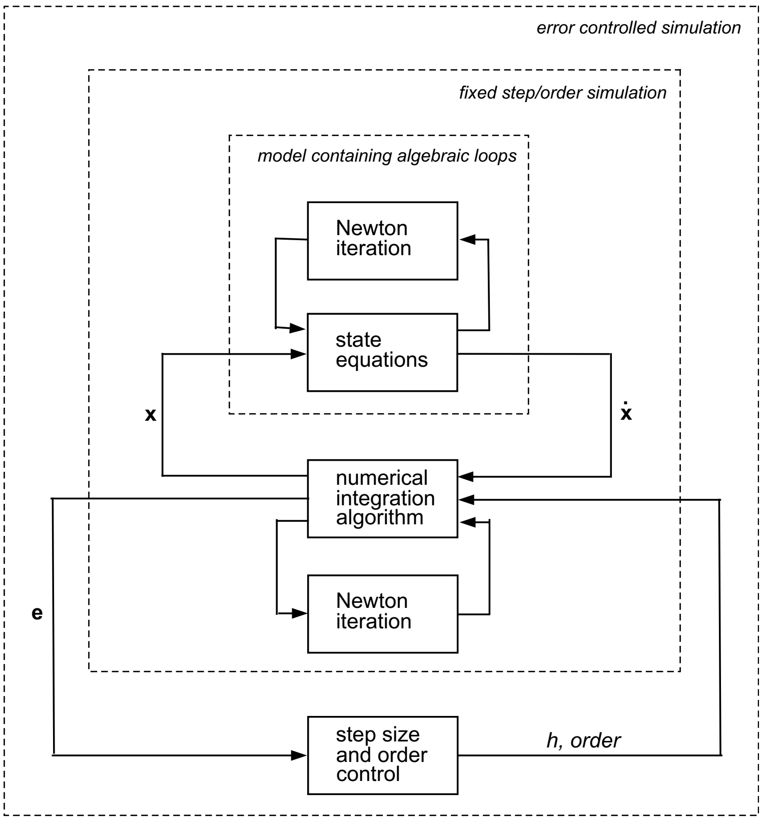
error controlled simulation
fixed step/order simulation
model containing algebraic loops
Newton iteration
state equations
x
ẋ
numerical integration algorithm
Newton iteration
e
step size and order control
h, order

FIGURE 8.1. The three simulation loops.

new light of their seeming complexity to ascertain whether this complexity is truly necessary.

We may start by asking ourselves, whether algebraic loops, or any other numerical processes that require iterations, do really exist in this physical world of ours. Isn't the physical world truly *causal*, i.e., isn't it true that each event has one or several causes, and that a strictly sequential ordering is possible between causes and effects? Don't iterations defy the principle of strict causality?

Mutual causal dependencies do indeed exist in physics and are rather common. The relationship between voltage and current in a resistor is non–causal. It is not true that the potential difference at the two ends of the resistor makes current flow, or that the current flowing through the resistor causes a voltage drop. These are simply two different facets of one and the same physical phenomenon. Yet "causal loops" do not truly exist in the physical world. If we place two resistors in series, this will create an algebraic loop in our model. Yet, physics doesn't understand the concept of a loop. The idea of a loop implies a sequence of execution, i.e., *a* causes *b*, which in turn causes *c*, which is responsible for *a*. Physics doesn't understand the concept of a "sequence of execution." Physics is by its very nature completely non–causal. All phenomena observed are byproducts of the big balance equations that we call the conservation principles: conservation of energy, conservation of mass, and conservation of momentum.

If "causal loops" show up in our models, they are artifactual. They are byproducts of the way in which we are dealing with the equations. Our way of thinking is strictly cause–effect oriented, and this is also how we have built our digital computers. We try to turn everything into cause–effect relationships. Sometimes, this is not possible. Causal loops, and the need for iteration, are our way of expressing this problem. Clearly, there does not exist a natural (physical) way of looking at causal loops.

Let us go back right to the foundations of continuous system simulation, and ask ourselves where the so–called state–space description of a physical system came from. It originated with the desire to separate the process of *modeling* (in a simple–minded way of looking at things, the process of generating a state–space model out of physical observations) from that of *simulation* (the process of translating the state–space model into trajectory behavior).

It all made sense. In the context of using explicit integration algorithms (all integration algorithms that were used in the early days were explicit algorithms), this separation comes quite naturally. The state–space model computes $\dot{\mathbf{x}}(t_k)$ out of $\mathbf{x}(t_k)$, and the integration algorithm in turn computes $\mathbf{x}(t_{k+1})$ out of $\mathbf{x}(t_k)$ and $\dot{\mathbf{x}}(t_k)$ — a meaningful and clean separation of duties.

By the time implicit integration algorithms were introduced, this separation was no longer as clean and crisp and beautiful. We suddenly had to deal with a causal loop, since the state–space model and the integration algorithm now operated on the same time instant, i.e., they had to co–operate to find *simultaneously* $\mathbf{x}(t_{k+1})$ and $\dot{\mathbf{x}}(t_{k+1})$. However, tradition had imprinted this separation so deeply into the brains of the simulation practitioners of that epoch that no–one bothered to raise the question whether this separation was still useful, or whether it might not even be detrimental to our task.

Let us check what happens if we let go of the constraint that state–space models have to be formulated such that they compute the state derivatives explicitly. Instead, we are going to use the implicit model:

$$\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{u}, t) = 0.0 \tag{8.1}$$

Let us apply the BDF3 algorithm:

$$\mathbf{x_{k+1}} = \frac{6}{11} h \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} + \frac{18}{11}\mathbf{x_k} - \frac{9}{11}\mathbf{x_{k-1}} + \frac{2}{11}\mathbf{x_{k-2}} \tag{8.2}$$

to the model of Eq.(8.1). We can solve Eq.(8.2) for $\dot{\mathbf{x}}_{\mathbf{k+1}}$:

$$\dot{\mathbf{x}}_{\mathbf{k+1}} = \frac{1}{h}\left[\frac{11}{6}\cdot\mathbf{x}_{\mathbf{k+1}} - 3\mathbf{x}_{\mathbf{k}} + \frac{3}{2}\mathbf{x}_{\mathbf{k-1}} - \frac{1}{3}\mathbf{x}_{\mathbf{k-2}}\right] \tag{8.3}$$

and plug Eq.(8.3) into Eq.(8.1). We obtain a nonlinear vector equation $\mathbf{f}$ in the unknown parameter vector $\mathbf{x}_{\mathbf{k+1}}$:

$$\mathcal{F}(\mathbf{x}_{\mathbf{k+1}}) = 0.0 \tag{8.4}$$

which can be solved *directly* by Newton iteration.

In this new formulation, the distinction between iterating on the implicit integration step and iterating on nonlinear function evaluations has vanished. It becomes quite evident that these were not two separate processes, but only two different facets of one and the same process. It is this –very fruitful– idea, which had first been proposed by Bill Gear in 1971 in a frequently cited paper [8.13], that we shall pursue in this chapter in more detail.

8.2 Multi–step Formulae

We have learnt that implicit integration algorithms are most useful for dealing with stiff systems. Consequently, a DAE formulation in place of the former ODE formulation will be particularly fruitful in the context of the simulation of stiff systems, and it should contain a numerical formula that has been designed for dealing with stiff systems, such as a BDF algorithm.

Since we ultimately want to solve for $\mathbf{x}_{\mathbf{k+1}}$, we eliminate $\dot{\mathbf{x}}_{\mathbf{k+1}}$ from the implicit state–space model of Eq.(8.1), and this means that the integration formula will now have to be solved for $\mathbf{f}_{\mathbf{k+1}}$ rather than for $\mathbf{x}_{\mathbf{k+1}}$ as in the ODE case. Thus, we are now looking at *numerical differentiation formulae* rather than *numerical integration formulae.*

We have already seen what a DAE implementation of a BDF algorithm could look like. In order to assess the validity of this approach, we should ask ourselves what the stability and accuracy properties of such a BDF implementation are.

Let us start with a discussion of stability properties. The linear version of Eq.(8.1) can now be written as:

$$\mathbf{A}\cdot\mathbf{x} + \mathbf{B}\cdot\dot{\mathbf{x}} = 0.0 \tag{8.5}$$

Thus, our standard linear test problem has now two matrices, $\mathbf{A}$ and $\mathbf{B}$. Plugging Eq.(8.2) into Eq.(8.5), we obtain:

$$\mathbf{A}\cdot\mathbf{x}_{\mathbf{k+1}} + \frac{11\mathbf{B}}{6h}\cdot\left(\mathbf{x}_{\mathbf{k+1}} - \frac{18}{11}\mathbf{x}_{\mathbf{k}} + \frac{9}{11}\mathbf{x}_{\mathbf{k-1}} - \frac{2}{11}\mathbf{x}_{\mathbf{k-2}}\right) = 0.0 \tag{8.6}$$

or:

$$\left(-\mathbf{B} - \frac{6\mathbf{A}h}{11}\right)\mathbf{x_{k+1}} = -\frac{18\mathbf{B}}{11}\mathbf{x_k} + \frac{9\mathbf{B}}{11}\mathbf{x_{k-1}} - \frac{2\mathbf{B}}{11}\mathbf{x_{k-2}} \tag{8.7}$$

We already know that Newton iteration does not affect the stability domain of a method. Thus, we can solve Eq.(8.7) for $\mathbf{x_{k+1}}$ by use of matrix inversion without modifying the stability domain of the method. We find:

$$\mathbf{x_{k+1}} = \left(-\mathbf{B} - \frac{6\mathbf{A}h}{11}\right)^{-1} \cdot \left(-\frac{18\mathbf{B}}{11}\mathbf{x_k} + \frac{9\mathbf{B}}{11}\mathbf{x_{k-1}} - \frac{2\mathbf{B}}{11}\mathbf{x_{k-2}}\right) \tag{8.8}$$

Let us first discuss the simplest case:

$$\mathbf{B} = -\mathbf{I^{(n)}} \tag{8.9}$$

In this case, Eq.(8.5) degenerates to the explicit linear test problem:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} \tag{8.10}$$

and Eq.(8.8) becomes:

$$\mathbf{x_{k+1}} = \left(\mathbf{I^{(n)}} - \frac{6\mathbf{A}h}{11}\right)^{-1} \cdot \left(\frac{18}{11}\mathbf{x_k} - \frac{9}{11}\mathbf{x_{k-1}} + \frac{2}{11}\mathbf{x_{k-2}}\right) \tag{8.11}$$

which is identical to the equation that had been used in Chapter 4 to determine the stability domain of BDF3. Consequently, at least in this simple situation, the stability domain is not at all affected by the DAE formulation.

Let us assume next that $\mathbf{B}$ is a non–singular matrix. In this case, Eq.(8.5) can be rewritten as:

$$\dot{\mathbf{x}} = -\mathbf{B}^{-1} \cdot \mathbf{A} \cdot \mathbf{x} \tag{8.12}$$

Will the inversion of $\mathbf{B}$ have an effect on the stability domain? We can determine the stability domain of the method in the following way. We choose the eigenvalues of $-\mathbf{B}^{-1} \cdot \mathbf{A}$ along the unit circle of the complex plane, then apply the so found $\mathbf{A}$– and $\mathbf{B}$–matrices to the $\mathbf{F}$–matrix:

$$\mathbf{F} = \begin{pmatrix} \mathbf{O^{(n)}} & \mathbf{I^{(n)}} & \mathbf{O^{(n)}} \\ \mathbf{O^{(n)}} & \mathbf{O^{(n)}} & \mathbf{I^{(n)}} \\ \frac{2}{11}\left(\mathbf{B} + \frac{6}{11}\mathbf{A}h\right)^{-1}\mathbf{B} & -\frac{9}{11}\left(\mathbf{B} + \frac{6}{11}\mathbf{A}h\right)^{-1}\mathbf{B} & \frac{18}{11}\left(\mathbf{B} + \frac{6}{11}\mathbf{A}h\right)^{-1}\mathbf{B} \end{pmatrix} \tag{8.13}$$

and determine h such that the dominant eigenvalues of $\mathbf{F}$ are on the unit circle. We arbitrarily chose several different non–singular $\mathbf{B}$–matrices of dimensions 2×2, and computed the corresponding $\mathbf{A}$–matrices using:

$$\mathbf{A} = -\mathbf{B} \cdot \begin{pmatrix} 0 & 1 \\ -1 & 2\cos(\alpha) \end{pmatrix} \tag{8.14}$$

We then plugged these matrices into Eq.(8.13), and computed the stability domains. It turned out that, in every single case, the stability domain was exactly the same as in the ODE case. This is generally true. Non–singular **B**–matrices do not influence the numerical stability properties of the method in any way.

These are good news indeed. Notice that we just solved the *algebraic loop problem* once and for all — at least in the context of stiff system simulation. There is no longer any need to apply a Newton iteration to algebraic loops that form part of the state–space model, and then apply a separate Newton iteration around the first one for bringing the implicit integration scheme to convergence. The two iterations have turned out to be two different facets of one and the same process.

Let us now look at the case where **B** is singular. Let the rank of **B** be $r < n$. We can then perform a singular value decomposition on the matrix **B**, as indicated in Fig.8.2.

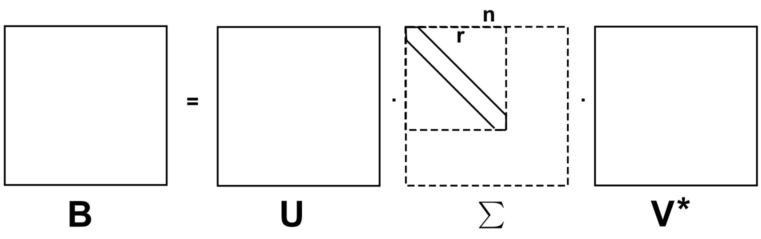

FIGURE 8.2. Singular value decomposition.

where **U** and **V** are two unitary matrices (each row vector is orthogonal to all other row vectors and of length 1.0, the same applies to all column vectors), and **Σ** is a diagonal matrix. Since both **U** and **V** have full rank, i.e.:

$$\text{rank}(\mathbf{U}) = \text{rank}(\mathbf{V}) = n \tag{8.15}$$

the **Σ**–matrix has the same rank as **B**, thus:

$$\text{rank}(\mathbf{\Sigma}) = \text{rank}(\mathbf{B}) = r \tag{8.16}$$

$\mathbf{V}^*$ denotes the Hermitian transpose (the conjugate complex transpose) of **V**. Since:

$$\mathbf{B} = \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^* \tag{8.17}$$

Eq.(8.5) becomes:

$$\mathbf{A} \cdot \mathbf{x} + \mathbf{U} \cdot \mathbf{\Sigma} \cdot \mathbf{V}^* \cdot \dot{\mathbf{x}} = 0.0 \tag{8.18}$$

Since the inverse of a unitary matrix is its Hermitian transpose, we can rewrite Eq.(8.18) as:

$$\mathbf{U}^* \cdot \mathbf{A} \cdot \mathbf{x} + \mathbf{\Sigma} \cdot \mathbf{V}^* \cdot \dot{\mathbf{x}} = 0.0 \tag{8.19}$$

Let us now perform a variable substitution:

$$\mathbf{z} = \mathbf{V}^* \cdot \mathbf{x} \tag{8.20}$$

Plugging Eq.(8.20) into Eq.(8.19), we obtain:

$$\mathbf{U}^* \cdot \mathbf{A} \cdot \mathbf{V} \cdot \mathbf{z} + \mathbf{\Sigma} \cdot \dot{\mathbf{z}} = 0.0 \tag{8.21}$$

or:

$$\tilde{\mathbf{A}} \cdot \mathbf{z} + \mathbf{\Sigma} \cdot \dot{\mathbf{z}} = 0.0 \tag{8.22}$$

A graphical representation of Eq.(8.22) is shown in Fig.8.3.

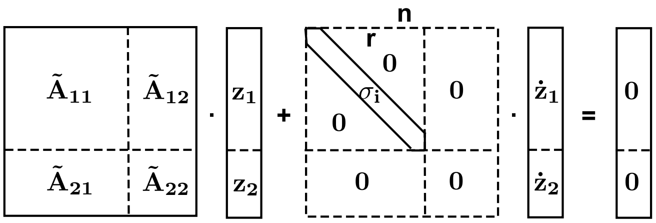
$\tilde{\mathbf{A}}_{11}$
$\tilde{\mathbf{A}}_{12}$
$\tilde{\mathbf{A}}_{21}$
$\tilde{\mathbf{A}}_{22}$
$\mathbf{z_1}$
$\mathbf{z_2}$
n
r
σ_i
0
$\dot{\mathbf{z}}_1$
$\dot{\mathbf{z}}_2$

FIGURE 8.3. Linear test problem after variable substitution.

Eq.(8.22) can be decomposed into:

$$\tilde{\mathbf{A}}_{11} \cdot \mathbf{z_1} + \tilde{\mathbf{A}}_{12} \cdot \mathbf{z_2} + \mathbf{\Sigma}_{11} \cdot \dot{\mathbf{z}}_1 = 0.0 \tag{8.23a}$$

$$\tilde{\mathbf{A}}_{21} \cdot \mathbf{z_1} + \tilde{\mathbf{A}}_{22} \cdot \mathbf{z_2} = 0.0 \tag{8.23b}$$

If $\tilde{\mathbf{A}}_{22}$ is non–singular, we can solve Eq.(8.23b) for $\mathbf{z_2}$:

$$\mathbf{z_2} = -\tilde{\mathbf{A}}_{22}^{-1} \cdot \tilde{\mathbf{A}}_{21} \cdot \mathbf{z_1} \tag{8.24}$$

and plugging Eq.(8.24) into Eq.(8.23a), we obtain:

$$\dot{\mathbf{z}}_1 = \mathbf{\Sigma}_{11}{}^{-1} \cdot \left(\tilde{\mathbf{A}}_{12} \cdot \tilde{\mathbf{A}}_{22}^{-1} \cdot \tilde{\mathbf{A}}_{21} - \tilde{\mathbf{A}}_{11}\right) \cdot \mathbf{z_1} \tag{8.25}$$

In the new state vector, $\mathbf{z}$, it becomes evident that only a subset of its variables, namely the vector $\mathbf{z_1}$ of length r is described by means of differential equations. The remaining variables appear only algebraically in Eq.(8.22). This means that the system does, in reality, not contain n different state variables or energy storages, but only r. Thus, in the terminology introduced in the previous chapter: a singular **B**–matrix corresponds to a *structurally singular model*, i.e., a *higher–index model*.

In the new state vector, $\mathbf{z_1}$, the situation is now the same as before, when **B** was assumed non–singular. Hence the stability domain is indeed not affected at all by the choice of the **B**–matrix.

We need not worry about *accuracy*. Since the numerical differentiation formula is n^{th}–order accurate, and since we have full control over what happens during the Newton iteration, the DAE solver using an n^{th}–order accurate BDF method must obviously as a whole be n^{th}–order accurate.

Until this point, we have focused our interest on DAE formulations of BDF algorithms. What happens if we decide to use other types of implicit multi–step techniques, such as the Adams–Moulton family of methods? Let us look at the case of AM3:

$$\mathbf{x}(t_{k+1}) \approx \mathbf{x}(t_k) + \frac{h}{12}\left(5\mathbf{f}_{k+1} + 8\mathbf{f_k} - \mathbf{f_{k-1}}\right) \tag{8.26}$$

We can turn this formula around, and obtain:

$$\dot{\mathbf{x}}(t_{k+1}) = \mathbf{f_{k+1}} \approx \frac{12}{5h}\left(\mathbf{x}(t_{k+1}) - \mathbf{x}(t_k)\right) - \frac{8}{5}\mathbf{f_k} + \frac{1}{5}\mathbf{f_{k-1}} \tag{8.27}$$

Plugging Eq.(8.27) into Eq.(8.1), we obtain again a nonlinear vector function in the unknown vector $\mathbf{x_{k+1}}$, whereas the quantities $\mathbf{x_k}$, $\mathbf{f_k}$, and $\mathbf{f_{k-1}}$ can be treated as known.

The problems with Eq.(8.27) are that we have eliminated the state derivatives from our system of equations, thus, we don't really know what values to use for $\mathbf{f_k}$ and $\mathbf{f_{k-1}}$. One way to overcome this difficulty is to solve two Newton iterations in sequence:

$$\begin{aligned}\mathcal{F}_1\left(\mathbf{x}(t_{k+1})\right) = \mathbf{f}\Big(&\mathbf{x}(t_{k+1}), \frac{12}{5h}\mathbf{x}(t_{k+1}) - \frac{12}{5h}\mathbf{x}(t_k) - \frac{8}{5}\mathbf{w}(t_k) \\ &+ \frac{1}{5}\mathbf{w}(t_{k-1}), \mathbf{u}(t_{k+1}), t_{k+1}\Big) = 0.0 \qquad (8.28a) \\ \mathcal{F}_2\left(\mathbf{w}(t_{k+1})\right) = \mathbf{f}&\left(\mathbf{x}(t_{k+1}), \mathbf{w}(t_{k+1}), \mathbf{u}(t_{k+1}), t_{k+1}\right) = 0.0 \qquad (8.28b)\end{aligned}$$

Equation (8.28a) determines $\mathbf{x}(t_{k+1})$. In this iteration, $\mathbf{u}(t_{k+1})$, $\mathbf{x}(t_k)$, $\mathbf{w}(t_k)$, and $\mathbf{w}(t_{k-1})$ are assumed known. Equation (8.28b) then evaluates $\mathbf{w}(t_{k+1})$. During that iteration, $\mathbf{x}(t_{k+1})$ can be assumed known as well. Clearly, $\mathbf{w}$ is just another name for $\dot{\mathbf{x}}$.

The BDF case was special, since in the BDF formulae, the state derivative vector shows up only once. In that case, the DAE formulation becomes particularly simple and efficient to implement, and half of the variables (the state derivatives) can be eliminated from the set of variables to be computed. An integration formula that shares this property is called a *one–leg method*. In general, linear one–leg methods can be written for the ODE case of Eq.(2.1) as:

$$\frac{1}{h}\sum_{j=0}^{n}\alpha_j\mathbf{x_{k-j+1}} = \mathbf{f}\left(\sum_{j=0}^{n}\beta_j\mathbf{x_{k-j+1}}, \sum_{j=0}^{n}\beta_j\mathbf{u_{k-j+1}}, \sum_{j=0}^{n}\beta_j t_{k-j+1}\right) \tag{8.29}$$

The left–hand side of Eq.(8.29) represents the state derivative vector evaluated at the intermediate time $\sum_{j=0}^{n} \beta_j t_{k-j+1}$. Equation (8.29) is turned into a numerical integration formula by solving the expression on the left–hand side for $\mathbf{x_{k+1}}$ moving all other terms to the right–hand side.

Equation (8.29) naturally extends to the following DAE formulation:

$$\begin{aligned}\mathcal{F}(\mathbf{x_{k+1}}) &= \mathbf{f}\left(\sum_{j=0}^{n} \beta_j \mathbf{x_{k-j+1}}, \frac{1}{h}\sum_{j=0}^{n} \alpha_j \mathbf{x_{k-j+1}}, \sum_{j=0}^{n} \beta_j \mathbf{u_{k-j+1}}, \sum_{j=0}^{n} \beta_j t_{k-j+1}\right) \\ &= 0.0 \end{aligned} \tag{8.30}$$

Let us now discuss the (linear) stability properties of AM3 in its DAE formulation. To this end, we plug Eqs.(8.28a–b) into Eq.(8.5). We still want to assume $\mathbf{B}$ to be non–singular. We obtain:

$$\begin{aligned}\mathbf{x}(t_{k+1}) \approx \left(\mathbf{B} + \frac{5}{12}\mathbf{A}h\right)^{-1} \cdot \bigg(\mathbf{B}\mathbf{x}(t_k) &+ \frac{2}{3}\mathbf{B}h\mathbf{w}(t_k) \\ &- \frac{1}{12}\mathbf{B}h\mathbf{w}(t_{k-1})\bigg) \end{aligned} \tag{8.31a}$$

$$\mathbf{w}(t_{k+1}) \approx -\mathbf{B}^{-1}\mathbf{A}\mathbf{x}(t_{k+1}) \tag{8.31b}$$

and by letting:

$$\mathbf{z_k} = \begin{pmatrix} \mathbf{x_k} \\ \mathbf{w_{k-1}} \\ \mathbf{w_k} \end{pmatrix} \tag{8.32}$$

we obtain the following $\mathbf{F}$–matrix:

$$\mathbf{F} = \begin{pmatrix} \mathbf{F_{11}} & \mathbf{F_{12}} & \mathbf{F_{13}} \\ \mathbf{O^{(n)}} & \mathbf{O^{(n)}} & \mathbf{I^{(n)}} \\ \mathbf{F_{31}} & \mathbf{F_{32}} & \mathbf{F_{33}} \end{pmatrix} \tag{8.33}$$

where:

$$\mathbf{F_{11}} = \left(\mathbf{B} + \frac{5}{12}\mathbf{A}h\right)^{-1} \mathbf{B} \tag{8.34a}$$

$$\mathbf{F_{12}} = -\frac{1}{12}\left(\mathbf{B} + \frac{5}{12}\mathbf{A}h\right)^{-1} \mathbf{B}h \tag{8.34b}$$

$$\mathbf{F_{13}} = \frac{2}{3}\left(\mathbf{B} + \frac{5}{12}\mathbf{A}h\right)^{-1} \mathbf{B}h \tag{8.34c}$$

$$\mathbf{F_{31}} = -\mathbf{B}^{-1}\mathbf{A}\left(\mathbf{B} + \frac{5}{12}\mathbf{A}h\right)^{-1} \mathbf{B} \tag{8.34d}$$

$$\mathbf{F_{32}} = \frac{1}{12}\mathbf{B}^{-1}\mathbf{A}\left(\mathbf{B} + \frac{5}{12}\mathbf{A}h\right)^{-1} \mathbf{B}h \tag{8.34e}$$

$$\mathbf{F_{33}} = -\frac{2}{3}\mathbf{B}^{-1}\mathbf{A}\left(\mathbf{B} + \frac{5}{12}\mathbf{A}h\right)^{-1} \mathbf{B}h \tag{8.34f}$$

We select **B** arbitrarily as a non–singular 2×2 matrix, and compute **A** in accordance with Eq.(4.37). We checked with different selections of **B**. The results were always the same as shown in Fig.4.2. The (linear) stability behavior of the method was not influenced by the DAE formulation. This may at first look like a surprising result, since the **F**–matrix of Eq.(4.41) used to compute the stability domain shown in Fig.4.2 is a 4×4 matrix, whereas the new **F**–matrix of Eq.(8.33) is a 6×6 matrix. However, $\mathbf{w_k}$ is linear in $\mathbf{x_k}$, thus **F** is singular. All we did was to add two more spurious eigenvalues at the origin. These eigenvalues will never influence the stability behavior. Eigenvalues located at the origin of the discrete system correspond to eigenvalues of the continuous system located at $-\infty$. Such eigenvalues are completely harmless.

