For the implicit Euler method applied to the index one DAE (4.6.4), there exists an expansion of the form (4.6.5) where $\varepsilon_n^j = 0$ for $n \geq 1$. Therefore, the error in the extrapolate T_{jj} is $O(H^j)$ for all j [171]. For this reason, the expense of iterating the implicit Euler method to convergence may be justified when extrapolation of the values is employed and a high order of accuracy is desired, for difficult nonlinear systems.

We note that if $B(y)$ is nonsingular as in an index zero DAE (i.e., an implicit ODE), there exists an unperturbed asymptotic error expansion for both linearly implicit schemes.

Extrapolation based on the linearly implicit midpoint method has been experimented with by Deuflhard [95] and later by Hairer and Lubich [135] for use in solving stiff ODE's. A limited theoretical result is obtained in [135] which proves the existence of an perturbed h^2 expansion of the discretization error for this method when applied to a simple stiff system of the form,

$$\begin{aligned} y' &= f(y), \ y(0) = y_0 \\ \epsilon z' &= -z + g(y), \ z(0) = z_0. \end{aligned}$$

For the corresponding DAE ($\epsilon = 0$), the pertubation terms vanish.

It is important to note that extrapolation methods do not in general enforce the satisfaction of algebraic constraints. Specifically, consider the semi-explicit index one DAE (4.6.2) obtained when $\epsilon = 0$. Of course, the first column of the extrapolation tableau, generated by the implicit Euler method, satisfies the algebraic equations. However, the extrapolated values determined via (4.6.1) do not satisfy the algebraic constraints 'exactly' (i.e., to the round-off level of accuracy). In this case it is simple, although a bit expensive, to apply extrapolation to the y variables and to compute the z variables via a Newton iteration on the algebraic constraints.

The investigation of extrapolation methods for index two DAE's is just beginning. We have seen experimental evidence in Section 4.4 for semi-explicit index two systems which indicates that extrapolation based on the implicit Euler method may be very useful. However, a complete theory justifying the use of this method is not yet available. The existence of an unperturbed asymptotic h^2 expansion of the global error for semi-explicit index two DAE's of the form

$$\begin{aligned} y' &= f_0(y) + f_z(y)z \\ 0 &= g(y) \end{aligned}$$

is shown by Lubich [172] for an extension of Gragg's method [127]. In particular, the differential equations are discretized via an explicit midpoint rule, while the algebraic equations are discretized in an implicit way. This method would be appropriate for *nonstiff* index two systems. Finally, the extrapolation method of Bader and Deuflhard [13] (namely, a semi-implicit midpoint rule) can be extended in a similar way to this class of index two systems. Lubich shows that the resulting method also has an h^2 expansion.

Chapter 5

Software for DAE's

5.1 Introduction

The most compelling reason for developing methods and analysis is to solve problems from applications. But first the methods must be implemented in codes which are efficient, robust, easy to use, and well documented. In this chapter we will discuss some of the software issues which are important in developing and using a code for solving DAE's.

Several classes of methods emerge from the convergence analysis of the previous chapters as potential candidates for methods on which to base a variable stepsize variable order general purpose code for index one DAE's. Within the class of multistep methods, the BDF methods suffer no order reduction for index one systems. One-step methods such as the L-stable singly implicit methods of Burrage and Butcher [38,39,45] and the extrapolation methods of Deuflhard et al.[96,97] which are based on the semi-implicit Euler method also appear to be promising for this purpose. The most widely used production code for DAE's at this time is the code DASSL of Petzold [200], which is based on BDF methods. We will concentrate our attention in this chapter on the algorithms and issues which are important in the implementation of DASSL. However, it is important to note that many of these issues arise in the implementation of any numerical ODE method for the solution of DAE's.

DASSL is designed to be used for the solution of DAE's of index zero and one. The convergence analysis in Chapter 3 establishes that the BDF methods achieve the same order of convergence for this class of DAE's as they do for ODE's. Thus there is some theory providing the foundation for the software. However, there are issues other than the order of convergence of the methods which are important in successfully implementing and using the methods. These issues are more complex for DAE's than for ODE's. For example, the initial conditions for a DAE must be chosen to be consistent, the linear system which must be solved on each time step is ill conditioned for small stepsizes, error estimates used in the selection of stepsizes are sensitive to inconsistencies in the initial conditions and sharp changes in the solution,

and the solution methods are dependent on a more accurate approximation to the iteration matrix than is generally needed for ODE's.

Since it is possible to write a problem in the form $F(t, y, y') = 0$ which has index larger than one, it is important to be aware that a failure of the code could be due to a higher index formulation. In this case, the user of the code needs to know the possibilities of modifying the code to solve the higher index problem, or of rewriting the problem in a lower index form, and the advantages or disadvantages of these alternatives. In general, the diagnostics in DAE codes are not as well developed as those in nonstiff ODE codes. It is sometimes difficult for the code to distinguish, for example, between a failure due to inconsistent initial conditions and one due to a higher index problem formulation. Thus it is useful for anyone who plans to make more than a casual use of this type of code to be familiar with some of the details of how the code works, how it would likely behave in the event of different types of failure, and what the alternatives are for obtaining a solution in the event of code failure. On the other hand, it is our experience that the vast majority of DAE problems from applications are solved successfully with DASSL, often by users with little previous experience in solving DAE's and ODE's.