We have discussed how linear implicit multi–step methods can be converted from an ODE formulation to a DAE formulation by solving the formulae for $\dot{\mathbf{x}}_{\mathbf{k+1}}$ instead of for $\mathbf{x_{k+1}}$. This leads to two separate Newton iterations in n variables each, where n is the order of the system to be simulated. In the case of the one–leg methods, such as the BDF formulae, the DAE formulation becomes particularly simple, since the state derivative vector can be eliminated from the model, and one of the two Newton iterations becomes unnecessary.

How about explicit multi–step formulae? Let us look at AB3:

$$\mathbf{x}(t_{k+1}) \approx \mathbf{x}(t_k) + \frac{h}{12}\left(23\mathbf{f}_k - 16\mathbf{f}_{k-1} + 5\mathbf{f}_{k-2}\right) \tag{8.35}$$

Turning Eq.(8.35) around, we obtain:

$$\dot{\mathbf{x}}(t_{k+1}) \approx \frac{12}{23h}\left(\mathbf{x}(t_{k+2}) - \mathbf{x}(t_{k+1})\right) + \frac{16}{23}\mathbf{f}_k - \frac{5}{23}\mathbf{f}_{k-1} \tag{8.36}$$

Thus, *explicit numerical integration* formulae turn in the conversion to DAE form into *overimplicit numerical differentiation* formulae. Using such a formula will turn the entire simulation run into one giant iteration loop, which is certainly not justifiable.

However, this brings up another idea. We could search for overimplicit numerical integration schemes, which, until now, were quite useless, and turn those around for use in a DAE solver.

The third–order accurate overimplicit Adams formula is:

$$\mathbf{x}(t_{k+1}) \approx \mathbf{x}(t_k) + \frac{h}{12}\left(-\mathbf{f}(t_{k+2}) + 8\mathbf{f}(t_{k+1}) + 5\mathbf{f}(t_k)\right) \tag{8.37}$$

Solving for the newest state derivative, we obtain:

$$\dot{\mathbf{x}}(t_{k+1}) \approx \frac{12}{h}\left(\mathbf{x}(t_{k-1}) - \mathbf{x}(t_k)\right) + 8\mathbf{f}(t_k) + 5\mathbf{f}(t_{k-1}) \tag{8.38}$$

which is a third–order accurate explicit numerical differentiation formula.

Similarly, we can find a third–order accurate overimplicit BDF formula:

$$\mathbf{x}(t_{k+1}) \approx \frac{57}{26}\mathbf{x}(t_k) - \frac{21}{13}\mathbf{x}(t_{k-1}) + \frac{11}{26}\mathbf{x}(t_{k-2}) + \frac{6h}{26}\mathbf{f}(t_{k+2}) \tag{8.39}$$

which leads to the explicit third–order accurate numerical differentiation formula:

$$\dot{\mathbf{x}}(t_{k+1}) \approx \frac{1}{6h}\left(26\mathbf{x}(t_k) - 57\mathbf{x}(t_{k-1}) + 42\mathbf{x}(t_{k-2}) - 11\mathbf{x}(t_{k-3})\right) \tag{8.40}$$

Unfortunately, both formulae are unstable in the vicinity of the origin when plugged into the DAE. Explicit numerical differentiation is not a recommended procedure because of its poor stability properties and should therefore be avoided. If numerical differentiation is a necessity in a simulation model (e.g., if an input variable needs to be differentiated), we strongly suggest the use of an implicit numerical differentiation formula together with a DAE formulation for the overall simulation problem.

We have learnt that implicit multi–step formulae lend themselves splendidly for use in DAE solvers. To this end, they simply need to be turned around and solved for the newest value of the state derivative vector instead of the newest value of the state vector. Both AMi and BDFi formulae can be used in DAE solvers. However, the BDF formulae are more attractive since they, being one–legged formulae, allow to eliminate the state derivative vector from the simulation program altogether.

We remember that BDFi formulae are inefficient for use in non–stiff ODEs due to their poor accuracy properties. This was documented in Fig.6.12. The problem certainly hasn't vanished by reformulating the model in a DAE format. Thus, we might suspect that the AMi formulae will still work better than the BDFi formulae also in non–stiff DAE simulation. However, whether this is true or not will depend on the relative cost to be paid for the second Newton iteration. Let us ponder this question.

We shall rerun the wave equation example of Eqs.(6.53a–e), this time in DAE format, using once a BDF3 and once an AM3 algorithm. We computed the global accuracy of the two algorithms using the same step sizes as in Table 6.4. The results are tabulated in Table 8.1.

Figure 8.4 shows these results graphically.

Although the entries in Table 8.1 look *exactly* like the corresponding entries in Table 6.4 (after all, these *are* the same methods applied to the same problem), the graph of Fig.8.4 looks a little different from that of Fig.6.12 due to the need for a second Newton iteration in the case of the DAE formulation of AM3.

Of course in the given example, the Newton iterations converge in a single step since the problem is linear and the Jacobian has been computed accurately. Whereas the "economy" of the BDF3 algorithm in terms of the number of function evaluations required does not change between the ODE and DAE formulations, the AM3 algorithm has become more expensive

h	BDF3	AM3
0.1	garbage	unstable
0.05	garbage	unstable
0.02	garbage	unstable
0.01	garbage	unstable
0.005	0.9469e-2	0.8783e-8
0.002	0.1742e-6	0.2149e-8
0.001	0.4363e-7	0.2120e-8

TABLE 8.1. Comparison of accuracy of integration algorithms.

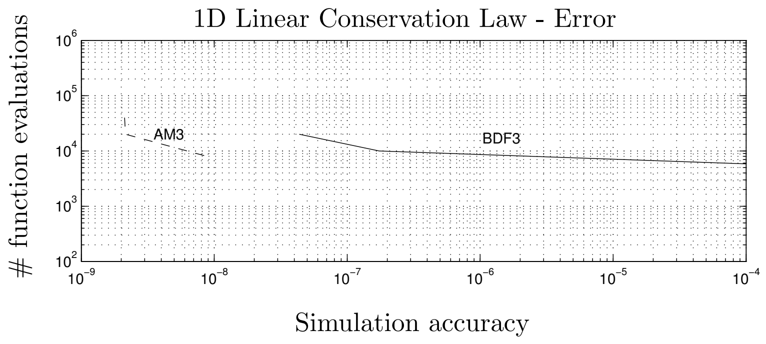

FIGURE 8.4. Cost–versus–accuracy plot for the 1D wave equation.

to compute in the DAE formulation due to the need for a second Newton iteration. The lesson to be learnt from this exercise: at least in the non–stiff case, it may well be worthwhile to convert the model first to ODE format, using the techniques described in the previous chapter of this book, before simulating it.

One problem we haven't discussed yet concerns the *initial conditions*. In the ODE case, it was sufficient for the user to specify initial values for the state vector **x** and for the input vector **u**, and the state derivative vector could then be computed from the state–space model. In the DAE case, we don't have an explicit formula to compute the state derivative vector.

If we decide to use the BDF technique (or any other one–legged algorithm), we can eliminate the state derivative vector from the model altogether, and in this special case, we don't have a problem ... except during the startup period. Of course, we can use order control during the startup period, and then, we won't need the state derivative vector ever. Let us explain. The implicit differentiation formulae using the inverted BDF algorithms are:

$$\dot{\mathbf{x}}(t_{k+1}) \approx \frac{1}{h}\mathbf{x}(t_{k+1}) - \frac{1}{h}\mathbf{x}(t_k) \tag{8.41a}$$

$$\dot{\mathbf{x}}(t_{k+1}) \approx \frac{3}{2h}\mathbf{x}(t_{k+1}) - \frac{2}{h}\mathbf{x}(t_k) + \frac{1}{2h}\mathbf{x}(t_{k-1}) \tag{8.41b}$$

$$\dot{\mathbf{x}}(t_{k+1}) \approx \frac{11}{6h}\mathbf{x}(t_{k+1}) - \frac{3}{h}\mathbf{x}(t_k) + \frac{3}{2h}\mathbf{x}(t_{k-1}) - \frac{1}{3h}\mathbf{x}(t_{k-2}) \tag{8.41c}$$

where Eq.(8.41a) is first–order accurate, Eq.(8.41b) is second–order accurate, and Eq.(8.41c) is third–order accurate. If we wish to simulate the DAE model using a third–order accurate formula, we can use Eq.(8.41a) during the first step, then Eq.(8.41b) during the second, and from then on, we can use Eq.(8.41c). Plugging Eqs.(8.41a–c) into Eq.(8.1), we obtain:

$$\mathbf{f}\left(\mathbf{x_1}, \frac{1}{h}\mathbf{x_1} - \frac{1}{h}\mathbf{x_0}, \mathbf{u_1}, t_1\right) = 0.0 \tag{8.42a}$$

$$\mathbf{f}\left(\mathbf{x_2}, \frac{3}{2h}\mathbf{x_2} - \frac{2}{h}\mathbf{x_1} + \frac{1}{2h}\mathbf{x_0}, \mathbf{u_2}, t_2\right) = 0.0 \tag{8.42b}$$

$$\mathbf{f}\left(\mathbf{x_3}, \frac{11}{6h}\mathbf{x_3} - \frac{3}{h}\mathbf{x_2} + \frac{3}{2h}\mathbf{x_1} - \frac{1}{3h}\mathbf{x_0}, \mathbf{u_3}, \mathbf{t_3}\right) = 0.0 \tag{8.42c}$$

$$\mathbf{f}\left(\mathbf{x_4}, \frac{11}{6h}\mathbf{x_4} - \frac{3}{h}\mathbf{x_3} + \frac{3}{2h}\mathbf{x_2} - \frac{1}{3h}\mathbf{x_1}, \mathbf{u_4}, t_4\right) = 0.0 \tag{8.42d}$$

etc.

In each step, we perform one Newton iteration in the unknown state vector at the current time. In this way, the state derivative vector has been eliminated from the model once and for all.

Unfortunately using this approach, we are faced with the meanwhile well–known accuracy problems. We shall have to employ a very small step size initially in order to be able to meet our accuracy requirements. Moreover, the approach won't work in the case of the AMi algorithms. Those algorithms don't eliminate the state derivative vector (the $\mathbf{w}$–vector of Eqs.(8.28a–b)), and we need to find an estimate for $\mathbf{w_0}$ at time t_0 by means of Newton iteration. The problem here is that there is no guarantee that the Newton iteration will converge at all or will converge to the right solution if our initial guesses for the state derivative values are far off. Therefore, most DAE solvers on the market request that the user specify not only the initial values for the state vector, but also good initial guesses for the state derivative vector to be used as starting values for the first Newton iteration on Eq.(8.1) at time t_0.

How about using higher–order Runge–Kutta algorithms for startup? This may turn out to again be a smart idea, but we need to postpone the discussion of this approach until we have talked about the DAE format of the single–step algorithms.

Step–size control, order control, and the readout problem don't cause any difficulties beyond those that were already discussed in Chapter 4 of this text.

8.3 Single–step Formulae

In principle, DAE formulations of all single–step algorithms are straightforward. For example, a DAE formulation of our standard explicit RK4 algorithm could be implemented in the following way:

$$\begin{aligned}
&\mathbf{f}\left(\mathbf{x_k}, \mathbf{k_1}, \mathbf{u}(t_k), t_k\right) = 0.0 \\
&\mathbf{f}\left(\mathbf{x_k} + \frac{h}{2}\mathbf{k_1}, \mathbf{k_2}, \mathbf{u}(t_k + \frac{h}{2}), t_k + \frac{h}{2}\right) = 0.0 \\
&\mathbf{f}\left(\mathbf{x_k} + \frac{h}{2}\mathbf{k_2}, \mathbf{k_3}, \mathbf{u}(t_k + \frac{h}{2}), t_k + \frac{h}{2}\right) = 0.0 \\
&\mathbf{f}\left(\mathbf{x_k} + h\mathbf{k_3}, \mathbf{k_4}, \mathbf{u}(t_k + h), t_k + h\right) = 0.0 \\
&\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{h}{6}\left(\mathbf{k_1} + 2\mathbf{k_2} + 2\mathbf{k_3} + \mathbf{k_4}\right)
\end{aligned}$$

Thus, this is exactly the same formula that we were using in Chapter 3 of this text, except that each and every formerly explicit evaluation of the state derivative vector needs to be replaced by a Newton iteration.

In general, this is too costly. Let us take the example of BI3. BI3 contains one explicit RK3 step forward and approximately four RK3 steps backward due to the Newton iteration. Thus, BI3 contains altogether five RK3 steps, each requiring three function evaluations. Consequently, BI3 calls for 15 function evaluations per step. This was for the ODE formulation. However, in the DAE formulation, each of these function evaluations turns itself into a Newton iteration requiring approximately four function evaluations, thus, we are now looking at 60 function evaluations per step. If nothing else killed the efficiency of the BI algorithms, this certainly will.

None of the techniques discussed in Chapter 3 will lead to efficient DAE implementations. The techniques that are least affected by the DAE formulation are the *Richardson extrapolation techniques*. They won't become less efficient by the DAE formulation ... but they had been terribly inefficient already for the ODE case.

Is it hopeless then? Salvation comes from the *fully–implicit Runge–Kutta algorithms* [8.16]. Let us look at one type of these algorithms, namely the *Radau IIA algorithms*. They can be represented by the following Butcher tableaus:

$$\begin{array}{c|cc}
1/3 & 5/12 & \text{-}1/12 \\
1 & 3/4 & 1/4 \\
\hline
x & 3/4 & 1/4
\end{array}$$

$$\begin{array}{c|ccc}
\frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\
\frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} \\
1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\
\hline
x & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9}
\end{array}$$

The method with the smaller Butcher tableau is a third–order accurate fully–implicit two–stage Runge–Kutta algorithm, whereas the method with the larger Butcher tableau is a fifth–order accurate fully–implicit three–stage Runge–Kutta algorithm.

The Butcher tableau of the third–order accurate method can be interpreted in the ODE case as:

$$\mathbf{k_1} = \mathbf{f}\left(\mathbf{x_k} + \frac{5h}{12}\mathbf{k_1} - \frac{h}{12}\mathbf{k_2}, \mathbf{u}(t_k + \frac{h}{3}), t_k + \frac{h}{3}\right) \tag{8.43a}$$

$$\mathbf{k_2} = \mathbf{f}\left(\mathbf{x_k} + \frac{3h}{4}\mathbf{k_1} + \frac{h}{4}\mathbf{k_2}, \mathbf{u}(t_k + h), t_k + h\right) \tag{8.43b}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{h}{4}\left(3\mathbf{k_1} + \mathbf{k_2}\right) \tag{8.43c}$$

In the DAE formulation, the method can be written as:

$$\mathbf{f}\left(\mathbf{x_k} + \frac{5h}{12}\mathbf{k_1} - \frac{h}{12}\mathbf{k_2}, \mathbf{k_1}, \mathbf{u}(t_k + \frac{h}{3}), t_k + \frac{h}{3}\right) = 0.0 \tag{8.44a}$$

$$\mathbf{f}\left(\mathbf{x_k} + \frac{3h}{4}\mathbf{k_1} + \frac{h}{4}\mathbf{k_2}, \mathbf{k_2}, \mathbf{u}(t_k + h), t_k + h\right) = 0.0 \tag{8.44b}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{h}{4}\left(3\mathbf{k_1} + \mathbf{k_2}\right) \tag{8.44c}$$

There is hardly any difference between the two formulations. In both cases, we are faced with a set of $2n$ coupled nonlinear equations in the $2n$ unknowns $\mathbf{k_1}$ and $\mathbf{k_2}$ that need to be solved simultaneously by Newton iteration. Just as in the case of the AMi algorithms, we need to provide the system with not only initial conditions for the state vector $\mathbf{x}(t_0)$, but also with a good estimate of the state derivative vector $\dot{\mathbf{x}}(t_0)$. We set initially:

$$\mathbf{k_1} = \mathbf{k_2} = \dot{\mathbf{x}}(t_0) \tag{8.45}$$

Let us now look at the accuracy and stability properties of the Radau IIA algorithms. To this end, we plug Eqs.(8.43a–c) into the linear test problem. We shall work with the ODE version, since it doesn't make any difference, which formulation we use. We obtain:

$$\mathbf{k_1} = \mathbf{A}\left(\mathbf{x_k} + \frac{5h}{12}\mathbf{k_1} - \frac{h}{12}\mathbf{k_2}\right) \tag{8.46a}$$

$$\mathbf{k_2} = \mathbf{A}\left(\mathbf{x_k} + \frac{3h}{4}\mathbf{k_1} + \frac{h}{4}\mathbf{k_2}\right) \tag{8.46b}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{h}{4}\left(3\mathbf{k_1} + \mathbf{k_2}\right) \tag{8.46c}$$

or solved for the unknowns $\mathbf{k_1}$ and $\mathbf{k_2}$:

$$\mathbf{k_1} = \left[\mathbf{I^{(n)}} - \frac{2\mathbf{A}h}{3} + \frac{(\mathbf{A}h)^2}{6}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \frac{\mathbf{A}h}{3}\right) \cdot \mathbf{A} \cdot \mathbf{x_k} \tag{8.47a}$$

$$\mathbf{k_2} = \left[\mathbf{I^{(n)}} - \frac{2\mathbf{A}h}{3} + \frac{(\mathbf{A}h)^2}{6}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} + \frac{\mathbf{A}h}{3}\right) \cdot \mathbf{A} \cdot \mathbf{x_k} \tag{8.47b}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{h}{4}\left(3\mathbf{k_1} + \mathbf{k_2}\right) \tag{8.47c}$$

and therefore:

$$\mathbf{F} = \mathbf{I^{(n)}} + \left[\mathbf{I^{(n)}} - \frac{2\mathbf{A}h}{3} + \frac{(\mathbf{A}h)^2}{6}\right]^{-1} \cdot \left(\mathbf{I^{(n)}} - \frac{\mathbf{A}h}{6}\right) \cdot (\mathbf{A}h) \tag{8.48}$$

Developing the denominator into a Taylor Series around $h = 0.0$, we find:

$$\mathbf{F} \approx \mathbf{I^{(n)}} + \mathbf{A}h + \frac{(\mathbf{A}h)^2}{2} + \frac{(\mathbf{A}h)^3}{6} + \frac{(\mathbf{A}h)^4}{36} \tag{8.49}$$

Thus, the method is indeed third–order accurate (we proved this at least for linear systems), and the error coefficient is:

$$\varepsilon = \frac{1}{72}(\mathbf{A}h)^4 \tag{8.50}$$

The fifth–order accurate Radau IIA method is characterized by the following **F**–matrix:

$$\mathbf{F} = \mathbf{I^{(n)}} + \left[\mathbf{I^{(n)}} - \frac{3\mathbf{A}h}{5} + \frac{3(\mathbf{A}h)^2}{20} - \frac{(\mathbf{A}h)^3}{60}\right]^{-1} \left(\mathbf{I^{(n)}} - \frac{\mathbf{A}h}{10} + \frac{(\mathbf{A}h)^2}{60}\right) \mathbf{A}h \tag{8.51}$$

Developing the denominator into a Taylor Series around $h = 0.0$, we find:

$$\mathbf{F} \approx \mathbf{I^{(n)}} + \mathbf{A}h + \frac{(\mathbf{A}h)^2}{2} + \frac{(\mathbf{A}h)^3}{6} + \frac{(\mathbf{A}h)^4}{24} + \frac{(\mathbf{A}h)^5}{120} + \frac{11(\mathbf{A}h)^6}{7200} \tag{8.52}$$

Thus, the method is indeed fifth–order accurate (at least for linear systems), and the error coefficient is:

$$\varepsilon = \frac{1}{7200}(\mathbf{A}h)^6 \tag{8.53}$$

A frequently used fourth–order accurate fully–implicit Runge–Kutta algorithm is *Lobatto IIIC* with the Butcher tableau:

$$\begin{array}{c|ccc} 0 & 1/6 & -1/3 & 1/6 \\ 1/2 & 1/6 & 5/12 & -1/12 \\ 1 & 1/6 & 2/3 & 1/6 \\ \hline x & 1/6 & 2/3 & 1/6 \end{array}$$

This method is characterized by the **F**–matrix:

$$\mathbf{F} = \mathbf{I}^{(\mathbf{n})} + \left[\mathbf{I}^{(\mathbf{n})} - \frac{3\mathbf{A}h}{4} + \frac{(\mathbf{A}h)^2}{4} - \frac{(\mathbf{A}h)^3}{24}\right]^{-1} \left(\mathbf{I}^{(\mathbf{n})} - \frac{\mathbf{A}h}{4} + \frac{(\mathbf{A}h)^2}{24}\right) \mathbf{A}h \tag{8.54}$$

Developing the denominator into a Taylor Series around $h = 0.0$, we find:

$$\mathbf{F} \approx \mathbf{I}^{(\mathbf{n})} + \mathbf{A}h + \frac{(\mathbf{A}h)^2}{2} + \frac{(\mathbf{A}h)^3}{6} + \frac{(\mathbf{A}h)^4}{24} + \frac{(\mathbf{A}h)^5}{96} \tag{8.55}$$

Thus, the method is indeed fourth–order accurate (at least for linear systems), and the error coefficient is:

$$\varepsilon = \frac{1}{480}(\mathbf{A}h)^5 \tag{8.56}$$

We plugged the three **F**–matrices into our general–purpose stability domain plotting routine. The results are shown in Fig.8.5.

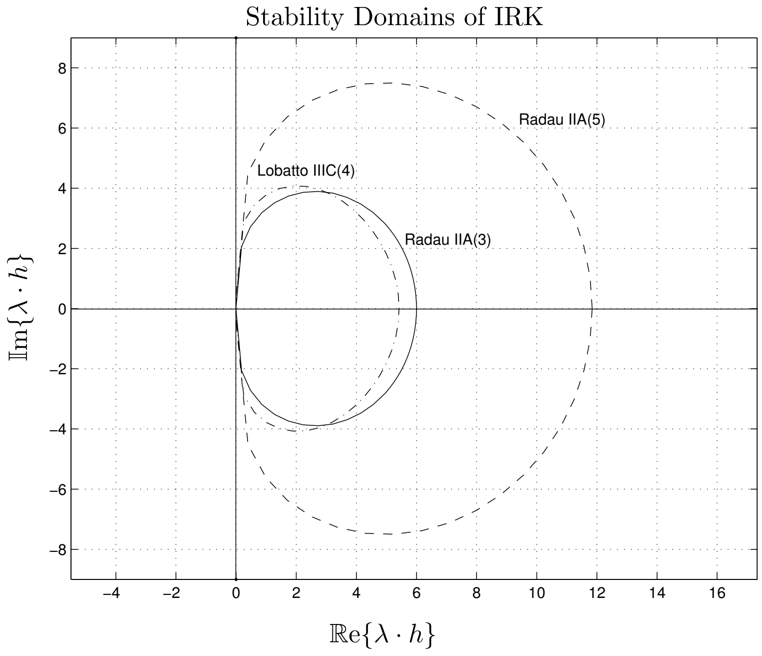

FIGURE 8.5. Stability domains of fully–implicit Runge–Kutta algorithms.

All three methods are A–stable, a desirable property that we hadn't been able to achieve with the higher–order BDF algorithms. The Radau techniques exhibit a somewhat larger unstable region in the right–half complex plane, which may be profitable at times.

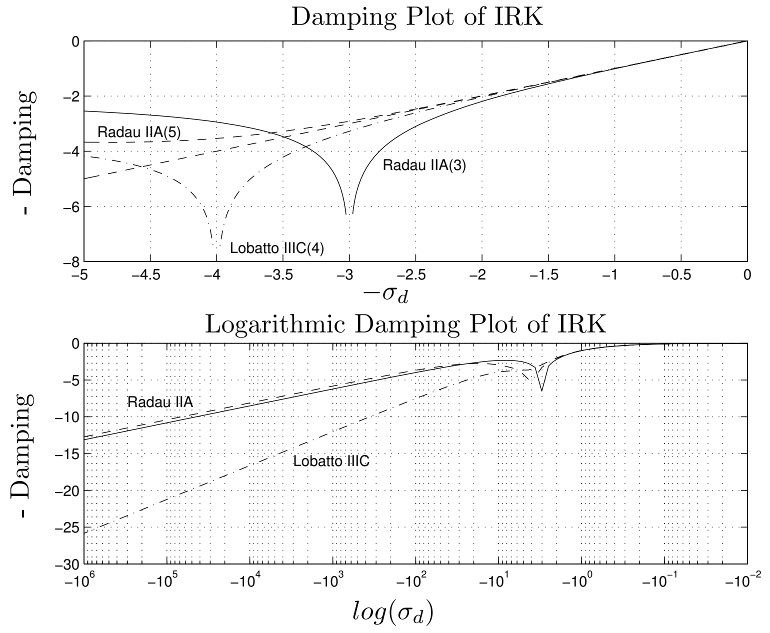

FIGURE 8.6. Damping plots of fully–implicit Runge–Kutta algorithms.

Let us also look at the damping plots for the three methods. These are shown on Fig.8.6.

All three methods exhibit satisfyingly large asymptotic regions, much more so than the BDF algorithms. Although Radau IIA(3) calls for a Newton iteration around two function evaluations, i.e., roughly eight function evaluations per step, and Radau IIA(5) as well as Lobatto IIIC(4) call for a Newton iteration around three function evaluations, adding up to approximately 12 function evaluations per step, all these techniques will allow us to use *much* larger step sizes than in the case of the BDF algorithms due to their large asymptotic regions. This may well balance off the additional cost. Consequently, fully–implicit Runge–Kutta algorithms can indeed be quite competitive in execution speed.

All three methods are obviously L–stable, thus, they are good methods for integrating stiff systems. The Lobatto IIIC technique has somewhat better damping characteristics than the Radau IIA algorithms for poles located far out in the left–half complex plane.

As the explicit RK algorithms are good starter algorithms for the Adams family of methods, and the BI algorithms are good starters for BDF techniques in ODE format, the fully–implicit Runge–Kutta algorithms can be used during startup of a BDF method in DAE format.

8.4 DASSL

DASSL is one of the most successful simulation codes on the market today. It implements the BDF formulae of orders one to five in their DAE format, as presented earlier in this chapter. DASSL is a variable–step, variable–order code that uses order buildup during the startup period. The code was written by Linda Petzold [8.4, 8.34].

The code has meanwhile been made the default simulator in Dymola [8.9, 8.10] in spite of its inefficiency when dealing with non–stiff problems due to the relatively large error coefficients of the BDF formulae, and in spite of its inefficiency when dealing with highly nonlinear problems that require frequent step–size adjustment, such as those that we shall look at in Chapter 9 of this book.

Why did the developers of Dymola choose a stiff–system solver as the default integration algorithm? Dymola was designed to be used in large–scale system modeling. Complex models are almost invariably stiff, as the complexity of the model usually arises from looking simultaneously at phenomena with different time constants. Furthermore, most engineering users don't know whether their models are stiff or not. Since a stiff–system solver is capable of dealing with non–stiff models as well (although not with optimal efficiency), whereas a non–stiff solver cannot deal with stiff models at all, it may be a good idea to use the vacuum cleaner approach, and offer, as the default simulation engine, a code that will be able to cope with most problems somewhat successfully. After all, computers have become fast in recent years, and therefore, optimal efficiency of the simulation engine may no longer be a prime requirement of a modeling and simulation environment.

Why did the developers of Dymola opt for DASSL as the default method rather than e.g. Radau IIA? From what we have learnt, we would expect Radau IIA to be much better suited than DASSL for dealing with highly nonlinear problems requiring frequent step–size adjustment. After all, most engineering models are highly nonlinear.

As we have mentioned earlier, the actual integration algorithm occupies maybe five percent of a production code. The other 95 percent of the code deal with step–size and order control, startup problems, readout problems, and other problems that we haven't looked at yet, such as discontinuity handling (the so–called root solving problem).

The reason is quite simple. There is no production code implementing the Radau IIA algorithms around that is as robust and well tested as the DASSL code. In fact, we haven't even talked yet about such issues as step–size control in implicit Runge–Kutta algorithms.

According to [8.4], DASSL is able to simulate problems of perturbation indices 0 and 1, whereas it may fail when confronted with higher–index problems. Dymola usually reduces the perturbation index of the model to zero, before simulating the model, i.e., although DASSL is capable of solving DAE problems directly, Dymola converts the model to explicit ODE form first, before handing it over to DASSL for simulation.

This decision again sacrifices efficiency for convenience. Multiple Newton iterations may be set up within each other, as Dymola may set up Newton iterations as part of the state–space model, and DASSL employs an overall Newton iteration as part of the simulation process. Yet, solving DAEs directly may be hard on the user, because in the DAE formulation, it is not always evident, how many initial conditions are needed, and where they must be specified. The conversion to explicit ODE form serves the purpose of ensuring that a complete and consistent set of initial conditions is available to properly initialize the simulation run.

Before bringing the discussion of DASSL to an end, let us discuss one more problem that DASSL users may face, a problem that is caused by exploiting the one–legged nature of the BDF formulae in setting up the DAE solver.

Let us look once more at an explicit linear state–space model:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} + \mathbf{B} \cdot \mathbf{u} \tag{8.57}$$

Let us use BDF3 in its ODE form to simulate this system:

$$\mathbf{x_{k+1}} = \frac{6}{11} h \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} + \frac{18}{11}\mathbf{x_k} - \frac{9}{11}\mathbf{x_{k-1}} + \frac{2}{11}\mathbf{x_{k-2}} \tag{8.58}$$

We eliminate the state derivative vector from the Newton iteration by plugging Eq.(8.57) into Eq.(8.58):

$$\mathbf{x_{k+1}^{BDF3}} = \frac{6}{11}\cdot\mathbf{A}\cdot h\cdot\mathbf{x_{k+1}} + \frac{6}{11}\cdot\mathbf{B}\cdot h\cdot\mathbf{u_{k+1}} + \frac{18}{11}\mathbf{x_k} - \frac{9}{11}\mathbf{x_{k-1}} + \frac{2}{11}\mathbf{x_{k-2}} \tag{8.59}$$

We set the zero function for the Newton iteration up as follows:

$$\mathcal{F}(\mathbf{x_{k+1}})^{\mathbf{ODE}} = \mathbf{x_{k+1}^{true}} - \mathbf{x_{k+1}^{BDF3}} \tag{8.60}$$

i.e., we compute the difference between the true yet unknown value of $\mathbf{x_{k+1}^{true}}$ and the approximation of the value using the integration algorithm, $\mathbf{x_{k+1}^{BDF3}}$. Thus, the Hessian can be computed as:

$$\mathcal{H}(\mathbf{x_{k+1}})^{\mathbf{ODE}} = \frac{\partial \mathcal{F}(\mathbf{x_{k+1}})^{\mathbf{ODE}}}{\partial \mathbf{x_{k+1}}} = \mathbf{I^{(n)}} - \frac{6}{11} \cdot \mathbf{A} \cdot h \tag{8.61}$$

or more generally:

$$\mathcal{H}(\mathbf{x_{k+1}})^{\mathbf{ODE}} = \mathbf{I^{(n)}} - \frac{6}{11} \cdot \mathcal{J} \cdot h \tag{8.62}$$

where $\mathcal{J}$ is the Jacobian of the system:

$$\mathcal{J}(\mathbf{x_{k+1}}) = \frac{\partial \mathbf{f}(\mathbf{x_{k+1}})}{\partial \mathbf{x_{k+1}}} \tag{8.63}$$

Let us now analyze the DAE formulation instead. We use the BDF3 formula in its derivative form:

$$\dot{\mathbf{x}}_{\mathbf{k+1}}^{\mathbf{BDF3}} = \frac{1}{h}\left[\frac{11}{6}\cdot\mathbf{x_{k+1}} - 3\mathbf{x_k} + \frac{3}{2}\mathbf{x_{k-1}} - \frac{1}{3}\mathbf{x_{k-2}}\right] \tag{8.64}$$

We set the zero function up as follows:

$$\mathcal{F}(\mathbf{x_{k+1}})^{\mathbf{DAE}} = \dot{\mathbf{x}}_{\mathbf{k+1}}^{\mathbf{st\_eq}} - \dot{\mathbf{x}}_{\mathbf{k+1}}^{\mathbf{BDF3}} \tag{8.65}$$

Thus, we subtract the BDF approximation of the state derivative vector, $\dot{\mathbf{x}}_{\mathbf{k+1}}^{\mathbf{BDF3}}$ from the state derivative vector computed from the state equations, $\dot{\mathbf{x}}_{\mathbf{k+1}}^{\mathbf{st\_eq}}$, i.e., from Eq.(8.57). Both of these approximations are functions of $\mathbf{x_{k+1}}$.

Hence the Hessian can now be computed as:

$$\mathcal{H}(\mathbf{x_{k+1}})^{\mathbf{DAE}} = \frac{\partial\mathcal{F}(\mathbf{x_{k+1}})^{\mathbf{DAE}}}{\partial\mathbf{x_{k+1}}} = \mathbf{A} - \frac{11}{6h}\cdot\mathbf{I^{(n)}} \tag{8.66}$$

or more generally:

$$\mathcal{H}(\mathbf{x_{k+1}})^{\mathbf{DAE}} = \mathbf{J} - \frac{11}{6h}\cdot\mathbf{I^{(n)}} \tag{8.67}$$

What happens when the step size, h, is made very small? In the ODE case, we find:

$$\lim_{h\to 0}\mathcal{H}(\mathbf{x_{k+1}})^{\mathbf{ODE}} = \mathbf{I^{(n)}} \tag{8.68}$$

Thus, the Hessian approaches the identity matrix as the step size approaches zero. In the DAE case, we find:

$$\lim_{h\to 0}\mathcal{H}(\mathbf{x_{k+1}})^{\mathbf{DAE}} \to \infty \tag{8.69}$$

As the step size approaches zero, the Hessian approaches infinity. For small step sizes, the Hessian is highly sensitive to a change in the step size. This forebodes trouble.

Although the ODE and DAE formulations of the BDF formulae are the same algorithms in theory, they may behave quite differently from a numerical point of view for small step sizes due to roundoff.

Let us now look at the most general case of a nonlinear implicit model of the type:

$$\mathcal{F}(\mathbf{x}, \dot{\mathbf{x}}, \mathbf{u}, t) = 0 \tag{8.70}$$

In accordance with Chapter 4 of this book, the different BDF algorithms can be written as:

$$\mathbf{x_{k+1}} = h \cdot \mathbf{f_{k+1}} + \mathbf{x_k} \tag{8.71a}$$

$$\mathbf{x_{k+1}} = \frac{2}{3} \cdot h \cdot \mathbf{f_{k+1}} + \frac{4}{3} \cdot \mathbf{x_k} - \frac{1}{3} \cdot \mathbf{x_{k-1}} \tag{8.71b}$$

$$\mathbf{x_{k+1}} = \frac{6}{11} \cdot h \cdot \mathbf{f_{k+1}} + \frac{18}{11} \cdot \mathbf{x_k} - \frac{9}{11} \cdot \mathbf{x_{k-1}} + \frac{2}{11} \cdot \mathbf{x_{k-2}} \tag{8.71c}$$

$$\text{etc.} \tag{8.71d}$$

Thus in general, we can write all of these equations in the form:

$$\mathbf{x_{k+1}} = \bar{h} \cdot \mathbf{f_{k+1}} + \text{old}(\mathbf{x}) \tag{8.72}$$

where $\bar{h}$ is proportional in the step size h, and old($\mathbf{x}$) is a function of previous values of the state vector, which won't influence the Newton iteration at this time.

We can turn Eq.(8.72) around:

$$\mathbf{f_{k+1}} = \frac{\mathbf{x_{k+1}} - \text{old}(\mathbf{x})}{\bar{h}} \tag{8.73}$$

When DASSL is applied to the model of Eq.(8.70), it plugs Eq.(8.73) into Eq.(8.70):

$$\mathcal{F}(\mathbf{x_{k+1}}, \frac{\mathbf{x_{k+1}} - \text{old}(\mathbf{x})}{\bar{h}}, \mathbf{u_{k+1}}, t_{k+1}) = 0 \tag{8.74}$$

at time t_{k+1}, and iterates on $\mathbf{x_{k+1}}$. In setting up the Newton iteration, we don't actually need to perform the substitution, as we can see from Eq.(8.74) what contributions the state derivative vector produces in the computation of the Hessian:

$$\mathcal{H}(\mathbf{x_{k+1}}) = \mathcal{J}_{\mathbf{x}}(\mathbf{x_{k+1}}) + \frac{1}{\bar{h}} \cdot \mathcal{J}_{\dot{\mathbf{x}}}(\mathbf{x_{k+1}}) \tag{8.75}$$

where:

$$\mathcal{J}_{\mathbf{x}}(\mathbf{x_{k+1}}) = \left.\frac{\partial \mathcal{F}}{\partial \mathbf{x}}\right|_{x=x_{k+1}, \dot{x}=\dot{x}_{k+1}} \tag{8.76a}$$

$$\mathcal{J}_{\dot{\mathbf{x}}}(\mathbf{x_{k+1}}) = \left.\frac{\partial \mathcal{F}}{\partial \dot{\mathbf{x}}}\right|_{x=x_{k+1}, \dot{x}=\dot{x}_{k+1}} \tag{8.76b}$$

are partial Jacobians without substitution.

Hence we can set up the Newton iteration in the following way:

$$\left(\mathcal{J}_{\mathbf{x}} + \frac{1}{\bar{h}} \cdot \mathcal{J}_{\dot{\mathbf{x}}}\right) \cdot \delta^{\ell} = \mathcal{F}(\mathbf{x}^{\ell}, \dot{\mathbf{x}}^{\ell}, t) \tag{8.77a}$$

$$\mathbf{x}^{\ell+1} = \mathbf{x}^{\ell} - \delta^{\ell} \tag{8.77b}$$

$$\dot{\mathbf{x}}^{\ell+1} = \dot{\mathbf{x}}^{\ell} - \frac{1}{\bar{h}} \cdot \delta^{\ell} \tag{8.77c}$$

By multiplying Eq.(8.77a) by the step size h, we can write the linear system inside the Newton iteration as:

$$(\bar{h} \cdot \mathcal{J}_{\mathbf{x}} + \mathcal{J}_{\dot{\mathbf{x}}}) \cdot \delta^{\ell} = \bar{h} \cdot \mathcal{F}(\mathbf{x}^{\ell}, \dot{\mathbf{x}}^{\ell}, t) \tag{8.78}$$

Which variables need to be included in the iteration vector, $\mathbf{x}$, of the Newton iteration? If the problem to be solved is an index–0 problem, the iteration vector is identical to the vector of independent state variables. However, if the problem to be solved is an index–1 problem, then at least the tearing variables of the algebraic loops need to be included in the iteration vector, $\mathbf{x}$, as well.

What happens if we let the normalized step size, $\bar{h}$, go to zero? The Hessian then degenerates to:

$$\lim_{\bar{h} \rightarrow 0} \mathcal{H} = \mathcal{J}_{\dot{\mathbf{x}}} \tag{8.79}$$

which, in the case of an index–1 problem, unfortunately is a singular matrix. Thus, the smaller the step size, the more poorly conditioned the Newton iteration will become in the simulation of an index–1 problem.

Unfortunately, small step sizes will haunt us throughout Chapters 9 and 10 of this book, which is yet another reason, why the producers of Dymola chose to symbolically convert all DAEs to explicit ODE form prior to letting DASSL handle the simulation.

8.5 Inline Integration

You, the reader, may meanwhile have come to the conclusion that direct simulation of an index–1 DAE problem is a bad idea after all. Yet, the problems that we encountered are not directly related to the index–1 DAE problem, but rather to the way, in which DASSL was set up. When Linda Petzold developed the code, she still clung to the idea that the simulation engine must be separated from the model equations, in order to protect the hapless user. In 1983, when DASSL was developed, computers were still slow, memory was still expensive, and consequently, compilers were still limited in their capabilities.

It turns out that direct simulation of a stiff index–1 DAE problem may still be a good idea at times, but before we can attempt such a direct simulation, the final barrier between the simulation engine and the model equations must come down.

For the time being, let us restrict our discussion to the use of backward Euler, i.e., BDF1:

$$\mathbf{x}^{\mathbf{BE}}_{\mathbf{k+1}} = \mathbf{x}_{\mathbf{k}} + h \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{8.80}$$

Let us look once more at the first of our three circuit problems. Its schematic is displayed in Fig.8.7.

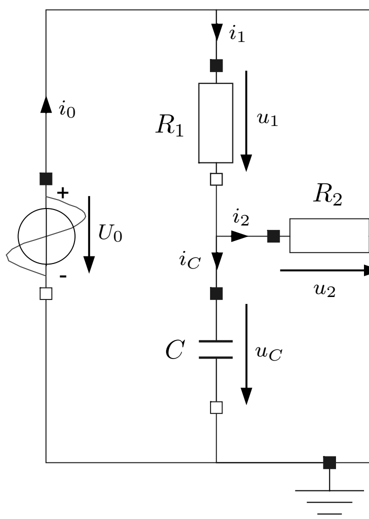

FIGURE 8.7. Schematic of electrical RLC circuit.

However this time around, we shall insert the integration equations directly into the model. The enhanced set of model equations can be written as follows:

$$u_0 = f(t) \tag{8.81a}$$
$$u_1 = R_1 \cdot i_1 \tag{8.81b}$$
$$u_2 = R_2 \cdot i_2 \tag{8.81c}$$
$$u_L = L \cdot di_L \tag{8.81d}$$
$$i_C = C \cdot du_C \tag{8.81e}$$
$$u_0 = u_1 + u_C \tag{8.81f}$$
$$u_L = u_1 + u_2 \tag{8.81g}$$
$$u_C = u_2 \tag{8.81h}$$
$$i_0 = i_1 + i_L \tag{8.81i}$$
$$i_1 = i_2 + i_C \tag{8.81j}$$
$$i_L = \text{pre}(i_L) + h \cdot di_L \tag{8.81k}$$
$$u_C = \text{pre}(u_C) + h \cdot du_C \tag{8.81l}$$

$\text{pre}(i_L)$ denotes the previous value of i_L. At time $t = 0$, we set $\text{pre}(i_L) = i_{L_0}$, i.e., we apply the initial conditions of the state variables to the vector of previous states, and evaluate the model equations for the first time at $t = h$.

By inserting ("inlining") the integrator equations into the model, we eliminated the *differential equations* altogether [8.8]. We are now faced with a set of *difference equations* that we need to solve once per step at times $t = h$, $t = 2h$, etc. di_L and du_C are no longer state derivatives. They have now turned into algebraic variables with funny names.

The structure digraph of the above difference equation (ΔE) model is shown in Fig.8.8.

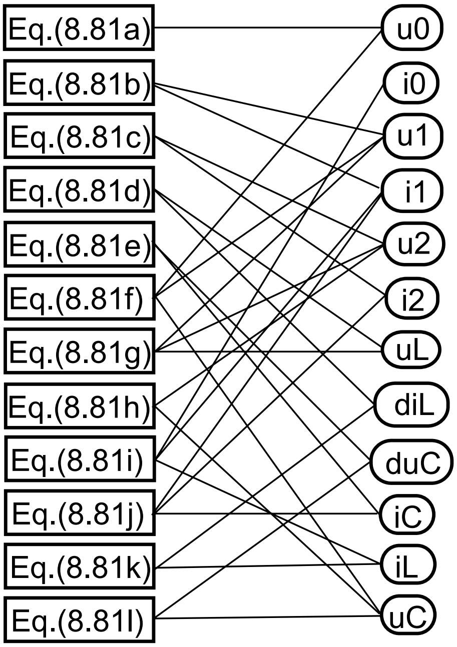

FIGURE 8.8. Structure digraph of electrical circuit.

Let us start to causalize the structure digraph. The results of our efforts are shown in Fig.8.9.

We were able to causalize five of our 12 equations, before encountering an algebraic loop. Whereas the original DAE problem had been an index–0 problem, i.e., a problem not leading to an algebraic loop, the converted ΔE problem contains an algebraic loop, which calls for a Newton iteration. This is the Newton iteration caused by the implicit integration algorithm.

Let us find a suitable tearing structure. We shall not use our usual heuristics. The reason is that we don't want the step size, h, to show up in the denominator of any equation. Thus, we shall use Eq.(8.81l) as our residual equation, which we solve for the tearing variable, u_C. It turns out that, with this choice, we are able to causalize all remaining equations. The results of the causalization are shown in Fig.8.10.

Using DASSL, we would have required two iteration variables, namely the two state variables, i_L and u_C. Using inline integration, we only require a single iteration variable, the tearing variable, u_C.

The fact that we were using backward Euler in the above analysis is

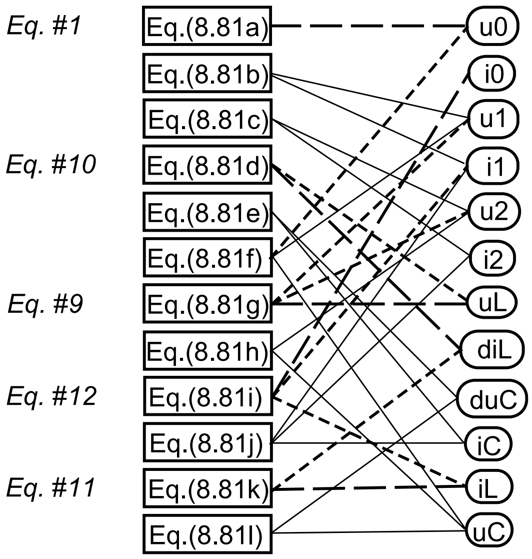

FIGURE 8.9. Partially causalized structure digraph of electrical circuit.

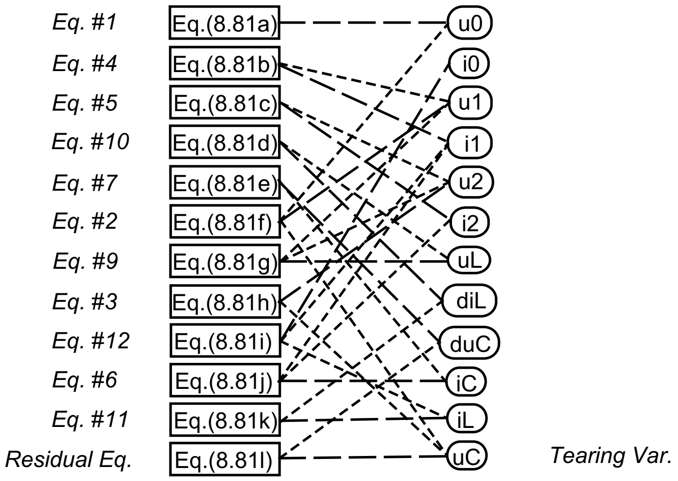

FIGURE 8.10. Completely causalized structure digraph of electrical circuit.

actually irrelevant. We could have used any BDF algorithm, or in fact, we even could have used a variable–step and variable–order BDF technique. All we would have had to do is to replace the step size h by the normalized step size $\bar{h}$, and pre($\mathbf{x}$) by old($\mathbf{x}$). Neither of these two substitutions modifies the structure digraph.

The causal set of ΔEs can be written as follows:

$$u_0 = f(t) \tag{8.82a}$$
$$u_1 = u_0 - u_C \tag{8.82b}$$
$$u_2 = u_C \tag{8.82c}$$
$$i_1 = \frac{1}{R_1} \cdot u_1 \tag{8.82d}$$
$$i_2 = \frac{1}{R_2} \cdot u_2 \tag{8.82e}$$
$$i_C = i_1 - i_2 \tag{8.82f}$$
$$du_C = \frac{1}{C} \cdot i_C \tag{8.82g}$$
$$u_C = \text{pre}(u_C) + h \cdot du_C \tag{8.82h}$$
$$u_L = u_1 + u_2 \tag{8.82i}$$
$$di_L = \frac{1}{L} \cdot u_L \tag{8.82j}$$
$$i_L = \text{pre}(i_L) + h \cdot di_L \tag{8.82k}$$
$$i_0 = i_1 + i_L \tag{8.82l}$$

Let us apply variable substitution to come up with a completely causal set of equations.

$$\begin{aligned}
u_C &= \text{pre}(u_C) + h \cdot du_C \\
&= \text{pre}(u_C) + \frac{h}{C} \cdot i_C \\
&= \text{pre}(u_C) + \frac{h}{C} \cdot i_1 - \frac{h}{C} \cdot i_2 \\
&= \text{pre}(u_C) + \frac{h}{R_1 \cdot C} \cdot u_1 - \frac{h}{R_2 \cdot C} \cdot u_2 \\
&= \text{pre}(u_C) + \frac{h}{R_1 \cdot C} \cdot u_0 - \frac{h}{R_1 \cdot C} \cdot u_C - \frac{h}{R_2 \cdot C} \cdot u_C
\end{aligned}$$

and therefore:

$$\left[1 + \frac{h}{R_1 \cdot C} + \frac{h}{R_2 \cdot C}\right] \cdot u_C = \text{pre}(u_C) + \frac{h}{R_1 \cdot C} \cdot u_0$$

or:

$$[R_1 \cdot R_2 \cdot C + h \cdot (R_1 + R_2)] \cdot u_C = R_1 \cdot R_2 \cdot C \cdot \text{pre}(u_C) + h \cdot R_2 \cdot u_0$$

which can be solved for u_C:

$$u_C = \frac{R_1 \cdot R_2 \cdot C}{R_1 \cdot R_2 \cdot C + h \cdot (R_1 + R_2)} \cdot \text{pre}(u_C) + \frac{h \cdot R_2}{R_1 \cdot R_2 \cdot C + h \cdot (R_1 + R_2)} \cdot u_0 \tag{8.83}$$

If we let the step size go to zero, we find:

$$\lim_{h \to 0} u_C = \text{pre}(u_C) \tag{8.84}$$

which is non–singular. Since the original DAE problem had been of index 0, this is not further surprising.

Let us now look at the second of our circuits. Its schematic is shown in Fig.8.11.

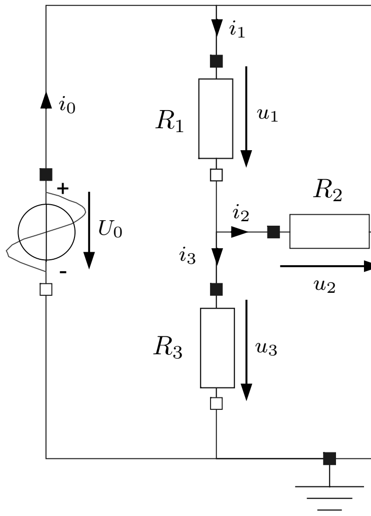

FIGURE 8.11. Schematic of modified electrical RLC circuit.

Remember this circuit represents an index–1 problem. Inlining the single integrator, we get the following set of acausal equations:

$$u_0 = f(t) \tag{8.85a}$$
$$u_1 = R_1 \cdot i_1 \tag{8.85b}$$
$$u_2 = R_2 \cdot i_2 \tag{8.85c}$$
$$u_3 = R_3 \cdot i_3 \tag{8.85d}$$
$$u_L = L \cdot di_L \tag{8.85e}$$
$$u_0 = u_1 + u_3 \tag{8.85f}$$
$$u_L = u_1 + u_2 \tag{8.85g}$$
$$u_3 = u_2 \tag{8.85h}$$

$$i_0 = i_1 + i_L \tag{8.85i}$$
$$i_1 = i_2 + i_3 \tag{8.85j}$$
$$i_L = \text{pre}(i_L) + h \cdot di_L \tag{8.85k}$$

Its structure digraph is shown in Fig.8.12.

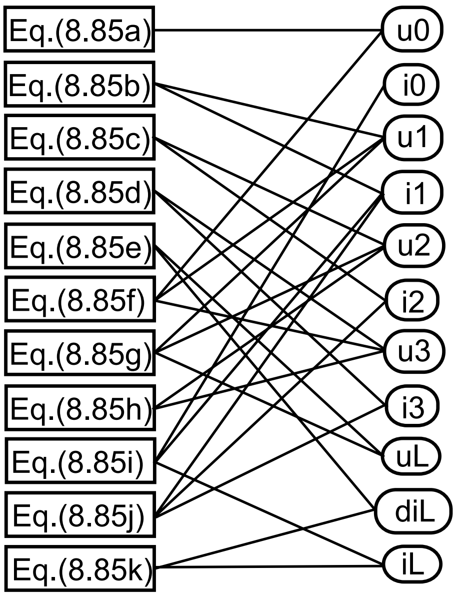

FIGURE 8.12. Structure digraph of modified electrical circuit.

We begin to causalize the structure digraph. The partially causalized structure digraph is shown in Fig.8.13.

We were able to causalize five of the 11 equations. Let us apply our heuristic procedure to select a first residual equation and a first tearing variable. The results of our efforts are shown in Fig.8.14.

A single tearing variable sufficed to causalize the entire equation system. DASSL would have required at least two iteration variables, the state variable, i_L, and the single tearing variable of the algebraic loop, i_3, of the index–1 DAE system. Inline integration is more economical. We get away with a single iteration variable, the tearing variable, i_1, of the ΔE system.

Let us write down the causal equations:

$$u_0 = f(t) \tag{8.86a}$$
$$u_1 = R_1 \cdot i_1 \tag{8.86b}$$
$$u_3 = u_0 - u_1 \tag{8.86c}$$
$$u_2 = u_3 \tag{8.86d}$$

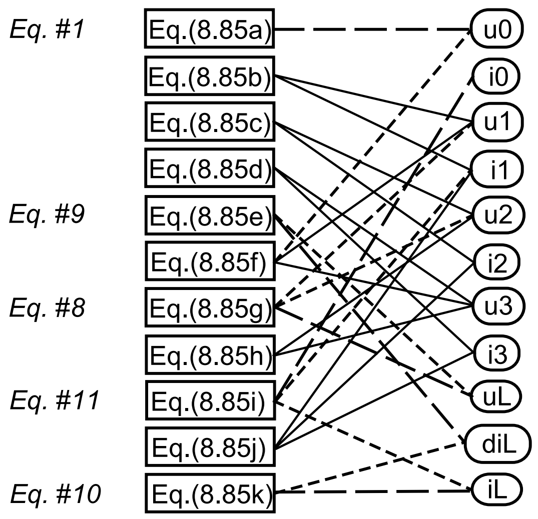

FIGURE 8.13. Partially causalized structure digraph of modified electrical circuit.

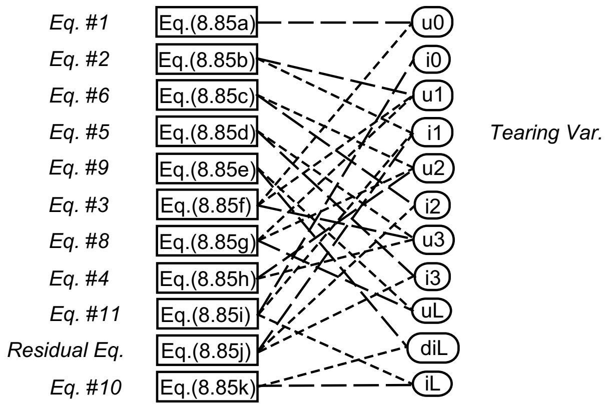

FIGURE 8.14. Completely causalized structure digraph of modified electrical circuit.

$$i_3 = \frac{1}{R_3} \cdot u_3 \tag{8.86e}$$

$$i_2 = \frac{1}{R_2} \cdot u_2 \tag{8.86f}$$

$$i_1 = i_2 + i_3 \tag{8.86g}$$

$$u_L = u_1 + u_2 \tag{8.86h}$$

$$di_L = \frac{1}{L} \cdot u_L \tag{8.86i}$$

$$i_L = \text{pre}(i_L) + h \cdot di_L \tag{8.86j}$$

$$i_0 = i_1 + i_L \tag{8.86k}$$

Using the variable substitution technique, we can find a closed–form expression for the tearing variable:

$$i_1 = \frac{R_2 + R_3}{R_1 \cdot R_2 + R_1 \cdot R_3 + R_2 \cdot R_3} \cdot u_0 \tag{8.87}$$

The expression for i_1 is not even a function of the step size h, i.e., it is non–singular for any value of h.

Let us now analyze the third circuit. Its schematic is shown in Fig.8.15. Remember this is an index–2 problem.

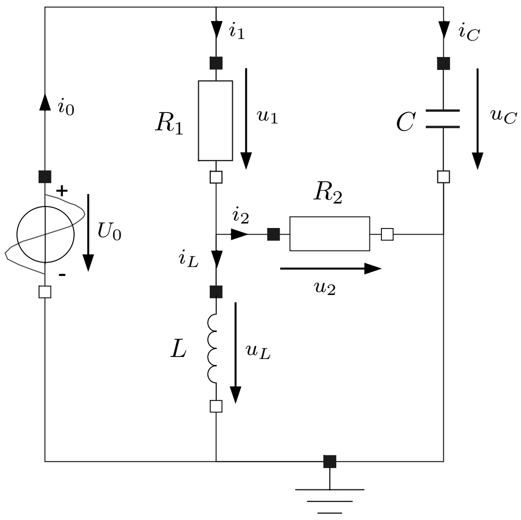

FIGURE 8.15. Schematic of once more modified electrical RLC circuit.

After inlining the integrator equations, the acausal equations present themselves in the following form:

$$u_0 = f(t) \tag{8.88a}$$
$$u_1 = R_1 \cdot i_1 \tag{8.88b}$$
$$u_2 = R_2 \cdot i_2 \tag{8.88c}$$
$$u_L = L \cdot di_L \tag{8.88d}$$
$$i_C = C \cdot du_C \tag{8.88e}$$
$$u_0 = u_1 + u_L \tag{8.88f}$$
$$u_C = u_1 + u_2 \tag{8.88g}$$
$$u_L = u_2 \tag{8.88h}$$
$$i_0 = i_1 + i_C \tag{8.88i}$$

$$i_1 = i_2 + i_L \tag{8.88j}$$
$$i_L = \text{pre}(i_L) + h \cdot di_L \tag{8.88k}$$
$$u_C = \text{pre}(u_C) + h \cdot du_C \tag{8.88l}$$

The structure digraph is shown in Fig.8.16.

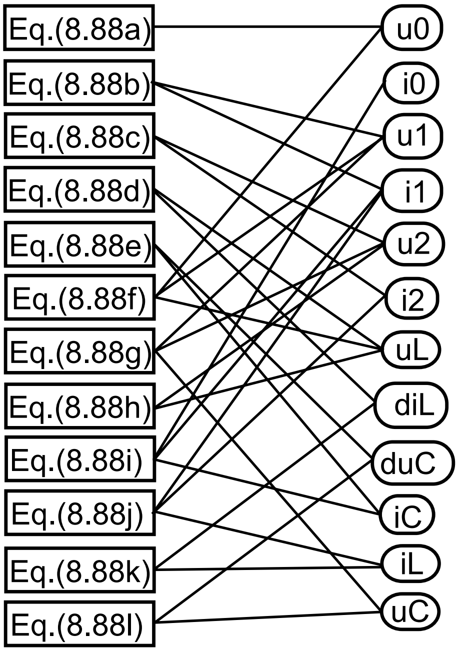

FIGURE 8.16. Structure digraph of once more modified electrical circuit.

We begin to causalize the structure digraph. The results of our efforts are shown in Fig.8.17.

We were able to causalize five of the 12 equations, before ending up with an algebraic loop. Evidently, since the computational causalities of all energy storage elements have been freed up after inlining the integrator equations, we don't obtain any constraint equation any longer.

Unfortunately, we already got ourselves into trouble, because Eq.(8.88l) needs to be solved for du_C:

$$du_C = \frac{u_C - \text{pre}(u_C)}{h} \tag{8.89}$$

i.e., we ended up with the step size, h, in the denominator, which invariably will cause numerical difficulties, when we try to simulate the system using a small step size. We had no choice in the matter, as the derivative causality on the capacitor was dictated upon us.

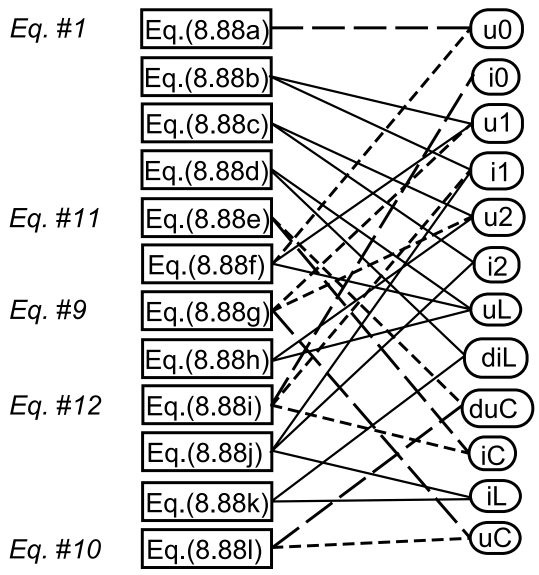

FIGURE 8.17. Partially causalized structure digraph of once more modified electrical circuit.

Let us nevertheless continue by applying our heuristic procedure for selecting a first residual equation and a first tearing variable. The results of our efforts are shown in Fig.8.18.

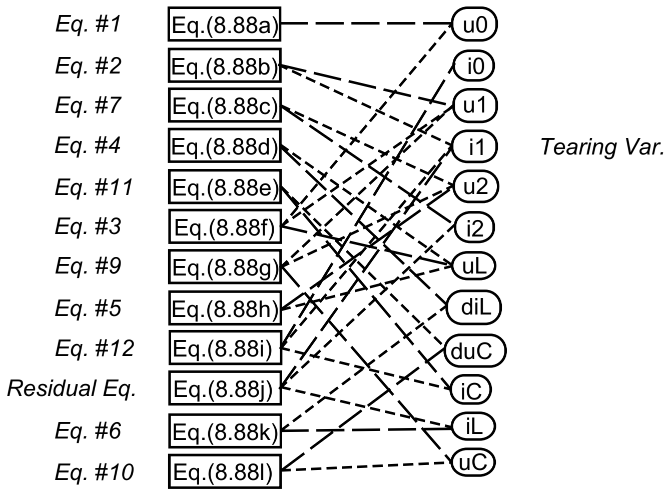

FIGURE 8.18. Completely causalized structure digraph of once more modified electrical circuit.

A single tearing variable suffices to causalize the entire equation system.

The causal equations can be read out of the structure digraph of Fig.8.18.

$$u_0 = f(t) \tag{8.90a}$$
$$u_1 = R_1 \cdot i_1 \tag{8.90b}$$
$$u_L = u_0 - u_1 \tag{8.90c}$$
$$di_L = \frac{1}{L} \cdot u_L \tag{8.90d}$$
$$u_2 = u_L \tag{8.90e}$$
$$i_L = \text{pre}(i_L) + h \cdot di_L \tag{8.90f}$$
$$i_2 = \frac{1}{R_2} \cdot u_2 \tag{8.90g}$$
$$i_1 = i_2 + i_L \tag{8.90h}$$
$$u_C = u_1 + u_2 \tag{8.90i}$$
$$du_C = \frac{u_C - \text{pre}(u_C)}{h} \tag{8.90j}$$
$$i_C = C \cdot du_C \tag{8.90k}$$
$$i_0 = i_1 + i_C \tag{8.90l}$$

Using the variable substitution technique, we can find a closed–form equation for the tearing variable, i_1.

$$i_1 = \frac{L + h \cdot R_2}{L \cdot (R_1 + R_2) + h \cdot R_2} \cdot u_0 + \frac{R_2 \cdot L}{L \cdot (R_1 + R_2) + h \cdot R_2} \cdot \text{pre}(i_L) \tag{8.91}$$

If we let the step size go to zero, we find:

$$\lim_{h \to 0} i_1 = \frac{1}{R_1 + R_2} \cdot u_0 + \frac{R_2}{R_1 + R_2} \cdot \text{pre}(i_L) \tag{8.92}$$

At least in the given example, inlining was able to solve also the higher–index problem directly. This discovery shall prove important in the context of the next chapter of this book. Yet, inlining the higher–index problem directly came at a price, as we ended up with the step size, h, in the denominator of one of the model equations. Thus, it is usually preferred to first apply the index reduction algorithm by Pantelides.

We have shown that inline integration can solve DAE problems directly and more economically than the standard version of DASSL[1] In all of these examples, we have used the backward Euler formula for inlining.

[1]The standard version of DASSL comes with a regular matrix solver and a band–matrix solver. In addition, DASSL offers an interface for supplying other matrix solvers externally. A sparse matrix solver can improve the efficiency of DASSL significantly fo large numbers of states [8.31].

However, this is not necessary. If we replace the true step size, h, by the normalized step size $\bar{h}$, and the previous value of the state vector, pre($\mathbf{x}$), by a combination of old state information, old($\mathbf{x}$), we can inline any and all of the BDF algorithms in exactly the same fashion.

If we wish to implement a step–size and/or order controlled algorithm, we can do so using the same techniques that were advocated in Chapter 4 of this book. Since both the new step size and the new order depend on previous state information only, the equations for step–size and order control do not need to be inlined. Only the integration formula itself must be inlined, which can be accomplished for all BDF algorithms in the manner demonstrated in this section.

8.6 Inlining Implicit Runge–Kutta Algorithms

How can the inlining technique be generalized to implicit Runge–Kutta algorithms as well? For each stage of the multi–stage algorithm, we need to replicate the entire set of equations once. Let us explain the technique by means of the first of the three circuit examples. We shall inline the third–order accurate Radau IIA algorithm. Since this is a two–stage algorithm, we need to write down the equations twice, once for each of the two stages, for the time instant, when that stage needs to be computed.

The first stage is computed at time $t = t_k + h/3$, whereas the second stage is computed at time $t = t_k + h = t_{k+1}$. The integrator formulae can thus be written as:

$$\mathbf{x_{k+\frac{1}{3}}} = \mathbf{x_k} + \frac{5h}{12} \cdot \dot{\mathbf{x}}_{\mathbf{k+\frac{1}{3}}} - \frac{h}{12} \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{8.93a}$$

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{3h}{4} \cdot \dot{\mathbf{x}}_{\mathbf{k+\frac{1}{3}}} + \frac{h}{4} \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{8.93b}$$

Hence the complete set of equations for the circuit example can be written as:

$$v_0 = f(t + \frac{h}{3}) \tag{8.94a}$$

$$v_1 = R_1 \cdot j_1 \tag{8.94b}$$

$$v_2 = R_2 \cdot j_2 \tag{8.94c}$$

$$v_L = L \cdot dj_L \tag{8.94d}$$

$$j_C = C \cdot dv_C \tag{8.94e}$$

$$v_0 = v_1 + v_C \tag{8.94f}$$

$$v_L = v_1 + v_2 \tag{8.94g}$$

$$v_C = v_2 \tag{8.94h}$$

$$j_0 = j_1 + j_L \tag{8.94i}$$
$$j_1 = j_2 + j_C \tag{8.94j}$$

$$u_0 = f(t+h) \tag{8.94k}$$
$$u_1 = R_1 \cdot i_1 \tag{8.94l}$$
$$u_2 = R_2 \cdot i_2 \tag{8.94m}$$
$$u_L = L \cdot di_L \tag{8.94n}$$
$$i_C = C \cdot du_C \tag{8.94o}$$
$$u_0 = u_1 + u_C \tag{8.94p}$$
$$u_L = u_1 + u_2 \tag{8.94q}$$
$$u_C = u_2 \tag{8.94r}$$
$$i_0 = i_1 + i_L \tag{8.94s}$$
$$i_1 = i_2 + i_C \tag{8.94t}$$

$$j_L = \text{pre}(i_L) + \frac{5h}{12} \cdot dj_L - \frac{h}{12} \cdot di_L \tag{8.94u}$$
$$v_C = \text{pre}(u_C) + \frac{5h}{12} \cdot dv_C - \frac{h}{12} \cdot du_C \tag{8.94v}$$
$$i_L = \text{pre}(i_L) + \frac{3h}{4} \cdot dj_L + \frac{h}{4} \cdot di_L \tag{8.94w}$$
$$u_C = \text{pre}(u_C) + \frac{3h}{4} \cdot dv_C + \frac{h}{4} \cdot du_C \tag{8.94x}$$

Thus, we end up with a difference equation (ΔE) system in 24 equations and 24 unknowns. Since the two stages are implicitly coupled to each other, they cannot be executed in sequence. They are simulated together leading to a model containing twice as many equations and unknowns [8.6, 8.38].

We are only interested in the variables of the second stage. At the end of the step, i_L and u_C need to be copied to the previous state vector, pre(i_L) and pre(u_C). Yet, we must compute the variables of the first stage simultaneously with those of the second stage due to the coupling between the two stages.

We shall refrain from drawing the structure digraph for this ΔE system. Let us summarize the results. 10 of the 24 equations can be causalized at once. The heuristic procedure chooses v_C as the first tearing variable, and Eq.(8.94v) as the first residual equation. With this choice, seven additional equations can be causalized. The procedure then chooses i_1 as the second tearing variable, and Eq.(8.94t) as the second residual equation. With this choice, the remaining seven equations can be causalized.

Hence we can simulate this problem using the third–order accurate Radau IIA algorithm with only two iteration variables in the Newton iteration.

8.7 Stiffly Stable Step–size Control of Radau IIA

A difficult problem with these types on numerical solvers concerns the control of the step size. To this end, it is necessary to find an estimate for the integration error, the order of approximation accuracy of which is one order higher than that of the solver itself. Typically, designers of such solvers will look for a second solver of the same or higher order of approximation accuracy to compare it against the solver to be used for the simulation.

While it is always possible to run two independent solvers in parallel for the purpose of step–size control, this approach is clearly undesirable, as it makes the solver highly inefficient. In explicit Runge–Kutta algorithms, it has become customary to search for an *embedding method*, i.e., a second solver that has most of the computations in common with the original solver, such that they share a large portion of the computational load between them. Unfortunately, this approach won't work in the case of fully implicit Runge–Kutta algorithms, since these algorithms are so compact and so highly optimized that there simply is not enough freedom left in these algorithms for embedding methods to co–exist with them.

One solution that comes to mind immediately is to use a Backward Difference Formula in parallel with the implicit Runge–Kutta technique. This solution can be implemented cheaply, because an appropriately accurate state derivative at time t_{k+1} can be obtained up front using the Runge–Kutta approximation, i.e., no Newton iteration is necessary. For example, the 3^{rd}–order accurate Radau IIA algorithm could be accompanied by a 3^{rd}–order accurate BDF solver implemented as:

$$\mathbf{x_{k+1}^{BDF}} = \frac{18}{11}\mathbf{x_k} - \frac{9}{11}\mathbf{x_{k-1}} + \frac{2}{11}\mathbf{x_{k-2}} + \frac{6}{11} \cdot h \cdot \mathbf{f_{k+1}} \tag{8.95}$$

where $\mathbf{f_{k+1}}$ is the function value evaluated from the state–space model at time t_{k+1}:

$$\mathbf{f_{k+1}} = \mathbf{f}(\mathbf{x_{k+1}^{IRK}}, \mathbf{u_{k+1}}, t_{k+1}) \tag{8.96}$$

and $\mathbf{x_{k+1}^{IRK}}$ is the solution found by the Radau IIA algorithm. Unfortunately, such a solution inherits all the difficulties associated with step–size control in linear multi–step methods. Alternatively, the step size can only be modified once every n steps, where n is the order of the algorithm, which eliminates an important aspect of the elegance and efficiency of Runge–Kutta methods. For these reasons, we propose a different route.

Clearly, an embedding method cannot be found using only information that is being used by the Radau IIA algorithm. In each step, there are four pieces of information available: $\mathbf{x_k}$, $\mathbf{x_{k+\frac{1}{3}}}$, $\dot{\mathbf{x}}_{\mathbf{k+\frac{1}{3}}}$, and $\dot{\mathbf{x}}_{\mathbf{k+1}}$ to estimate $\mathbf{x_{k+1}}$. Evidently, there is only one 3^{rd}–order accurate polynomial going through these four pieces of information, and it is this polynomial that defines the Radau IIA algorithm. However, enough redundancy can be obtained to define an embedding algorithm if information from the two last steps is being used. In this case, the following eight pieces of information are available: $\mathbf{x_{k-1}}$, $\mathbf{x_{k-\frac{2}{3}}}$, $\mathbf{x_k}$, $\mathbf{x_{k+\frac{1}{3}}}$, $\mathbf{\dot{x}_{k-\frac{2}{3}}}$, $\mathbf{\dot{x}_k}$, $\mathbf{\dot{x}_{k+\frac{1}{3}}}$, and $\mathbf{\dot{x}_{k+1}}$. It was decided to look for 4^{th}–order accurate polynomials that go through any five of these eight pieces of information. This technique defines 56 possible embedding methods. Out of these 56 methods, only six are stiffly stable. Two of those six techniques are not A–stable, i.e., have unstable regions within the left half complex $\lambda \cdot h$ plane. One method has a stability domain with a discontinuous derivative at the real axis, which is suspicious. The remaining three methods are:

$$\begin{aligned}
\mathbf{x^1_{k+1}} &= -\frac{25}{279}\,\mathbf{x_{k-1}} + \frac{6}{31}\,\mathbf{x_{k-\frac{2}{3}}} + \frac{250}{279}\,\mathbf{x_k} + \frac{25h}{31}\,\mathbf{\dot{x}_{k+\frac{1}{3}}} + \frac{65h}{279}\,\mathbf{\dot{x}_{k+1}} \\
\mathbf{x^2_{k+1}} &= -\frac{1}{36}\,\mathbf{x_{k-1}} + \frac{16}{9}\,\mathbf{x_k} - \frac{3}{4}\,\mathbf{x_{k+\frac{1}{3}}} + h\,\mathbf{\dot{x}_{k+\frac{1}{3}}} + \frac{2h}{9}\,\mathbf{\dot{x}_{k+1}} \qquad (8.97) \\
\mathbf{x^3_{k+1}} &= -\frac{2}{23}\,\mathbf{x_{k-\frac{2}{3}}} + \frac{50}{23}\,\mathbf{x_k} - \frac{25}{23}\,\mathbf{x_{k+\frac{1}{3}}} + \frac{25h}{23}\,\mathbf{\dot{x}_{k+\frac{1}{3}}} + \frac{5h}{23}\,\mathbf{\dot{x}_{k+1}}
\end{aligned}$$

All of these three techniques have nice stability domains looping in the right half complex $\lambda \cdot h$ plane. Each of them is A–stable. It is possible to write these methods in the linear case as:

$$\mathbf{x}_{k+1} = \mathbf{F} \cdot \mathbf{x}_{k-1} \qquad (8.98)$$

The $\mathbf{F}$–matrices of the three methods can be expanded into Taylor series around $h = 0$. The three $\mathbf{F}$–matrices then take the form:

$$\mathbf{F}^1 \approx \mathbf{I^{(n)}} + 2\;\mathbf{A}h + 4\;\frac{(\mathbf{A}h)^2}{2} + \frac{2224}{279}\;\frac{(\mathbf{A}h)^3}{6} + \frac{877}{58}\;\frac{(\mathbf{A}h)^4}{24} \qquad (8.99a)$$

$$\mathbf{F}^2 \approx \mathbf{I^{(n)}} + 2\;\mathbf{A}h + 4\;\frac{(\mathbf{A}h)^2}{2} + \frac{73}{9}\;\frac{(\mathbf{A}h)^3}{6} + \frac{859}{54}\;\frac{(\mathbf{A}h)^4}{24} \qquad (8.99b)$$

$$\mathbf{F}^3 \approx \mathbf{I^{(n)}} + 2\;\mathbf{A}h + 4\;\frac{(\mathbf{A}h)^2}{2} + \frac{188}{23}\;\frac{(\mathbf{A}h)^3}{6} + \frac{374}{23}\;\frac{(\mathbf{A}h)^4}{24} \qquad (8.99c)$$

What would have been expected of a 4th–order accurate method is:

$$\mathbf{F} \approx \mathbf{I^{(n)}} + 2\;\mathbf{A}h + 4\;\frac{(\mathbf{A}h)^2}{2} + 8\;\frac{(\mathbf{A}h)^3}{6} + 16\;\frac{(\mathbf{A}h)^4}{24} \qquad (8.100)$$

since the expansion is over a double step. Unfortunately, neither of these three methods is even 3^{rd}–order accurate. The problem is that although $\mathbf{x_{k+1}}$ is 3^{rd}–order accurate, the first stage of the method, $\mathbf{x_{k+\frac{1}{3}}}$ is only 2^{nd}–order accurate. We evidently cannot expect the order of approximation accuracy of our 4^{th}–order polynomials to be any higher than that of its supporting values, and indeed, all three of our 4th–order polynomials are only 2nd–order accurate.

Luckily, there are three such methods available. Hence it should be possible to blend them:

$$\mathbf{x_{k+1}^{blended}} = \alpha\ \mathbf{x_{k+1}^1} + \beta\ \mathbf{x_{k+1}^2} + (1-\alpha-\beta)\ \mathbf{x_{k+1}^3} \tag{8.101}$$

such that the coefficients of the Taylor–series expansion of the blended method are correct up to the quartic term. Unfortunately, this doesn't work, because the three methods are not linearly independent of each other. There really are only two methods. The third one is a linear combination of the other two. However, it is possible to blend any two of these three methods with the solution found by Radau IIA:

$$\mathbf{x_{k+1}^{blended}} = \alpha \cdot \mathbf{x_{k+1}^1} + \beta \cdot \mathbf{x_{k+1}^2} + (1-\alpha-\beta) \cdot \mathbf{x_{k+1}^{Radau}} \tag{8.102}$$

These three techniques are indeed independent of each other. The resulting algorithm is:

$$\mathbf{x_{k+1}^{blended}} = \mathbf{x_{k-1}} - 2\ \mathbf{x_{k-\frac{2}{3}}} + 2\ \mathbf{x_{k+\frac{1}{3}}} - \frac{h}{2}\ \dot{\mathbf{x}}_{\mathbf{k+\frac{1}{3}}} + \frac{h}{2}\ \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{8.103}$$

This method is indeed 4th–order accurate. It has highly appealing coefficients. It has only one disadvantage. It is totally unstable everywhere.

It should be possible to find 4th–order accurate embedding methods spanned by the information collected from Radau IIA over two steps. Yet, for the purpose of step–size control, it is sufficient to find another 3rd–order accurate embedding method. To this end, it suffices to blend any two of the three algorithms found above:

$$\mathbf{x_{k+1}^{blended}} = \vartheta \cdot \mathbf{x_{k+1}^1} + (1-\vartheta) \cdot \mathbf{x_{k+1}^2} \tag{8.104}$$

The resulting method is:

$$\mathbf{x_{k+1}^{blended}} = -\frac{1}{13}\ \mathbf{x_{k-1}} + \frac{2}{13}\ \mathbf{x_{k-\frac{2}{3}}} + \frac{14}{13}\ \mathbf{x_k} - \frac{2}{13}\ \mathbf{x_{k+\frac{1}{3}}} + \frac{11h}{13}\ \dot{\mathbf{x}}_{\mathbf{k+\frac{1}{3}}} + \frac{3h}{13}\ \dot{\mathbf{x}}_{\mathbf{k+1}} \tag{8.105}$$

Also this method has beautifully simple rational coefficients. it is indeed 3rd–order accurate:

$$\mathbf{F} \approx \mathbf{I^{(n)}} + 2\ \mathbf{A}h + 4\ \frac{(\mathbf{A}h)^2}{2} + 8\ \frac{(\mathbf{A}h)^3}{6} + \frac{149}{156} \cdot 16\ \frac{(\mathbf{A}h)^4}{24} \tag{8.106}$$

i.e., the error coefficient of the method is:

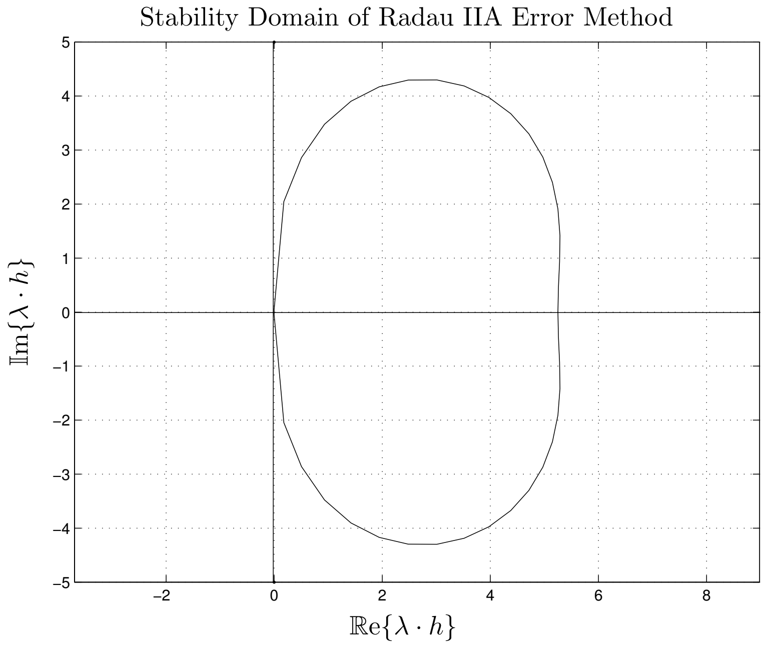

FIGURE 8.19. Stability domain of blended Radau IIA embedding method

$$\varepsilon = \frac{-7}{3744}(\mathbf{A}h)^4 \tag{8.107}$$

The stability domain of the blended method is given in Figure 8.19.

The blended method relies on h not changing its values between the two steps used in the approximation. It may be easiest to prevent the step size from changing two steps in a row. This seems a small price to pay. After the step size has remained constant for two consecutive steps, it is free to change in any way suitable. The code needed to perform step–size control can be merged with the model equations and the simulation equations, i.e., it can be inlined as well, but this is not truly necessary. The step–size control code can be kept in a separate routine called upon by the simulation engine whenever needed.

Which of the two approximations should be propagated to the next step? The error coefficient of the embedding method is considerably smaller than that of Radau IIA. Hence on a first glance, it seems reasonable to propagate the approximation of the embedding technique. However, there are two problems with this choice.

First, the embedding technique was designed assuming that the Radau IIA result would be propagated. If the embedding technique is being propagated, the **F**–matrices change, and the blended method may no longer be 3rd–order accurate.

Second, Figure 8.20 shows the damping plot of the embedding method. Comparing it with the damping plot of Radau IIA, it can be seen that the embedding method is not L–stable, i.e., the damping does not approach infinity as the eigenvalues of the model move further and further to the left.

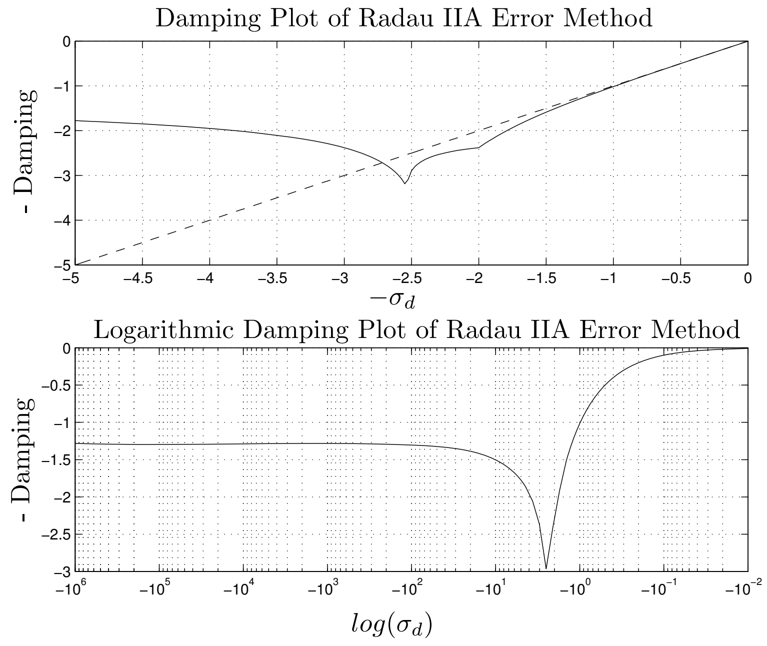

FIGURE 8.20. Damping plot of the blended Radau IIA embedding method

Thus, in spite of the smaller error coefficient, the embedding method should only be used for step–size control, not for propagation.

The fifth–order accurate Radau IIA method (Rad5) can be analyzed analogously. A single step of Rad5 stores six pieces of information: $\mathbf{x_k}$, $\mathbf{x1_k}$, $\mathbf{x2_k}$, $\dot{\mathbf{x}}\mathbf{1_k}$, $\dot{\mathbf{x}}\mathbf{2_k}$, and $\dot{\mathbf{x}}_{\mathbf{k+1}}$, where $\mathbf{x1_k}$ and $\mathbf{x2_k}$ are the approximations of the two intermediate stages. There is only one 5^{th}–order accurate polynomial going through these six pieces of information, and it is this polynomial that defines the Rad5 algorithm. Again, enough redundancy can be obtained to define an embedding algorithm if information from the two last steps is being used. In this case, the following 12 pieces of information are available: $\mathbf{x_{k-1}}$, $\mathbf{x1_{k-1}}$, $\mathbf{x2_{k-1}}$, $\mathbf{x_k}$, $\mathbf{x1_k}$, $\mathbf{x2_k}$, $\dot{\mathbf{x}}\mathbf{1_{k-1}}$, $\dot{\mathbf{x}}\mathbf{2_{k-1}}$, $\dot{\mathbf{x}}_\mathbf{k}$, $\dot{\mathbf{x}}\mathbf{1_k}$, $\dot{\mathbf{x}}\mathbf{2_k}$, and $\dot{\mathbf{x}}_{\mathbf{k+1}}$.

Searching for 6^{th}–order polynomials going through seven of these twelve supporting values, there are 792 methods to be evaluated. Of those, 26 are A–stable methods that can be blended to form an alternate 5^{th}–order accurate embedding method.

Although Rad5 as a whole is 5^{th}–order accurate, its first two stages are only 3rd–order accurate. Thus, we should not expect any of these 6th–order polynomials to reach a higher order of approximation accuracy than three, and indeed, this is what we get. Hence we need to blend at least three of the methods to obtain a 5th–order accurate embedding method.

There exist 2600 combinations of blended methods from the 26 individual methods. We need to eliminate those among them that are not A–stable. We furthermore should choose a method with a small error coefficient and decent damping characteristics. It would be an additional benefit if we could come up with a method that has conveniently small rational coefficients.

A very good embedding method is the following:

$$\begin{aligned}\mathbf{x}_{\mathbf{k+1}}^{\mathbf{blended}} &= c_1 \cdot \mathbf{x}_{\mathbf{k-1}} + c_2 \cdot \dot{\mathbf{x}}_{\mathbf{1}_{\mathbf{k-1}}} + c_3 \cdot \mathbf{x}_{\mathbf{2}_{\mathbf{k-1}}} + c_4 \cdot \dot{\mathbf{x}}_{\mathbf{2}_{\mathbf{k-1}}} + c_5 \cdot \mathbf{x}_{\mathbf{k}} \\ &+ c_6 \cdot \mathbf{x}_{\mathbf{1}_{\mathbf{k}}} + c_7 \cdot \dot{\mathbf{x}}_{\mathbf{1}_{\mathbf{k}}} + c_8 \cdot \mathbf{x}_{\mathbf{2}_{\mathbf{k}}} + c_9 \cdot \dot{\mathbf{x}}_{\mathbf{2}_{\mathbf{k}}} + c_{10} \cdot \dot{\mathbf{x}}_{\mathbf{k+1}} \end{aligned} \tag{8.108a}$$

with the coefficients:

$$c_1 = -0.00517140382204 \tag{8.109a}$$
$$c_2 = -0.00094714677404 \tag{8.109b}$$
$$c_3 = -0.04060469717694 \tag{8.109c}$$
$$c_4 = -0.01364429384901 \tag{8.109d}$$
$$c_5 = +1.41786808325433 \tag{8.109e}$$
$$c_6 = -0.17475783086782 \tag{8.109f}$$
$$c_7 = +0.48299282769491 \tag{8.109g}$$
$$c_8 = -0.19733415138754 \tag{8.109h}$$
$$c_9 = +0.55942205973218 \tag{8.109i}$$
$$c_{10} = +0.10695524944855 \tag{8.109j}$$

We did program the computation of the coefficients also using MATLAB's symbolic toolbox, but the resulting expressions are quite awful, thus we decided to offer the numerical versions instead.

The blending method is indeed 5th–order accurate. It exhibits a nice convex A–stable stability domain, which is shown in Fig.8.21.

The damping plot exhibits a nice large asymptotic region and decent damping characteristics far out in the left–half complex $\lambda \cdot h$–plane. The method is not L–stable, but that is neither surprising nor truly necessary. The damping plot is presented in Fig.8.22.

8.8 Stiffly Stable Step–size Control of Lobatto IIIC

Let us now look at the Lobatto IIIC algorithm. Since the algorithm is less compact than the Radau IIA algorithms, it should be easier to find suitable

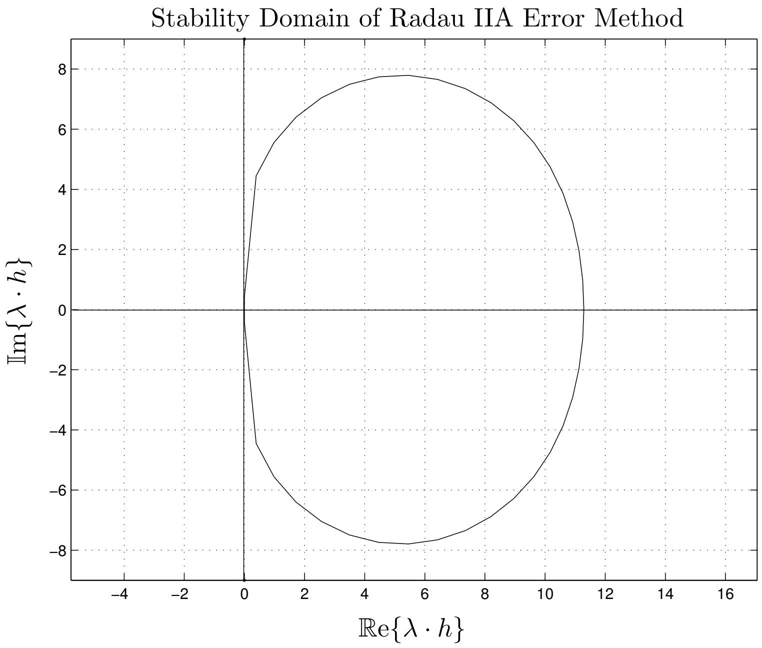

FIGURE 8.21. Stability domain of blended 5th–order Radau IIA embedding method

embedding methods. Yet, each algorithm is accompanied by its own set of difficulties.

First, we checked the order of approximation accuracy of the intermediate stages of the Lobatto IIIC algorithm. Unfortunately, they are only 2nd–order accurate. Hence we shall still need to blend three methods to raise the order of approximation accuracy of the embedding algorithm to four.

Secondly, although we again are working with 12 pieces of information across two steps, Lobatto IIIC has a peculiarity. It experiences a zero time advance between the third stage of one step and the first stage of the next. Although $\mathbf{x_k}$ and $\mathbf{x_{1_k}}$ represent the state vector at the same time instant, they are two different approximations. In particular, $\mathbf{x_k}$ is 4th–order accurate, whereas $\mathbf{x_{1_k}}$ is only 2nd–order accurate. The zero time advance reduces the flexibility in finding suitable error methods, as no individual error method can use both $\mathbf{x_k}$ and $\mathbf{x_{1_k}}$ simultaneously.

16 individual error methods were found that are all A–stable. Two of them are even L–stable. Of course, none of these error methods is of higher order of approximation accuracy than two.

We then proceeded to blend any three of these methods. The best among the 4th–order accurate blended methods is presented in the sequel. It can be written as:

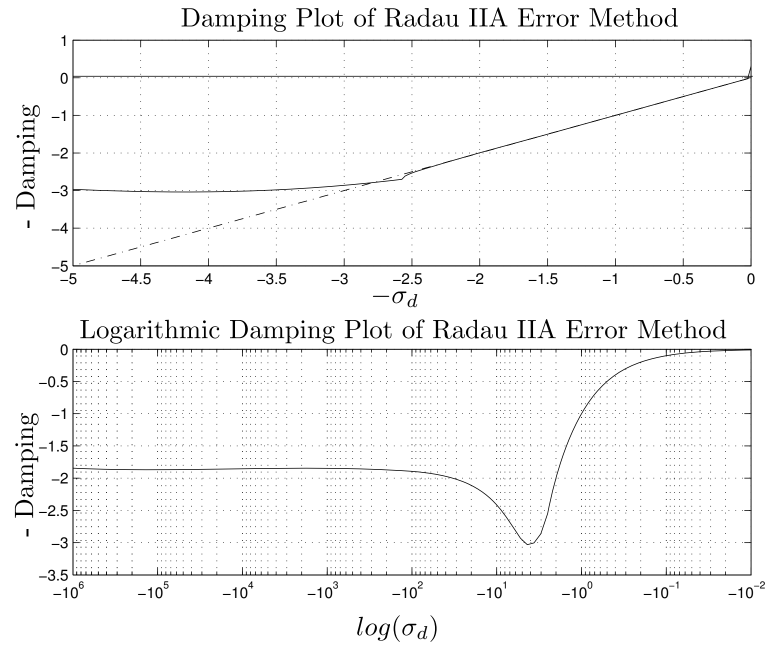

FIGURE 8.22. Damping plot of the blended 5^{th}–order Radau IIA embedding method

$$\begin{aligned}\mathbf{x_{k+1}^{blended}} = {} & \frac{63}{4552} \cdot \mathbf{x_{1_{k-1}}} - \frac{91}{81936} \cdot \mathbf{\dot{x}_{1_{k-1}}} + \frac{1381}{81936} \cdot \mathbf{\dot{x}_{2_{k-1}}} + \frac{3101}{2276} \cdot \mathbf{x_k} \\ & - \frac{393}{4552} \cdot \mathbf{x_{1_k}} + \frac{775}{3414} \cdot \mathbf{\dot{x}_{1_k}} - \frac{165}{569} \cdot \mathbf{x_{2_k}} + \frac{62179}{81936} \cdot \mathbf{\dot{x}_{2_k}} \\ & + \frac{12881}{81936} \cdot \mathbf{\dot{x}_{k+1}} \end{aligned} \tag{8.110}$$

The coefficients of the blending method were calculated using MATLAB's symbolic toolbox.

The embedding method offers a beautiful convex stability domain, as shown in Fig.8.23.

The damping characteristics of the embedding method are shown in Fig.8.24.

The method is characterized by a large asymptotic region and a decently large damping value far out in the left–half complex $\lambda \cdot h$ plane.

8.9 Inlining Partial Differential Equations

Let us return once more to the simulation of parabolic PDEs converted to sets of ODEs using the MOL approach. In Chapter 6 of this book, we

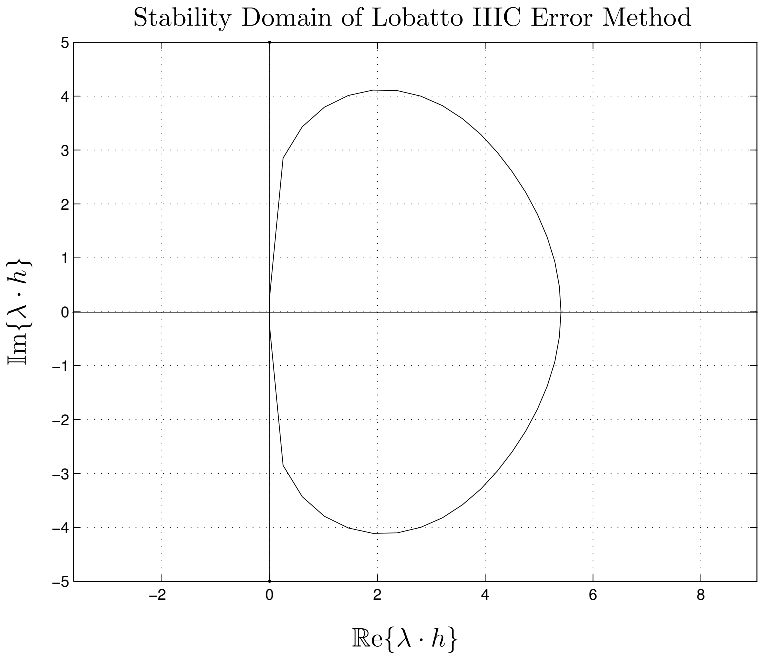

FIGURE 8.23. Stability domain of blended Lobatto IIIC embedding method

simulated these types of problems using a stiff–system solver, such as a BDF algorithm. Whereas this approach worked quite well, the efficiency of the simulations was less than satisfactory. What killed our attempts at solving these problems efficiently was not the step size. The number of function evaluations was actually quite low, at least as long as we computed the Jacobian analytically. What made our simulations excruciatingly slow was the computation of the inverse Hessians.

Let us discuss once more the 1D heat diffusion problem discretized using 5th–order accurate central differences, as described in Eqs.(6.39a–h). We use 50 segments, $n = 50$.

Using the approach advertised in Chapter 6, we ended up with 50 ODEs, requiring a Hessian matrix of size 50×50 to be inverted. More precisely, a linear system of 50 equations in 50 unknowns had to be solved using Gaussian elimination during every iteration step.

Let us now apply inline integration to the problem. Let us start by inlining a variable step and variable order BDF algorithm. We can write the inlined ΔE system in matrix form as follows:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b} \cdot u \tag{8.111a}$$
$$\mathbf{x} = \mathrm{old}(\mathbf{x}) + \bar{h} \cdot \dot{\mathbf{x}} \tag{8.111b}$$

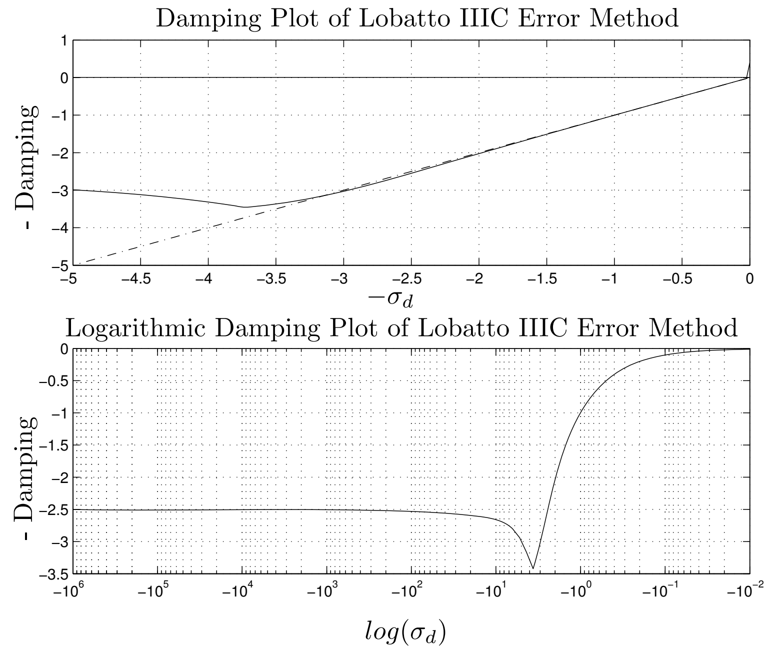

FIGURE 8.24. Damping plot of the blended Lobatto IIIC embedding method

where **A** is the matrix:

$$\mathbf{A} = \frac{n^2}{120\pi^2} \cdot \begin{pmatrix} -15 & -4 & 14 & -6 & 1 & \dots & 0 & 0 & 0 & 0 \\ 16 & -30 & 16 & -1 & 0 & \dots & 0 & 0 & 0 & 0 \\ -1 & 16 & -30 & 16 & -1 & \dots & 0 & 0 & 0 & 0 \\ 0 & -1 & 16 & -30 & 16 & \dots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & 16 & -30 & 16 & -1 \\ 0 & 0 & 0 & 0 & 0 & \dots & -1 & 16 & -31 & 16 \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & -2 & 32 & -30 \end{pmatrix} \tag{8.112}$$

We can no longer hope to tear this equation system by hand. Thus, we encoded our heuristic procedure in a MATLAB routine, and applied that routine to the problem at hand. Since MATLAB works more naturally with matrices than with linked lists, we based our implementation on the structure incidence matrix instead of the structure digraph.

The structure incidence matrix for this problem is shown in Fig.8.25. We numbered the equations such that we started with the state variables, and concatenated them with the state derivatives.

Two trivial tearing structures come to mind immediately. We can either plug Eq.(8.111a) into Eq.(8.111b), and thereby eliminate the state derivatives from the set of iteration variables:

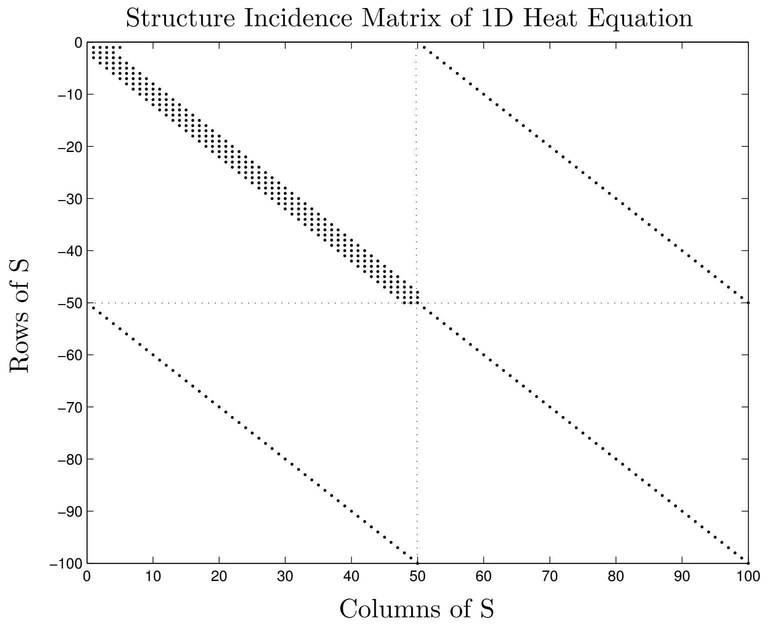

FIGURE 8.25. Structure incidence matrix of inlined 1D heat diffusion problem using a BDF algorithm.

$$\mathbf{x} = \text{old}(\mathbf{x}) + \bar{h} \cdot (\mathbf{A} \cdot \mathbf{x} + \mathbf{b} \cdot u) \tag{8.113}$$

or alternatively, we can plug Eq.(8.111b) into Eq.(8.111a), and thereby eliminate the state variables from the set of iteration variables:

$$\dot{\mathbf{x}} = \mathbf{A} \cdot (\text{old}(\mathbf{x}) + \bar{h} \cdot \dot{\mathbf{x}}) + \mathbf{b} \cdot u \tag{8.114}$$

In either case, we reduce the number of iteration variables back to 50, i.e., we end up with a Hessian of the same size as in Chapter 6.

Let us check, whether our heuristic procedure can do better. Applying the heuristic procedure as proposed, we obtain immediately a solution in 32 residual equations and 32 tearing variables. If we modify our heuristic procedure somewhat by searching for the number of additional equations to be causalized across all unknowns appearing in a candidate residual equation, rather than limiting the search to those variables with the largest number of black (solid) lines attached to them, we obtain a solution with 25 residual equations and 25 tearing variables.

Using this modified heuristic procedure, we have extended the search somewhat, thereby reducing the efficiency of the algorithm, but in return, we have obtained a more economical tearing structure.

We suspect that the solution with 25 iteration variables is indeed the optimal solution, but we are not sure of it. We did not attempt to solve the *np*–complete exhaustive search across all possible combinations of residual equations and tearing variables.

This is a big improvement. The computational effort of the Gaussian elimination algorithm grows quadratically in the size of the linear equation system. Hence by reducing the size of the Hessian from 50×50 to 25×25, we increase the simulation speed by a full factor of four.

Let us now discuss what happens when we inline the 3^{rd}–order accurate Radau IIA algorithm instead. Our ΔE system can now be written down as follows:

$$\dot{\mathbf{y}} = \mathbf{A} \cdot \mathbf{y} + \mathbf{b} \cdot u(t_{k+\frac{1}{3}}) \tag{8.115a}$$

$$\dot{\mathbf{x}} = \mathbf{A} \cdot \mathbf{x} + \mathbf{b} \cdot u(t_{k+1}) \tag{8.115b}$$

$$\mathbf{y} = \text{pre}(\mathbf{x}) + \frac{5}{12} \cdot h \cdot \dot{\mathbf{y}} - \frac{1}{12} \cdot h \cdot \dot{\mathbf{x}} \tag{8.115c}$$

$$\mathbf{x} = \text{pre}(\mathbf{x}) + \frac{3}{4} \cdot h \cdot \dot{\mathbf{y}} + \frac{1}{4} \cdot h \cdot \dot{\mathbf{x}} \tag{8.115d}$$

If we number the variables starting with $\mathbf{y}$, concatenating to it $\mathbf{x}$, then $\dot{\mathbf{y}}$, and finally $\dot{\mathbf{x}}$, the structure incidence matrix assumes the structure shown in Fig.8.26.

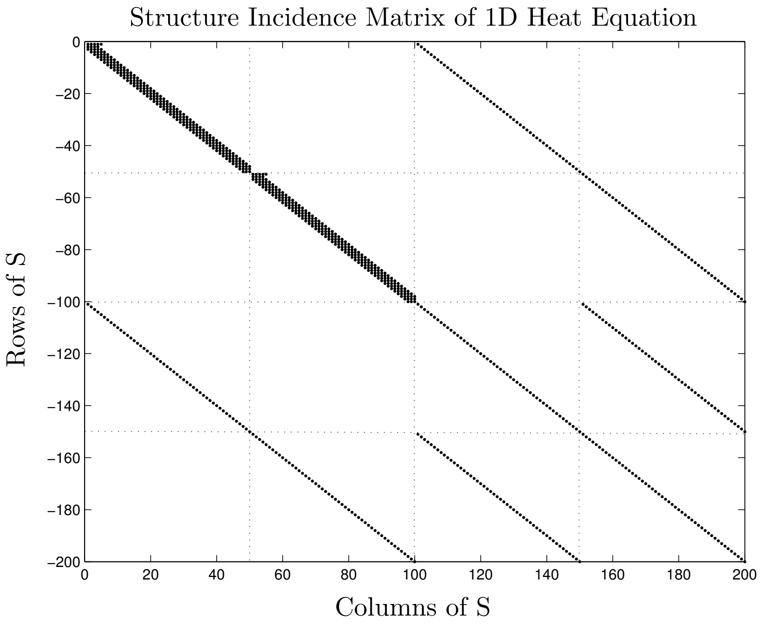

FIGURE 8.26. Structure incidence matrix of inlined 1D heat diffusion problem using Radau IIA.

Two trivial tearing structures come to mind. We can either plug the two sets of state equations into the two sets of integration equations, thereby eliminating all state derivatives from the set of iteration variables:

$$\begin{aligned}\mathbf{y} &= \text{pre}(\mathbf{x}) + \frac{5}{12} \cdot h \cdot (\mathbf{A} \cdot \mathbf{y} + \mathbf{b} \cdot u(t_{k+\frac{1}{3}})) \\ &\quad - \frac{1}{12} \cdot h \cdot (\mathbf{A} \cdot \mathbf{x} + \mathbf{b} \cdot u(t_{k+1})) \qquad (8.116a)\\ \mathbf{x} &= \text{pre}(\mathbf{x}) + \frac{3}{4} \cdot h \cdot (\mathbf{A} \cdot \mathbf{y} + \mathbf{b} \cdot u(t_{k+\frac{1}{3}})) \\ &\quad + \frac{1}{4} \cdot h \cdot (\mathbf{A} \cdot \mathbf{x} + \mathbf{b} \cdot u(t_{k+1})) \qquad (8.116b)\end{aligned}$$

or alternatively, we can plug the two sets of integration equations into the two sets of state equations, thereby eliminating all state variables from the set of iteration variables:

$$\dot{\mathbf{y}} = \mathbf{A} \cdot (\text{pre}(\mathbf{x}) + \frac{5}{12} \cdot h \cdot \dot{\mathbf{y}} - \frac{1}{12} \cdot h \cdot \dot{\mathbf{x}}) + \mathbf{b} \cdot u(t_{k+\frac{1}{3}}) \qquad (8.117a)$$

$$\dot{\mathbf{x}} = \mathbf{A} \cdot (\text{pre}(\mathbf{x}) + \frac{3}{4} \cdot h \cdot \dot{\mathbf{y}} + \frac{1}{4} \cdot h \cdot \dot{\mathbf{x}}) + \mathbf{b} \cdot u(t_{k+1}) \qquad (8.117b)$$

In either case, we end up with 100 iteration variables.

Let us see whether our heuristic procedure can do better. Unfortunately, the algorithm breaks down after having chosen about 60 tearing variables, and after having causalized about 120 equations. The heuristic procedure has maneuvered itself into a corner. Every further selection of a combination of residual equation and tearing variable leads to a structural singularity. Although the algorithm had been programmed to ignore selections that would lead to a structural singularity at once, it hadn't been programmed to backtrack beyond the last selection, i.e., throw earlier residual equations and tearing variables away to avoid future mishap.

This is why we wrote in Chapter 7 that the computational complexity of the heuristic procedure grows quadratically with the size of the equation system *for most applications.* It does so, if no backtracking is required.

I was curious how the tearing algorithm built into Dymola would fare when faced with this problem. I quickly programmed the equation system into Dymola Version 4.1d, and asked for a compilation. Whereas Dymola usually tears equation systems with tens of thousands of equations within a few seconds, it chewed on this problem for a very long time. I watched an entire movie (Animal Farm) on TV, while Dymola was thinking about the problem.

It turned out that the heuristic algorithm built into Version 4.1d of Dymola did not break down. Evidently, it is programmed to backtrack sufficiently to get itself out of a corner. Unfortunately after thinking hard, Dymola came up with one of the two trivial tearing structures.

The above paragraph had been written almost two years ago. Now, before sending the manuscript off to the printer, we decided to run the example once more through the current version of Dymola, which is Version 5.3d. This time around, Dymola came up with an answer after only six seconds of compilation time.

We had sent an earlier version of this chapter to Hilding Elmqvist. Whenever someone stumbles upon an example that the tearing algorithm does not handle well, the good folks up at Dynasim go into overdrive, trying to come up with an improved version of their tearing algorithm as fast as they can.

The answer, however, was still the same. Dymola chose one of the two trivial structures as the most suitable tearing structure for this system. We thus suspect that the trivial structures are indeed the optimal tearing structures in this case, but of course, we aren't sure. Going through an exhaustive search for finding the optimal tearing structure would be too painful to even consider.

Unfortunately, these are bad news. If indeed we pay for using Radau IIA instead of BDF3 with increasing the size of the Hessian by a factor of four, Radau IIA would have to be able to use step sizes that are on average at least 16 times larger than those used by BDF for the same accuracy. Otherwise, Radau IIA is not competitive for dealing with this problem. We doubt very much that Radau IIA will be able to do so.

PDE problems are notoriously difficult simulation problems. Although tearing is a very powerful symbolic sparse matrix technique, it cannot make an intrinsically difficult problem easy to solve.

8.10 Overdetermined DAEs

At this point, we shall resume the discussion of the mechanical pendulum problem that we began towards the end of Chapter 7. The mechanical pendulum schematic is presented once more in Fig.8.27.

We had already come up with a set of causal equations without solvability issues describing the motion of the mechanical pendulum. Let us use the variable substitution technique to come up with a closed–form formula for the tearing variable. Doing so, we find the following explicit ODE description of the pendulum problem.

$$x = \ell \cdot \sin(\varphi) \tag{8.118a}$$

$$v_x = \ell \cdot \cos(\varphi) \cdot \dot{\varphi} \tag{8.118b}$$

$$y = \ell \cdot \cos(\varphi) \tag{8.118c}$$

$$v_y = -\ell \cdot \sin(\varphi) \cdot \dot{\varphi} \tag{8.118d}$$

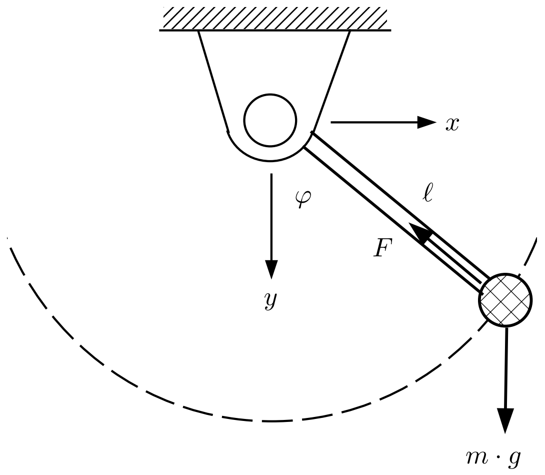

FIGURE 8.27. Mechanical pendulum.

$$dv_x = -\frac{x \cdot \ell \cdot \dot{\varphi}^2 + x \cdot \cos(\varphi) \cdot g}{x \cdot \sin(\varphi) + y \cdot \cos(\varphi)} \tag{8.118e}$$

$$\ddot{\varphi} = \frac{dv_x}{\ell \cdot \cos(\varphi)} + \frac{\sin(\varphi)}{\cos(\varphi)} \cdot \dot{\varphi}^2 \tag{8.118f}$$

$$dv_y = -\ell \cdot \sin(\varphi) \cdot \ddot{\varphi} - \ell \cdot \cos(\varphi) \cdot \dot{\varphi}^2 \tag{8.118g}$$

$$F = \frac{m \cdot g \cdot \ell}{y} - \frac{m \cdot \ell \cdot dv_y}{y} \tag{8.118h}$$

Eq.(8.118e) could have been simplified further, but this is unimportant for the discussion at hand. The formula, as presented above, is the one that Dymola will come up with, since it knows that $\sin^2 \varphi + \cos^2 \varphi = 1$, but it doesn't know to multiply both the numerator and the denominator with ℓ to eliminate the trigonometric functions from the expression.

We know that the pendulum, as described, is a conservative (Hamiltonian) system, since no friction was assumed anywhere. Hence the pendulum, once disturbed, should swing forever with the same frequency and amplitude. The total free energy, E_f:

$$E_f = E_p + E_k \tag{8.119}$$

which is the sum of the potential energy, E_p, and the kinetic energy, E_k, should be constant. The potential energy can be modeled as:

$$E_p = m \cdot g \cdot (y_0 - y) \tag{8.120}$$

and the kinetic energy can be expressed using the formula:

$$E_k = \frac{1}{2} \cdot m \cdot v_x^2 + \frac{1}{2} \cdot m \cdot v_y^2 \tag{8.121}$$

Let us add these three equations to the model.

Let us simulate this problem for a pendulum with $g = 9.81\ m/(sec^2)$, $m = 10\ kg$, $\ell = 1\ m$, $\varphi_0 = +45^o = \pi/4\ rad$, and $\dot{\varphi}_0 = 0\ rad/sec$. Thus, $y_0 = \sqrt{2}/2\ m$.

We shall inline the forward Euler algorithm, thus we add the two equations:

$$\dot{\varphi}_{k+1} = \dot{\varphi}_k + h \cdot \ddot{\varphi}_k \tag{8.122a}$$
$$\varphi_{k+1} = \varphi_k + h \cdot \dot{\varphi}_k \tag{8.122b}$$

We can now simulate the problem by simply iterating over these 13 equations. We shall simulate the problem during 10 *sec* with a fixed step size of $h = 0.01$. The results of this simulation are shown in Fig.8.28.

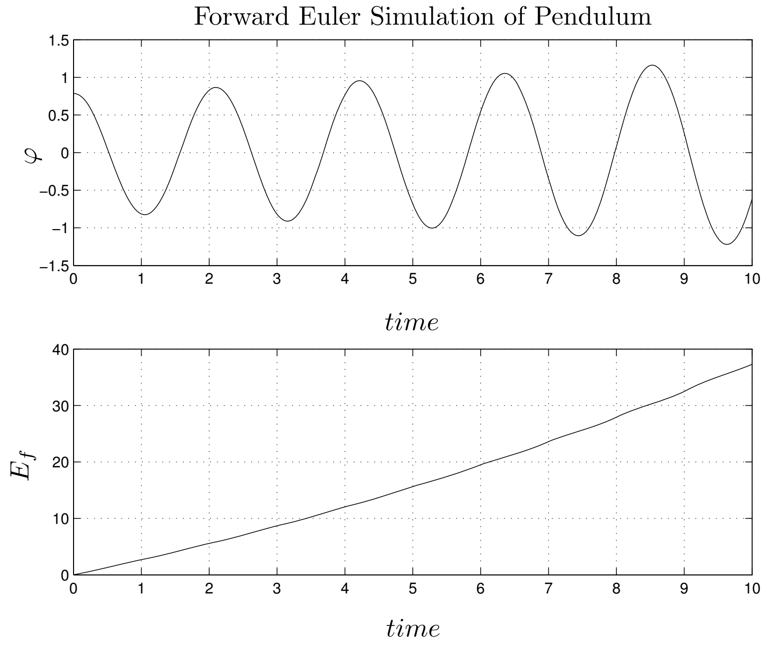

FIGURE 8.28. Inlined FE simulation of mechanical pendulum.

We just solved the world's energy crisis once and for all. Evidently, we are able to generate free energy out of thin air.

Let us see whether backward Euler fares any better. Instead of inlining Eqs.(8.122a–b), we inline the equations:

$$\dot{\varphi} = \text{pre}(\dot{\varphi}) + h \cdot \ddot{\varphi} \tag{8.123a}$$
$$\varphi = \text{pre}(\varphi) + h \cdot \dot{\varphi} \tag{8.123b}$$

Since the backward Euler algorithm is an implicit integration method, we expect to encounter another algebraic loop. The partially causalized structure digraph for this equation system is shown in Fig.8.29.

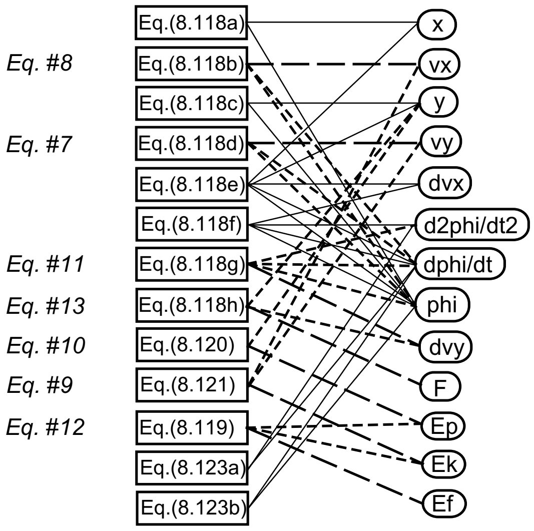

FIGURE 8.29. Partially causalized structure digraph of mechanical pendulum after BE inlining.

We indeed encountered an algebraic loop in six equations and six unknowns. Figure 8.30 shows the completely causalized equation system after a residual equation and a tearing variable have been chosen.

In mechanical systems, it is generally a good idea to select accelerations as tearing variables, and since the model equations had already been causalized before, i.e., each variable appears exactly once to the left side of the equal sign and does so in a linear fashion, it makes sense to use the equation that defines the angular acceleration $\ddot{\varphi}$ as the residual equation.

The causal equations can be read out of Fig.8.30. They are:

$$\dot{\varphi} = \text{pre}(\dot{\varphi}) + h \cdot \ddot{\varphi} \tag{8.124a}$$
$$\varphi = \text{pre}(\varphi) + h \cdot \dot{\varphi} \tag{8.124b}$$
$$y = \ell \cdot \cos(\varphi) \tag{8.124c}$$

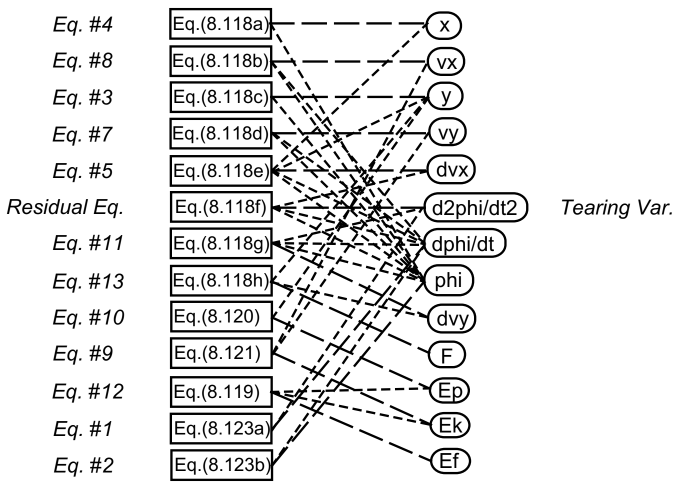

FIGURE 8.30. Completely causalized structure digraph of mechanical pendulum after BE inlining.

$$x = \ell \cdot \sin(\varphi) \tag{8.124d}$$

$$dv_x = -\frac{x \cdot \ell \cdot \dot{\varphi}^2 + x \cdot \cos(\varphi) \cdot g}{x \cdot \sin(\varphi) + y \cdot \cos(\varphi)} \tag{8.124e}$$

$$\ddot{\varphi} = \frac{dv_x}{\ell \cdot \cos(\varphi)} + \frac{\sin(\varphi)}{\cos(\varphi)} \cdot \dot{\varphi}^2 \tag{8.124f}$$

$$v_y = -\ell \cdot \sin(\varphi) \cdot \dot{\varphi} \tag{8.124g}$$

$$v_x = \ell \cdot \cos(\varphi) \cdot \dot{\varphi} \tag{8.124h}$$

$$E_k = \frac{1}{2} \cdot m \cdot v_x^2 + \frac{1}{2} \cdot m \cdot v_y^2 \tag{8.124i}$$

$$E_p = m \cdot g \cdot (y_0 - y) \tag{8.124j}$$

$$dv_y = -\ell \cdot \sin(\varphi) \cdot \ddot{\varphi} - \ell \cdot \cos(\varphi) \cdot \dot{\varphi}^2 \tag{8.124k}$$

$$E_f = E_p + E_k \tag{8.124l}$$

$$F = \frac{m \cdot g \cdot \ell}{y} - \frac{m \cdot \ell \cdot dv_y}{y} \tag{8.124m}$$

The first six of these equations, Eqs.(8.124a–f), constitute the algebraic loop. This time, we used Newton iteration in the single tearing variable, $\ddot{\varphi}$, to solve the algebraic loop. We computed the Hessian by means of algebraic differentiation.

The simulation results are shown in Fig.8.31.

It didn't work any better than before. This algorithm is losing energy, where it shouldn't. The result is easily explainable. This is a conservative system. The two eigenvalues of its Jacobian are located on the imaginary axis, at least on average. However, the numerical stability domain of the FE

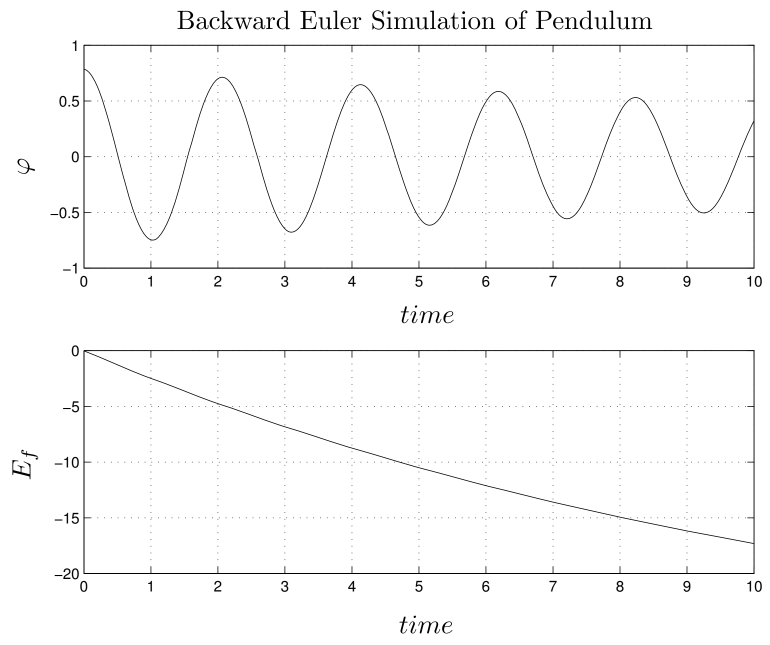

FIGURE 8.31. Inlined BE simulation of mechanical pendulum.

algorithm loops into the left–half complex plane. Thus, the two eigenvalues of the system are seen as mildly unstable, and the oscillation is growing. The algorithm adds energy to the system. On the other hand, the stability domain of the BE algorithm loops into the right–half complex plane. Consequently, the two marginally stable eigenvalues are seen as mildly damped, and the oscillation decays.

An F–stable algorithm, such as the BI technique, should be expected to work better. Let us implement BI2 as a cyclic method, toggling between a step of FE followed by a step of BE. The simulation results are presented in Fig.8.32.

The approach worked. Yet, it only worked, because we were able to analyze the problem and come up with a suitable solution. The code itself still has no inkling that it is supposed to conserve the free energy. It does so by accident rather than by design.

Let us try to change that. We shall force the backward Euler algorithm to preserve the free energy. To this end, we simply add the equation:

$$E_f = 0 \tag{8.125}$$

to the set of equations.

This is a completely new situation. We haven't added any new variables to the set of equations. We only added another equation. Thus, we now have 14 equations in 13 unknowns. Clearly, this problem is constrained.

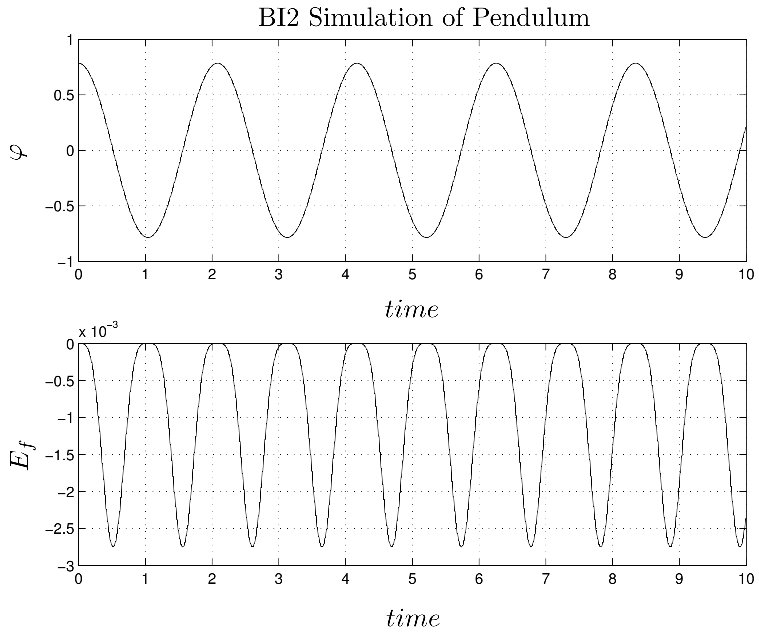

FIGURE 8.32. Inlined BI2 simulation of mechanical pendulum.

If we present this problem to the Pantelides algorithm, it will differentiate itself to death, or rather, until the compiler runs out of virtual memory. The Pantelides algorithm always adds exactly as many equations as variables, thus after each application of the algorithm, the number of equations is still one larger than the number of variables.

Inlining again saves our neck. We simply add the constraint equation to the iteration equations of the Newton iteration. Thus, the set of zero functions can now be written as:

$$\mathcal{F} = \begin{pmatrix} \ddot{\varphi}_{new} - \ddot{\varphi} \\ E_f \end{pmatrix} \tag{8.126}$$

and therefore:

$$\mathcal{H} = \begin{pmatrix} \partial\ddot{\varphi}_{new}/\partial\ddot{\varphi} - 1 \\ \partial E_f/\partial\ddot{\varphi} \end{pmatrix} \tag{8.127}$$

The Newton iteration can be written as:

$$\mathcal{H}^\ell \cdot \mathbf{dx}^\ell = \mathcal{F}^\ell \tag{8.128a}$$

$$\mathbf{x}^{\ell+1} = \mathbf{x}^\ell - \mathbf{dx}^\ell \tag{8.128b}$$

However, $\mathcal{H}$ is no longer a square matrix. It is now a rectangular matrix with 2 rows and 1 column. In general with n model equations and p constraints, the Hessian turns out to be a rectangular matrix with $n+p$ rows and n columns. Thus, Eq.(8.128a) is overdetermined. It cannot be satisfied exactly. The $\mathbf{dx}$–vector can only be determined in a least square sense. This can be accomplished by multiplying Eq.(8.128a) from the left with $\mathcal{H}^*$, i.e., with the Hermitian transpose of $\mathcal{H}$. If the rank of $\mathcal{H}$ is n, then $\mathcal{H}^* \cdot \mathcal{H}$ is a Hermitian matrix of full rank. Thus, we can compute $\mathbf{dx}$ as:

$$\mathbf{dx} = (\mathcal{H}^* \cdot \mathcal{H})^{-1} \cdot \mathcal{H}^* \cdot \mathcal{F} \tag{8.129}$$

where $(\mathcal{H}^* \cdot \mathcal{H})^{-1} \cdot \mathcal{H}^*$ is called the *Penrose–Moore pseudoinverse* of $\mathcal{H}$. In MATLAB, this can be abbreviated as:

$$\mathbf{dx} = \mathcal{H} \backslash \mathcal{F} \tag{8.130}$$

The results of the simulation are shown in Fig.8.33.

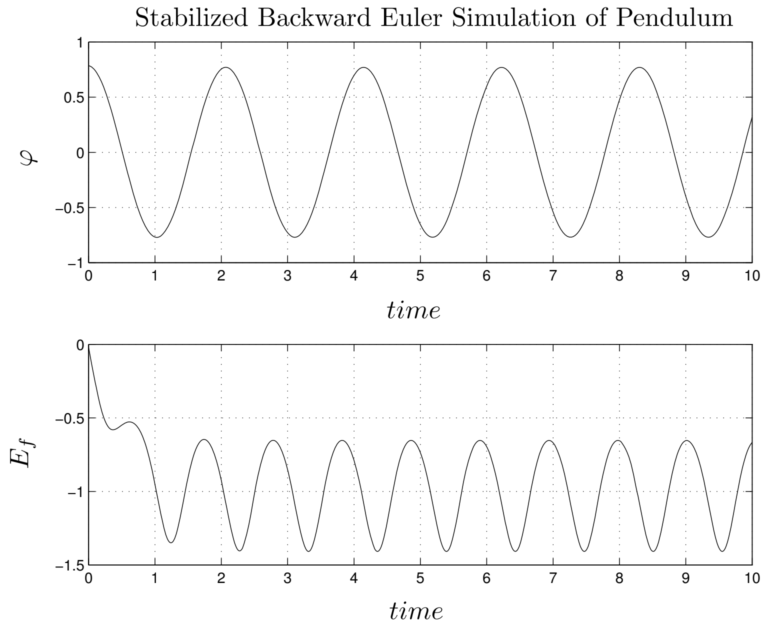

FIGURE 8.33. Inlined stabilized BE simulation of mechanical pendulum.

The oscillation has indeed been stabilized. Of course, the equation:

$$\mathcal{F} = 0 \tag{8.131}$$

can no longer be solved precisely. The equation system does not contain enough freedom to do so. Yet, the error is minimized in a least square sense, and both the oscillation and the free energy are now stable by design. Initially, the approach still loses a bit of free energy, but the loss stops after the solution is stabilized. The solution using backinterpolation turns out to be better, but the solution using an overdetermined equation set is more robust.

There are DAE solvers on the market that can handle overdetermined DAEs, such as ODASSLRT (a "dialect" of DASSL) [8.12] and MEXX (a code using Richardson extrapolation) [8.20]. Overdetermined DAE solvers have become popular primarily among specialists of multibody dynamics, and the early codes tackling this problem indeed evolved in the engineering community. Most of these early codes were quite specialized. More recently, the problem was discovered by mainstream applied mathematicians [8.15], and it can therefore be expected that more general–purpose codes for the numerical solution of overdetermined DAE systems will soon become available.

Yet, the problem of merging overdetermined linear system solvers with general–purpose DAE codes is a difficult one. Most DAE solvers cannot deal with higher–index problems, yet overdetermined DAEs have much in common with higher–index problems. In our view, inlining overdetermined DAEs is generally a better approach than trying to keep the model equations separate from the simulator equations.

Most applied mathematicians have shunned away from the inlining approach, because inlining without a powerful model compiler, such as Dymola [8.9, 8.10], is a toy. Drawing structure digraphs by hand only works for toy problems.

Hairer and his colleagues thus went a different route [8.15]. Rather than constraining the DAE system, they generalized on the BI2 solution presented earlier in this section. It was not by accident that the BI2 solution produced the correct answer to the problem.

To understand this result, the reader needs to remember our introduction to the backinterpolation algorithms in Chapter 3 of this book. We recognized that we can implement a class of implicit Runge–Kutta algorithms by integrating the regular explicit Runge–Kutta algorithms backward through time. As we simply replace the step size h by $-h$, the stability domains of these methods get mirrored on the imaginary axis of the complex $\lambda \cdot h$–plane.

Some algorithms do not change, when h is replaced by $-h$. For example, the trapezoidal rule:

$$\mathbf{x_{k+1}} = \mathbf{x_k} + \frac{h}{2} \cdot (\dot{\mathbf{x}}_\mathbf{k} + \dot{\mathbf{x}}_\mathbf{k+1}) \tag{8.132}$$

turns into:

$$\mathbf{x_k} = \mathbf{x_{k+1}} - \frac{h}{2} \cdot (\dot{\mathbf{x}}_\mathbf{k+1} + \dot{\mathbf{x}}_\mathbf{k}) \tag{8.133}$$

i.e., the formula doesn't change. Such an ODE solver is called a *symmetric integration algorithm.* The stability domains of symmetric ODE solvers are symmetric to the imaginary axis. In particular, all of the F–stable integration algorithms introduced in Chapter 3 are symmetric ODE solvers.

Symmetric integration algorithms are not only symmetric to the imaginary axis w.r.t. their stability properties, but also w.r.t. their damping properties. Thus, symmetric integration algorithms are accompanied by symmetric order stars as well.

This symmetry can be exploited in the simulation of Hamiltonian systems. At least, if we carefully choose our step size to be in sync with the eigenfrequency of oscillation of the system, we can ensure that the damping errors committed during the integration over a full period cancel out, such that the solution at the end of one cycle coincides with that at the beginning of the cycle.

Yet, we still prefer the constrained solution proposed in this section, as it is considerably more robust. It works with any numerical integration scheme and enforces the physical constraint directly rather than indirectly.

8.11 Electronic Circuit Simulators

One important application area where DAEs are frequently used is electronic circuit modeling. Let us briefly relate the topics of Chapters 7 and 8 of this book to the discussions presented in Chapters 3 and 6 of the companion book on *Continuous System Modeling* [8.5].

We have seen that object–oriented modeling of electrical and electronic circuits invariably leads to implicit DAE descriptions. We have furthermore seen that the resulting sets of DAEs are often index–2 models.

You, the reader, may have come to the conclusion that whether or not a set of DAEs describing an electrical circuit presents itself as a higher–index model depends on the topology of the circuit. However, that conclusion is too simple. The perturbation index of a model is influenced by the abstraction mechanism chosen in its mathematical description. In the case of nonlinear systems, it even depends on the selection of state variables, as nonlinear transformations performed on a model can influence the perturbation index.

In an explicit ODE description of a system, the state variables are predetermined. They are simply the outputs of the integrators. However in an implicit DAE description, the answer is no longer as clear cut. First and second derivatives can show up multiple times anywhere within the implicit equations. How do we even know, how many degrees of freedom a DAE model really has? Here we have a true choice in deciding, which state variables to use, and the perturbation index of the model is dependent on that choice.

Inline integration blurs the situation even further. After inlining, all variables have become algebraic variables. The number of initial values needed to simulate an inlined model depends on the number of tearing variables. We need to specify one initial guess for each tearing variable, as well as initial conditions for all variables that appear in 'pre(.)' clauses.

In the companion book, we started out in Chapter 3 explaining the derivations of circuit equations in terms of either *mesh currents* or *cutset potentials.* We chose a "tree" that defined either a minimal set of mesh currents or a minimal set of cutset potentials.

We now understand much better, what it is that we accomplished. We designed techniques to come up with small sets of *tearing variables.* The mesh currents assume the role of tearing variables, if we work with mesh equations, whereas the cutset potentials assume the same role, when we work with cutset (node) equations.

Branch currents and/or branch voltages are poor choices as state variables, because they frequently lead to higher–index DAE models. If we place two capacitors in parallel, we cannot choose the voltages across these two capacitors as independent state variables. Similarly, if we place two inductors in series, we cannot select the currents flowing through them as independent state variables. If we work with such selections, we invariable end up with index–2 models.

The problem disappears if we choose a subset of either mesh currents of cutset potentials as state variables. These are always independent of each other by design. The most difficult problem is to decide, which variables and how many to select as tearing variables.

Commercial electronic circuit simulators, such as Spice [8.26, 8.39] or Saber [8.24], work with the node potentials as their tearing variables [8.40]. Yet, rather than substituting the equations into each other, as we proposed in the companion book, they simply write the equations down as is, and iterate on them using either Newton iteration or at least fixed–point iteration.

Let us start with the simplest case: an electronic circuit without voltage sources and inductors. Spice [8.26, 8.39] uses all of the node potentials as tearing variables. In Spice, this is called the *nodeset.* Evidently, this is not a minimal set, but it makes the algorithm of finding the tearing variables trivial. In fact, there are even two different nodesets in use. The *reduced nodeset* contains all of the node potentials of user–defined nodes, whereas the *complete nodeset* also includes the internal nodes of the circuit expansions of the active devices (primarily BJT and MOS transistors).

We can formulate *Kirchhoff's Current Law (KCL)* as follows:

$$\boldsymbol{\Psi} \cdot \mathbf{i} = 0 \tag{8.134}$$

where $\mathbf{i}$ is the vector of branch currents, and $\boldsymbol{\Psi}$ is the *reduced node incidence matrix.* $\boldsymbol{\Psi}$ has as many rows as there are nodes in the circuit, and it has as many columns as there are branches. It is called the *reduced* node incidence matrix, because it only considers those nodes and branches that are explicitly formulated in the model, excluding the expansions of the active devices. The element Ψ_{ij} has a value of $+1$, when the branch leaves the node, i.e., when the positive direction of the branch current is away from the node. If assumes a value of -1, if the branch arrives at the node, and it assumes a value of 0, if the branch is not connected to the node at all. Eq.(8.134) simply states that the sum of all currents into a node is zero.

The equation:

$$\mathbf{u} = \boldsymbol{\Psi}^{\mathbf{T}} \cdot \mathbf{v} \tag{8.135}$$

relates the node potentials $\mathbf{v}$ to the branch voltages $\mathbf{u}$.

Finally, the equation:

$$\mathbf{i} = \mathbf{g}(\mathbf{u}, \dot{\mathbf{u}}, \mathbf{v}, t) \tag{8.136}$$

captures the element law for each of the branches, relating the voltages and potentials to the currents.

Capacitors are implemented using the DAE formulation of a BDF formula, i.e.:

$$\dot{\mathbf{u}}_{\mathbf{C}} = \frac{\mathbf{u}_{\mathbf{C}} - \text{old}(\mathbf{u}_{\mathbf{C}})}{\bar{h}} \tag{8.137}$$

Notice that Spice made use of this approach years before DASSL was written, but since the program was specialized to dealing with electronic circuits only, the mathematical community hardly paid any attention to it.

If all elements are either current sources, or resistors (including the nonlinear diodes), or capacitors, we are already in business. If we assume the node potentials, $\mathbf{v}$, as known, we can use Eq.(8.135) to determine all branch voltages, $\mathbf{u}$. We can then use the implicit numerical differentiators of Eq.(8.137) to compute the derivatives of the voltages for each of the capacitors. We can then use the elemental laws for each branch, Eq.(8.136), to compute the branch currents.

Hence we can set up the Newton iteration as follows:

$$\mathcal{F} = \boldsymbol{\Psi} \cdot \mathbf{i} = 0 \tag{8.138a}$$

$$\mathcal{H} = \boldsymbol{\Psi} \cdot \frac{\partial \mathbf{i}}{\partial \mathbf{v}} \tag{8.138b}$$

$$\mathbf{v}^{\ell+1} = \mathbf{v}^{\ell} - \mathcal{H} \backslash \mathcal{F} \tag{8.138c}$$

$\mathcal{H}$ is a square matrix with as many rows and columns as the circuit has nodes.

For the Newton iteration to converge properly, the circuit simulator will need a consistent set of initial values for all tearing variables. This is why Spice needs to compute an *OP–point*, i.e., a consistent set of initial conditions for the loop variables, before the "transient analysis" (simulation) can begin. If the initial OP–point does not converge, the program is in difficulties.

To overcome this problem, some Spice dialects offer automated *source ramping*. If all sources are initially set to zero and if all active devices are switched off, the initial nodeset is trivial: all node potentials are equal to zero. Then, the voltages are smoothly ramped up to their desired initial values in a pre–simulation run, and are then kept at their final (initial) values for some time to give the circuit a chance to settle into a steady state. The resulting node potentials are then used as the initial nodeset for the subsequent true transient analysis. Due to the special nature of circuit topologies, we have a simple and systematic way of determining a consistent set of initial conditions, a luxury that we do not have in all DAE simulations. Ramping had been described in Chapter 6 of the companion book.

How do we deal with inductors? Inductors can be implemented in similar ways as the capacitors. However, rather than using the BDF formula in its derivative form, we use it in its integral form:

$$\mathbf{i_L} = \text{old}(\mathbf{i_L}) + \bar{h} \cdot \frac{d\mathbf{i_L}}{dt} \tag{8.139}$$

Given the branch voltage, we compute the derivative of the current using the elemental law, then use the BDF formula in its integral form to find the current.

How do we deal with the ideal independent voltage sources? In the companion book, we proposed to move independent voltage sources into neighboring branches of the circuit. Commercial circuit simulators do it differently.

Let us assume an ideal voltage source is placed in branch i, which is located between node j and node k. The current through the voltage source is free to assume any value it needs to assume. It is not constrained by an elemental law.

Consequently, we can eliminate one row of the $\mathcal{F}$ vector, either the element number j, or the element number k, corresponding to either the j^{th} or the k^{th} row of the $\mathbf{\Psi}$ matrix. We add the equation specified by the eliminated row to the set of equations computing the currents, Eq.(8.136), solved for the unknown current through the voltage source.

Since the number of nodes has remained the same as before, we are now lacking one equation in $\mathcal{F}$ for the Newton iteration. We replace it by the new zero function:

$$v_j - v_k - u_i = 0 \tag{8.140}$$

We can now proceed as before.

This is a fairly generic description of how electronic circuit simulators may be implemented. The different simulators on the market vary in implementational details of how they make use of the above equations. In the circuit simulation literature, Eqs.(8.134–8.136) are generally called the *Sparse Tableau Equations* [8.14, 8.25, 8.26].

Many circuit simulators shun away from estimating the complete Hessian, and therefore, limit themselves to a fixed–point iteration only. In that case, we must iterate over all the loop variables, i.e., the tearing approach breaks down. In general, we are dealing with $n_n + 2\ n_b$ equations in the same number of unknowns, where n_n denotes the number of nodes, and n_b

is the number of branches. It may therefore be advantageous to reduce the number of variables contained in the loop. To this end, Eq.(8.134) can be combined with Eq.(8.136) in the following way:

$$\mathbf{\Psi} \cdot \mathbf{g}(\mathbf{u}, \dot{\mathbf{u}}, \mathbf{v}, t) = 0 \tag{8.141}$$

thereby eliminating the currents altogether from the iteration loop. Again, there exist different variations of this general scheme, usually referred to in the literature under the name *Modified Nodal Analysis (MNA)* [8.18, 8.25, 8.26].

All of the classical circuit simulators have in common that they limit the symbolic preprocessing to the interpretation of the network topology. The entire analysis is done numerically, using the equations pretty much as they come. Contrary to a general–purpose DAE solver, such as DASSL, the integration of the storage variables is performed in a decentralized manner, i.e., for each storage element separately.

The approach only works, because the structure of all equations is predetermined. For this reason, circuit simulators cannot be combined with other tools to form e.g. mechatronic system simulators. Even the thermal analysis offered by the traditional circuit simulators is fairly limited. They all allow a user to simulate a circuit at different temperatures (the circuit parameter values, such as R and C, can be specified as functions of temperature), but it is impossible to simulate how the flow of electrical current through the circuit heats up the circuit, and then simulate the effects of the change in temperature on the circuit's electrical performance simultaneously. This cannot be done, because the structure of the equations would change in such a way that it would violate the assumptions on which the modified nodal analysis are based.

A mixed symbolic and numerical approach, as pursued e.g. in Dymola, is therefore considerably more flexible and powerful. To preserve this generality, the developers of Dymola made it a point to make sure that Dymola knows absolutely nothing about physics. All it understands are mathematical algorithms. The entire physical knowledge is encoded in the models themselves, not in the underlying algorithms that are built (hardwired) into Dymola.

The price that we pay for this generality is small. Electronic circuit simulations performed by Dymola are as fast and accurate as their Spice or Saber counterparts. Furthermore, the maintenance of the electronic model library is considerably easier in Dymola than in either Spice or Saber, because of the object–oriented nature of the Dymola modeling environment. Furthermore, Dymola enables the user to simulate electronic circuits that are parts of larger mechatronic systems in a mechatronic system simulation. They also allow the electrical and thermal interactions of integrated circuits to be explored in full, e.g. in the design of packages [8.35, 8.36].

8.12 Multibody System Dynamics Simulators

Whereas Chapters 3 and 6 of the companion book describe fairly accurately the state–of–the–art of electronic circuit modeling, Chapters 4 and 7 *don't* describe state–of–the–art multibody system (MBS) dynamics modeling. This topic is simply too advanced to be presented in full in a general–purpose modeling class. The companion book limited a detailed discussion to one–dimensional devices. Unfortunately, this view does not extend smoothly to two– or even three–dimensional devices, such as robots or vehicles.

The problem is the following: Asking a user to come up with an ODE model describing the dynamics of a complicated MBS is not a practical proposition. DAE models, on the other hand, are fairly easy to derive. To this end, one simply describes the dynamics of each body separately, and adds the interactions between bodies as constraint equations. However, this usually leads to index 3 models with nasty nonlinear constraints. Relying on the Pantelides algorithm to blindly reduce the index down to index 1 will lead to an explosion in the complexity of the resulting equations, unless the user is very cautious about how he or she chooses the variables in the model. Furthermore, it often leads to models with solvability issues.

Selection of an appropriate coordinate system is absolutely essential. In the case of tree-structured robots, special selections of generalized coordinates both for the description of the *direct MBS dynamics* (motor torques are inputs, and positions and velocities of the end–effector are outputs), as well as for the description of *inverse MBS dynamics* (desired end–effector positions and velocities are inputs, and necessary motor torques to achieve those are outputs) have been found that don't lead to algebraic loops at all. Using these generalized coordinates, the number of equations will grow linearly in the number of bodies described in the model. Algorithms implementing this methodology are called *order–n* algorithms or order–f algorithms, depending on the particular reference consulted [8.2, 8.11, 8.21, 8.32].

MBS topologies with closed kinematic loops are more problematic, and the final word on how to efficiently model such systems hasn't been spoken yet. However, let us at least explain briefly how such systems are currently being modeled. Any tree–structured MBS can be brought into the form:

$$\mathbf{M}(\mathbf{q}, t) \cdot \ddot{\mathbf{q}} = \mathbf{h}(\mathbf{q}, \dot{\mathbf{q}}, t) + \mathbf{f}(\mathbf{q}, t) \tag{8.142}$$

where $\mathbf{q}$ are the generalized positions (including angular positions) of the tree-structured MBS, $\mathbf{M}$ is the so–called mass matrix, $\mathbf{h}$ models the effects of body–fixed coordinate systems (Coriolis and centripetal forces) as well as friction phenomena, and $\mathbf{f}$ are the generalized forces (including torques) acting on the joints.

If an MBS has kinematic constraints (closed kinematic loops), we can first cut these constraints open, thereby transforming the kinematically constrained MBS into a tree–structured MBS. For this so modified MBS, we can derive Eq.(8.142), e.g. using the algorithm described in [8.21]. We then add the constraints as additional constraint equations back into the overall DAE description, thereby transforming the carefully formulated index 1 model back into an index 3 model. Luckily, there often aren't too many of these additional constraint equations, and the Pantelides algorithm may work quite decently. The resulting index 1 model can then either be solved directly using a DAE solver (possibly using a symbolically generated Hessian matrix), or we can try to reduce the model further to index 0 by solving the algebraic loops. Luckily, the special structure of mechanical manipulators suggests immediately a set of tearing variables, namely the generalized accelerations, $\ddot{\mathbf{q}}$.

Meanwhile, an MBS library has been designed for Dymola [8.30] that enables even non–specialists of MBS dynamics to formulate efficient sets of DAEs for multibody systems in an effective object–oriented manner. The library contains models for most of the components that a user might need, such as different types of joints (revolute joints, prismatic joints, screws, etc.), rigid bodies and their connections, different types of force elements, and so on. A top–down description of the topology of an arbitrarily connected three–dimensional (or two–dimensional) tree–structured robot in an object–oriented fashion is made a fairly simple undertaking using the MBS library. The generated code compares favorably with other commercial MBS systems such as Adams [8.17, 8.27], or SD–Fast [8.19] in terms of run–time efficiency.

However, contrary to the more specialized tools, Dymola lends itself elegantly to modeling and simulation of general mechatronic systems, i.e., the drive chains, motors, and controllers of these robots can be described together with the MBS dynamics in a unified framework [8.3, 8.29, 8.30, 8.33].

Dymola does a fairly good job of coming up on its own with suitable tearing structures even in the case of closed kinematic loops. However, Dymola's multibody systems (MBS) library [8.30] still supports Dymola in this task by making sure that the (fully automated) tearing algorithm starts out with a suitable set of equations. Let us explain.

In the MBS library, *translational variables* are being carried along in the inertial frame, whereas *rotational variables* are described in a body–centric coordinate system. This by itself already helps with generating efficient simulation code. Yet, the decision requires that the library perform coordinate transformations from one body to the next in the rotational variables.

The coordinate transformation inside a joint model can be written as:

$$\mathbf{x_2} = \mathbf{R} \cdot \mathbf{x_1} \tag{8.143}$$

where the vectors $\mathbf{x_1}$ and $\mathbf{x_2}$ contain generalized coordinates of the bodies to the left and to the right, and the matrix $\mathbf{R}$ is a rotation matrix.

This process was demonstrated in the research section of Chapter 4 of the companion book for a six–degree–of–freedom Stanford robotic arm using *Denavit–Hartenberg (DH) coordinates* [8.7].

Depending on where the inertial frame is, we need to use either Eq.(8.143) or the inverse equation:

$$\mathbf{x_1} = \mathbf{R}^{-1} \cdot \mathbf{x_2} \tag{8.144}$$

However, since $\mathbf{R}$ is an *orthogonal matrix*, the inverse of $\mathbf{R}$ can also be written as a transpose:

$$\mathbf{x_1} = \mathbf{R}^T \cdot \mathbf{x_2} \tag{8.145}$$

which is more economical.

Unfortunately, Dymola, although offering a matrix manipulation language similar to that of MATLAB, doesn't understand the concept of orthogonal matrices. The reason is that Dymola, in order to provide full flexibility for causalizing equations, expands all matrix expressions into scalar expressions upon compilation. In the scalar version, the orthogonality of the $\mathbf{R}$–matrix is no longer easily visible. Thus, Dymola will do it the hard way and solve a linear equation system, whenever it needs the transformation equations in reversed causality.

In order to support Dymola in producing efficient simulation code, the MBS library keeps track of where the root (inertial frame) is, and provides the coordinate transformation to Dymola in a form similar to:

```
if rooted($frame\_a$) then
    $x_2 = R * x_1$
else
    $x_1 = \text{transpose}(R) * x_2$
end if;
```

where $frame\_a$ denotes the connector of the body to the left. In this way, Dymola starts out with the best suited equation set when looking for tearing variables using its built–in tearing algorithm.

8.13 Chemical Process Dynamics Simulators

Chemical processes are another prime candidate for DAE formulations. Here, the problems are again quite different. Chemical processes are modeled through highly nonlinear equations describing the (i) reaction rate dynamics, (ii) mass flow dynamics, (iii) thermal dynamics, and (iv) energy balance.

In Chapters 8 and 9 of the companion book, the basic equations describing chemical processes were introduced. However, these models mostly served the purpose of furthering the understanding of what is going on physically within a chemical reaction system. In reality, chemical reaction processes are invariably distributed parameter systems that should be described by PDEs. Also, there is no such thing as a homogeneous medium. Consequently, we are dealing with accuracy problems. If a chemical engineer can determine, in a simulation run, what is going on in the real process with an accuracy of 1%, he or she is very lucky.

For these reasons, it isn't warranted for practical simulations to deal with the exact equations. Why use a very complicated model if it is inaccurate anyway? Moreover, the energy balance equations have a much faster time constant than e.g. the mass balance equations. Therefore, chemical reaction processes are usually approximated by implicit ODEs describing average reaction rates, implicit ODEs describing mass continuity, implicit ODEs describing average temperatures, and algebraic constraint equations for energy balances and the equation of state [8.22, 8.37].

The result is a set of higher–index DAEs with nasty nonlinearities. Index reduction can usually be accomplished more easily here than in the case of mechanical systems, since the nonlinearities are usually polynomial rather than trigonometric. Consequently, the Pantelides algorithm will work fine. In fact, it is for chemical process engineers that Constantinos Pantelides developed his algorithm in the first place.

Solving the resulting algebraic loops is a different matter. Various tearing strategies have been described for such purposes [8.23], but they are very specialized and too complicated for our taste. Whenever we are confronted in physics with an equation or an algorithm that is very complicated, we should get suspicious that, probably, we are looking at the problem from a wrong angle. A typical example are the equations describing celestial dynamics when adopting a geostationary world view. We believe strongly that all physical laws governing this universe are basically simple. Complexity is introduced into this universe of ours by having many different –and simple– equations interact with each other. Equations can become more messy when we are forced to average or aggregate (as is the case in chemical process engineering), but the DAEs themselves are still fairly harmless. It is the conversion to explicit ODE form that makes them become truly messy ... and this has to do with the previously made simplifications (aggregations). Thus, we should probably abstain from trying to convert these equations to explicit ODE form.

So, if we stay with DAEs, where does tearing fit in? Isn't tearing a concept related to the transition from DAEs to ODEs? In chemical process engineering, tearing has mostly been used to simplify the process of fixed–point iteration of the resulting set of algebraic equations after inlining the integration method into the implicit state–space model.

A first attempt at object–oriented modeling of chemical process dynamics has recently been reported [8.28]. Bernt Nilsson didn't use Dymola for that purpose, but its twin brother, called Omola [8.1].

8.14 Summary

In this chapter, mixtures of symbolic and numerical tools for the treatment of differential and algebraic systems of equations were introduced. DAE formulations of dynamic models are very natural to applications in many areas of science and engineering. However, a direct approach to numerically dealing with the DAEs as they present themselves initially may not be the wisest thing to do. Automated symbolic preprocessing of the DAE models into a form that is better suited for the subsequent numerical integration is an exciting new development in modeling and simulation research.

A symbolic manipulation tool, Dymola, was introduced that has been specifically designed for such purpose. Dymola is the most advanced tool for that purpose currently on the market. While more generic symbolic formula manipulation programs, such as Mathematica or Reduce, could be used alternatively, they are not efficient for the task at hand. Dymola has already proven its utility in very large MBS models, for example.

The chapter introduced the most common numerical algorithms for dealing with the resulting set of index 1 DAEs, namely the BDF methods and fully–implicit Runge–Kutta algorithms, and explained how they work. It turns out that the solution of (potentially large) sets of algebraically coupled equations are at the heart of dealing with DAE systems. It was shown how the problem of numerical DAE solution is reduced to one of Newton iteration, and symbolic generation of the Hessian matrix was proposed as an additional tool for improvement of efficiency in numerical DAE solution.

By inlining the symbolic equations describing the integration algorithm into the model, the derivative operator disappears from the model altogether, and Dymola generates directly a set of difference equations that can be solved by simply looping over the model.

The chapter ended with three application areas: electronic circuits, multibody system dynamics, and chemical process engineering. These application areas demonstrate vividly the convenience and importance of a mixture of symbolic and numerical tools that can deal with DAE formulations.

Interestingly enough, many topics became simpler rather than more complicated when looking at them from a DAE rather than an ODE perspective. Yet, the research area of DAEs is much younger than that of ODEs. This can be explained easily. When scientists and engineers became interested in numerically simulating dynamic phenomena, no computers were available as yet. Armies of "applied mathematicians" (at that time not a highly respected tag for a mathematician) were employed and placed in a room for weeks in a row. Pipelining algorithms were designed such that the first mathematician could calculate the first step at time zero, then pass his or her result on to the next mathematician who would then solve step two at time zero, while the first mathematician would start on step one for time h, etc. Implicit algorithms don't lend themselves that easily to pipelining, and so, the focus was entirely on explicit algorithms, where

ODE formulations are most natural.

Later on when computers became available, engineers and scientists had become so used to ODE formulations that it took a while before they reconsidered the issue. The most exciting part of the story is that here is a research area where not too much has happened yet. It has become very difficult to hit a mark in numerical ODE solutions. Too many excellent mathematicians have already ploughed the field in hope of finding a leftover grain that might grow and bloom and flourish. This is not so in numerical DAE solution. This is therefore an excellent field for young applied mathematicians (a proud and respected lot by now) to do research in.

8.15 References

[8.1] Mats Andersson. Omola — An Object–Oriented Modelling Language. Technical Report TFRT–7417, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1989.

[8.2] Helmut Brandl, Rainer Johanni, and Martin Otter. A Very Efficient Algorithm for the Simulation of Robots and Similar Multibody–Systems Without Inversion of the Mass Matrix. In P. Kopacek, Inge Troch, and K. Desoyer, editors, *Theory of Robots*, pages 95–100. Pergamon Press, Oxford, United Kingdom, 1986.

[8.3] Dag Brück, Hilding Elmqvist, Hans Olsson, and Sven-Erik Mattsson. Dymola for Multi–Engineering Modeling and Simulation. In *Proceedings* 2^{nd} *Intl. Modelica Conference*, pages 55:1–8, Munich, Germany, 2002.

[8.4] Kathryn E. Brenan, Stephen L. Campbell, and Linda R. Petzold. *Numerical Solution of Initial–Value Problems in Differential–Algebraic Equations.* North–Holland, New York, 1989. 256p.

[8.5] François E. Cellier. *Continuous System Modeling.* Springer Verlag, New York, 1991. 755p.

[8.6] François E. Cellier. Inlining Step–size Controlled Fully Implicit Runge–Kutta Algorithms for the Semi–analytical and Semi–numerical Solution of Stiff ODEs and DAEs. In *Proceedings Vth Conference on Computer Simulation*, pages 259–262, Mexico City, Mexico, 2000.

[8.7] Jacques Denavit and Richard S. Hartenberg. A Kinematic Notation for Lower–Pair Mechanisms Based on Matrices. *ASME Journal of Applied Mechanics*, 22(2):215–221, 1955.

[8.8] Hilding Elmqvist, Martin Otter, and François E. Cellier. Inline Integration: A New Mixed Symbolic/Numeric Approach for Solving Differential–Algebraic Equation Systems. In *Proceedings European Simulation Multiconference*, pages xxiii–xxxiv, Prague, Czech Republic, 1995.

[8.9] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems.* PhD thesis, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.

[8.10] Hilding Elmqvist. *Dymola — Dynamic Modeling Language, User's Manual, Version 5.3.* DynaSim AB, Research Park Ideon, Lund, Sweden., 2004.

[8.11] Roy Featherstone. The Calculation of Robot Dynamics Using Articulated–Body Inertias. *Internat. Journal of Robotics Research*, 2:13–30, 1983.

[8.12] Claus Führer and Ben J. Leimkuhler. Numerical Solution of Differential–Algebraic Equations for Constrained Mechanical Motion. *Numerische Mathematik*, 59:55–69, 1991.

[8.13] C. William Gear. The Simulataneous Numerical Solution of Differential–Algebraic Equations. *IEEE Trans. Circuit Theory*, CT–18(1):89–95, 1971.

[8.14] Gary D. Hachtel, Robert K. Brayton, and Fred G. Gustavson. The Sparse Tableau Approach to Network Analysis and Design. *IEEE Trans. Circuit Theory*, CT–18(1):101–118, 1971.

[8.15] Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric Numerical Integration: Structure–Preserving Algorithms for Ordinary Differential Equations.* Springer Verlag, Berlin, 2002. 515p.

[8.16] Ernst Hairer and Gerhard Wanner. *Solving Ordinary Differential Equations II: Stiff and Differential–Algebraic Problems*, volume 14 of *Series in Computational Mathematics.* Springer–Verlag, Berlin, Germany, 2^{nd} edition, 1996. 632p.

[8.17] Russell C. Hibbeler. *Engineering Mechanics: Dynamics.* Prentice Hall, Upper Saddle River, New Jersey, 9^{th} edition, 2001. 688p.

[8.18] Chung-Wen Ho, Albert E. Ruehli, and Pierce A. Brennan. The Modified Nodal Approach to Network Analysis. In *Proceedings IEEE Intl. Symposium on Circuits and Systems*, pages 505–509, San Francisco, California, 1974.

[8.19] Michael G. Hollars, Rosenthal Dan E., and Michael A. Sherman. SD/Fast: User's Manual. Technical report, Symbolic Dynamics, Inc., Mountain View, California, 2001.

[8.20] Christian Lubich. Extrapolation Integrators for Constrained Multibody Systems. *Impact on Computer Science and Engineering*, 3:213–234, 1991.

[8.21] Johnson Y. S. Luh, Michael W. Walker, and Richard P. Paul. On–Line Computational Scheme for Mechanical Manipulators. *Trans. ASME, Journal of Dynamic Systems Measurement and Control*, 102:69–76, 1980.

[8.22] William L. Luyben. *Process Modeling, Simulation, and Control for Chemical Engineers.* McGraw–Hill, New York, 1973.

[8.23] Richard S. H. Mah. *Chemical Process Structures and Information Flows.* Butterworth Publishing, London, United Kingdom, 1990.

[8.24] H. Alan Mantooth and Martin Vlach. Beyond Spice With Saber and MAST. In *Proceedings IEEE Intl. Symposium on Circuits and Systems*, pages 77–80, San Diego, California, 1993.

[8.25] William J. McCalla. *Fundamentals of Computer–Aided Circuit Simulation.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1988. 175p.

[8.26] Laurence W. Nagel. SPICE2: A Computer Program to Simulate Semiconductor Circuits. Technical Report ERL–M 520, Electronic Research Laboratory, University of California Berkeley, Berkeley, California, 1975.

[8.27] Dan Negrut and Harris Brett. ADAMS: Theory in a Nutshell. Technical report, Dept. of Mechanical Engineering, University of Michigan, Ann Arbor, Michigan, 2001.

[8.28] Bernt Nilsson. *Structured Modelling of Chemical Processes — An Object–Oriented Approach.* PhD thesis, Lund Institute of Technology, Lund, Sweden, 1989.

[8.29] Martin Otter, Hilding Elmqvist, and François E. Cellier. Modeling of Multibody Systems with the Object–Oriented Modeling Language Dymola. *J. Nonlinear Dynamics*, 9(1):91–112, 1996.

[8.30] Martin Otter, Hilding Elmqvist, and Sven Erik Mattsson. The New Modelica Multibody Library. In *Proceedings* 3rd *International Modelica Conference*, pages 311–330, Linköping, Sweden, 2003.

[8.31] Martin Otter, Sven Erik Mattsson, Hans Olsson, and Hilding Elmqvist. Simulator for Large Scale, Multi–physics Systems. Technical Report Deliverable D27, Report for Task 2.7, German Aerospace Center, Oberpfaffenhofen, Germany, 2002.

[8.32] Martin Otter and Clemens Schlegel. Symbolic generation of efficient simulation codes for robots. In *Proceedings Second European Simulation Multi–Conference*, pages 119–122, Nice, France, 1988.

[8.33] Martin Otter. *Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter.* PhD thesis, Dept. of Mech. Engr., Ruhr–University Bochum, Germany, 1994.

[8.34] Linda R. Petzold. A Description of DASSL: A Differential/Algebraic Equation Solver. In R.S. Stepleman, editor, *Scientific Computing*, pages 65–68. North–Holland, Amsterdam, The Netherlands, 1983.

[8.35] Michael C. Schweisguth and François E. Cellier. A bond graph model of the bipolar junction transistor. In *Proceedings SCS Intl. Conference on Bond Graph Modeling and Simulation*, pages 344–349, San Francisco, California, 1999.

[8.36] Michael C. Schweisguth. Semiconductor Modeling with Bondgraphs. Master's thesis, Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, Arizona, 1997.

[8.37] George Stephanopoulos. *Chemical Process Control: An Introduction to Theory and Practice.* Prentice–Hall, Englewood Cliffs, N.J., 1984. 696p.

[8.38] Vicha Treeaporn. Efficient Simulation of Physical System Models Using Inlined Implicit Runge–Kutta Algorithms. Master's thesis, Dept. of Electrical & Computer Engineering, University of Arizona, Tucson, Arizona, 2005.

[8.39] Paul W. Tuinenga. *Spice: A Guide to Circuit Simulation and Analysis Using PSpice.* Prentice Hall, Englewood Cliffs, N.J., 3^{rd} edition, 1988. 288p.

[8.40] Jiri Vlach and Kishore Singhal. *Computer Methods for Circuit Analysis and Design.* Van Nostrand Reinhold, New York, 2^{nd} edition, 1994. 712p.

8.16 Bibliography

[B8.1] Braden A. Brooks and François E. Cellier. Modeling of a Distillation Column Using Bond Graphs. In *Proceedings SCS International*

Conference on Bond Graph Modeling, pages 315–320, San Diego, California, 1993. SCS Publishing.

[B8.2] Roy Featherstone. *Robot Dynamics Algorithms.* Kluwer, Boston, Mass, 1997. 228p.

[B8.3] Steve Gallun. *Solution Procedures for Nonideal Equilibrium Stage Processes at Steady and Unsteady State Described by Algebraic or Differential–Algebraic Equations.* PhD thesis, Texas A&M University, 1979.

[B8.4] Ernst Hairer, Christian Lubich, and Michel Roche. *The Numerical Solution of Differential–Algebraic Systems by Runge–Kutta Methods.* Springer–Verlag, Berlin, Germany, 1989. 139p.

[B8.5] Daryl Hild and François E. Cellier. Object–Oriented Electronic Circuit Modeling Using Dymola. In *Proceedings OOS'94, SCS Object Oriented Simulation Conference*, pages 68–75, Tempe, Arizona, 1994.

[B8.6] Charles D. Holland and Athanasios I. Liapis. *Computer Methods for Solving Dynamic Separation Problems.* McGraw–Hill, New York, 1983. 475p.

[B8.7] Asghar Husain. *Chemical Process Simulation.* John Wiley & Sons, New York, 1986. 376p.

[B8.8] William L. Luyben. *Practical Distillation Control.* Van Nostrand Reinhold, New York, 1992. 533p.

[B8.9] Parviz E. Nikravesh. *Computer–Aided Analysis of Mechanical Systems.* Prentice–Hall, Englewood Cliffs, N.J., 1988. 370p.

[B8.10] Richard P. Paul. *Robot Manipulators: Mathematics, Programming, and Control — The Computer Control of Robot Manipulators.* MIT Press, Cambridge, Mass., 1981. 279p.

[B8.11] Mark W. Spong and Mathukumalli Vidyasagar. *Robot Dynamics and Control.* John Wiley & Sons, New York, 1989. 336p.

[B8.12] Michael W. Walker and David E. Orin. Efficient Dynamic Computer Simulation of Robotic Mechanisms. *Journal of Dynamic Systems, Measurement and Control*, 104:205–211, 1982.

8.17 Homework Problems

[H8.1] Inlining BDF3

Given the electrical circuit shown in Fig.H8.1a.

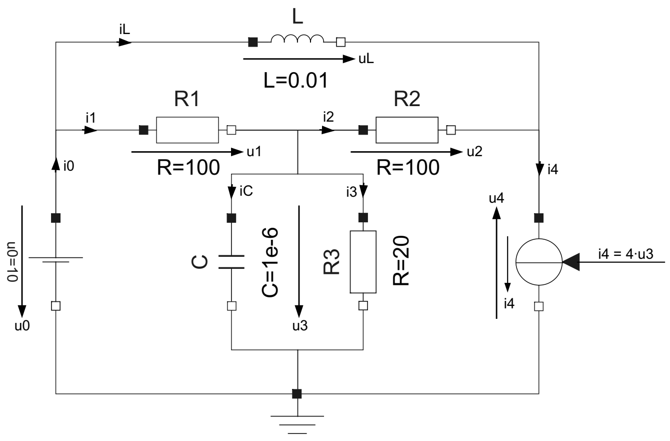

FIGURE H8.1a. Electrical circuit.

The circuit contains a constant voltage source, u_0, and a nonlinear (driven) current source, i_4, that depends on the voltage across the capacitor, C, and the resistor, R_3.

Write down the element equations for the seven circuit elements. Since the voltage u_3 is common to two circuit elements, these equations contain 13 rather than 14 unknowns. Add the voltage equations for the three meshes and the current equations for three of the four nodes. One current equation is redundant. Usually, the current equation for the ground node is therefore omitted. In this way, you end up with 13 equations in the 13 unknowns.

We wish to inline a fixed–step BDF3 algorithm, using order buildup during the startup phase. Draw the structure digraph of the inlined equation system, which now consists of 15 equations in 15 unknowns, and causalize it using the tearing method.

Simulate the ΔE system across 50 μsec using the inlined BDF3 algorithm with zero initial conditions on both the capacitor and the inductor. Choose a step size of $h = 0.5$ μsec. Use algebraic differentiation for the computation of the Hessian.

Plot the voltage u_3 and the current i_C on two separate subplots as functions of time.

[H8.2] Inlining Radau IIA

We wish to repeat Hw.[H8.1], this time inlining the 3^{rd}–order accurate Radau IIA algorithm. Draw the structure digraph of the inlined equation system, which now consists of 30 equations in 30 unknowns, and causalize it using the tearing method.

Simulate the ΔE system across 50 μsec using the inlined Radau IIA algorithm with zero initial conditions on both the capacitor and the inductor. Choose a step size of $h = 0.5$ μsec. Use algebraic differentiation for the computation of the Hessian.

Plot the voltage u_3 and the current i_C on two separate subplots as functions of time.

[H8.3] Step–size Control for Radau IIA

We wish to augment the solution to Hw.[H8.2] by adding a step–size control algorithm.

Use Eq.(8.105) as the embedding method for the purpose of error estimation, and use Fehlberg's step–size control algorithm, Eq.(3.89), for the computation of the next step size. Of course, the formula needs to be slightly modified, since it assumes the error estimate to be 5^{th}–order accurate, whereas in our algorithm, it is only 4^{th}–order accurate. Remember that the step size can never be modified two steps in a row.

Simulate the ΔE system across 50 μsec using the step–size controlled inlined Radau IIA algorithm with zero initial conditions on both the capacitor and the inductor. Count the number of Newton iterations. Multiply that number with the number of statements inside the loop. This should provide you with a decent estimate of the computational efficiency of the method.

Plot the voltage u_3 and the current i_C on two separate subplots as functions of time.

[H8.4] Inlining Lobatto IIIC

We wish to repeat Hw.[H8.2], this time inlining the 4^{th}–order accurate Lobatto IIIC algorithm. Draw the structure digraph of the inlined equation system, which now consists of 45 equations in 45 unknowns, and causalize it using the tearing method.

Simulate the ΔE system across 50 μsec using the inlined Lobatto IIIC algorithm with zero initial conditions on both the capacitor and the inductor. Choose a step size of $h = 0.5$ μsec. Use algebraic differentiation for the computation of the Hessian.

Plot the voltage u_3 and the current i_C on two separate subplots as functions of time.

[H8.5] Step–size Control for Lobatto IIIC

We wish to augment the solution to Hw.[H8.4] by adding a step–size control algorithm.

Use Eq.(8.110) as the embedding method for the purpose of error estimation, and use Fehlberg's step–size control algorithm, Eq.(3.89), for the computation of the next step size. Remember that the step size can never be modified two steps in a row.

Simulate the ΔE system across 50 μsec using the step–size controlled inlined Lobatto IIIC algorithm with zero initial conditions on both the capacitor and the inductor. Count the number of Newton iterations. Multiply that number with the number of statements inside the loop. This should provide you with a decent estimate of the computational efficiency of the method. MATLAB used to offer a better means of estimating the efficiency of a code by counting the number of floating point operations, using the built–in function *flops*. Unfortunately, this feature has been disabled in version 6 of MATLAB.

If you also solved Hw.[H8.3], you can compare the computational efficiency of the two algorithms for solving the given circuit problem against each other.

Plot the voltage u_3 and the current i_C on two separate subplots as functions of time.

[H8.6] Algebraic Differentiation

We wish to reproduce Fig.8.31 of this chapter. On purpose, we haven't shown you the details of how it has been derived. In particular, we didn't provide the symbolic equations for the computation of the Hessian by means of algebraic differentiation.

[H8.7] Stabilized BE Simulation of Overdetermined DAE System

We wish to reproduce Fig.8.33 of this chapter. On purpose, we haven't shown you the details of how it has been derived. In particular, we didn't provide you with a formula for when to end the Newton iteration. Since the linear system is now only solved in a least square sense, you can no longer test for $\|\mathcal{F}\|$ having decreased to a small value. The way we did it was to compute the norm of $\mathcal{F}$ and save that value between iterations. We then tested, whether the norm of $\mathcal{F}$ has no longer decreased significantly from one iteration to the next:

```
while $\text{abs}(\|\mathcal{F}^\ell\| - \|\mathcal{F}^{\ell-1}\|) < 1.0e-6$,
    perform iteration
end,
```

8.18 Projects

[P8.1] Inlining DIRK

There exists yet another interesting class of implicit stiffly stable Runge–Kutta algorithms that we haven't discussed in this chapter. These are called *diagonally implicit Runge–Kutta algorithms*, and are usually abbreviated as DIRK algorithms. One of the more fashionable among the DIRK algorithms is HW–SDIRK(3)4 [8.16] with the Butcher tableau:

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

HW–SDIRK(3)4 is a five–stage algorithm. DIRKs are much less compact than their IRK cousins, and therefore, allow proper embedding algorithms to exist within them. x represents a 3^{rd}–order accurate method, whereas $\hat{x}$ represents a 4^{th}–order accurate method.

DIRK methods are attractive alternatives to the IRK methods discussed in this chapter, since they can be implemented with one Newton iteration per stage, rather than with one Newton iteration across all stages.

Remember the dilemma that we were facing when we tried to inline parabolic PDEs. Inlining a BDF algorithm, we had to perform a Newton iteration in 25 tearing variables, whereas inlining the 3^{rd}–order accurate Radau IIA algorithm, we had to perform a Newton iteration in 100 tearing variables. Thus, Radau IIA would need to be able to use step sizes that are at least 16 times as large as those used by BDF3 in order to be competitive.

Inlining HW–SDIRK(3)4, we would expect to require five Newton iterations, each in 25 tearing variables. Thus, we would need to use only five times as large step sizes as those employed by BDF3, in order to be competitive.

Find the **F**–matrices of the two embedded methods, and perform Taylor–series expansions to verify that the two methods are indeed 3^{rd}–order and 4^{th}–order accurate, respectively. Compute the error coefficient of the error–controlled method.

Plot the stability domains as well as the damping plots of the two individual methods. Decide, which of the two estimates should be propagated to the next step.

Show how HW–SDIRK(3)4 can be inlined by means of the problem discussed in Hw.[H8.1].

Simulate the circuit using the step–size controlled inlined HW–SDIRK(3)4 algorithm.

8.19 Research

[R8.1] Inlining Parabolic PDEs

Develop suitable heuristic procedures for finding small sets of tearing variables for inlining parabolic PDEs in multiple space dimensions.

As we have discussed in Chapter 6 of this book, the simulation of parabolic PDEs converted to sets of ODEs by the MOL approach often requires internal Newton iterations due to either nonlinear boundary conditions or irregular domain boundaries. Hence inlining them might be quite attractive.

The numerical PDE literature is full of descriptions of sparse matrix algorithms for improving the efficiency of the simulation of such problems. Tearing can also be viewed as a sparse matrix technique, although it is applied in a symbolic form.

Compare the computational efficiency of the ΔE simulation after inlining with that of alternative ODE simulations without inlining.