In this chapter we will explore software development issues for DAE's. In Section 5.2 we describe the basic algorithms and strategies used in DASSL, including the predictor and corrector formulas, solution of the nonlinear system, stepsize and order selection and error control. In Section 5.3 we focus on using DASSL, including how to set up a problem, how to obtain consistent initial conditions, and how to interpret failures. We also describe codes based on DASSL which are written or are currently under development with extended capabilities. It is important to remember that DASSL and other general purpose DAE codes of which we are aware are designed for solving index zero and index one DAE systems. In general, these codes will fail for higher index systems. But we have seen in Chapter 3 and Chapter 4 that the BDF methods and some other methods converge for some important classes of higher index systems, most notably for semi-explicit index two systems. In Section 5.4 we discuss the solution of higher index systems, either by rewriting the system in a lower index form which has the same analytic solution, or by solving it directly using a code like DASSL, where the error control and other strategies have been modified appropriately.

DASSL is designed for solving initial value problems of the implicit form $F(t, y, y') = 0$ which are index zero or one. At this time we are aware of several other general purpose codes for solving related problems which have been used extensively in applications. The code LSODI, developed by Hindmarsh and Painter [144], is similar to DASSL in that it is based on BDF methods. This code is written for *linearly implicit* DAE's of the form $A(t, y)y' = f(t, y)$. The user must supply a subroutine for evaluating the matrix A times a vector. LSODI differs from DASSL in the way that it implements the BDF formulas, in the way that it stores and interpolates the past solution values needed by the

BDF formulas, and most notably in the stepsize and order selection and error control strategies. Our experience with the two codes is that on many problems they are quite similar in accuracy and efficiency. For DAE's and ODE's with eigenvalues close to the imaginary axis in the complex plane, DASSL has a more robust order selection strategy. The SPRINT code, developed by Berzins, Dew and Furzeland [16,18,19,20], also employs BDF methods for the solution of linearly implicit DAE's. This code uses the filtered error estimate described in Section 5.4.2, and has been used in a wide variety of method of lines applications. The FACSIMILE code, developed by Curtis [84], uses BDF methods to solve semi-explicit index one systems. Another code of which we are aware is the code LIMEX of Deuflhard et al.[97]. This code is based on extrapolation of the semi-implicit Euler method. The theory for this method is limited to linearly implicit DAE's. We do not have any experience using this code, but Maas and Warnatz [177] report good success in solving problems in combustion modeling. This code attempts to diagnose failures in the first step, but again the technique used cannot distinguish between systems of index two (or higher) and inconsistent initial conditions for an index one problem. One-step methods, such as the extrapolation method, have an inherent advantage over multistep methods for problems with frequent discontinuities simply because they can restart after a discontinuity with a higher order method, whereas the usual implementations of multistep methods restart with a first order method.

5.2 Algorithms and Strategies in DASSL

5.2.1 Basic Formulas

DASSL is a code for solving index zero and one systems of differential/algebraic equations of the form

$$\begin{aligned} F(t, y, y') &= 0 \\ y(t_0) &= y_0 \\ y'(t_0) &= y'_0, \end{aligned} \tag{5.2.1}$$

where F, y, and y' are N-dimensional vectors. The basic idea for solving DAE systems using numerical ODE methods, originating with Gear [113], is to replace the derivative in (5.2.1) by a difference approximation, and then to solve the resulting system for the solution at the current time t_{n+1} using Newton's method. For example, replacing the derivative in (5.2.1) by the first order backward difference, we obtain the implicit Euler formula

$$F\left(t_{n+1}, y_{n+1}, \frac{y_{n+1} - y_n}{h_{n+1}}\right) = 0, \tag{5.2.2}$$

where $h_{n+1} = t_{n+1} - t_n$. This nonlinear system is then usually solved using some variant of Newton's method. We will discuss the solution of the system

(5.2.2) in the next subsection. The algorithms used in DASSL are an extension of this basic idea. Instead of always using the first order formula (5.2.2), DASSL approximates the derivative using the kth order backward differentiation formula (BDF), where k ranges from one to five. On every step it chooses the order k and stepsize h_{n+1}, based on the behavior of the solution.

DASSL uses a variable stepsize variable order *fixed leading coefficient* [147] implementation of BDF formulas to advance the solution from one time step to the next. The fixed leading coefficient implementation is one way of extending the fixed stepsize BDF methods to variable stepsizes. There are three main approaches to extending fixed stepsize multistep methods to variable stepsize. These formulations are called fixed coefficient, variable coefficient and fixed leading coefficient. The fixed coefficient methods have the property that they can be implemented very efficiently for smooth problems, but suffer from inefficiency, or possible instability, for problems which require frequent stepsize adjustments. The variable coefficient methods are the most stable implementation, but have the disadvantage that they tend to require more evaluations of the Jacobian matrix in intervals when the stepsize is changing, and hence are usually considered to be less efficient than a fixed coefficient implementation for most problems. The fixed leading coefficient formulation is a compromise between the fixed coefficient and variable coefficient approaches, offering somewhat less stability, along with usually fewer Jacobian evaluations, than the variable coefficient formulation. In contrast to DASSL, the code LSODI [144] is a fixed coefficient implementation of the BDF formulas. The relative advantages and disadvantages of the various formulations are explained in much greater detail in Jackson and Sacks-Davis [147]. It is possible that with a stepsize selection strategy which is different from the ones usually implemented in BDF codes, the variable coefficient implementation could be the most efficient [230]. However, this idea has not been extensively tested. It is still an open question for stiff ODE's and for DAE's which formulation is best for a general purpose code.

Now we will describe the basic formulas used in DASSL. Suppose we have approximations y_{n-i} to the true solution $y(t_{n-i})$ for $i = 0, 1, \ldots, k$, where k is the order of the BDF method that we are currently planning to use. We would like to find an approximation to the solution at time t_{n+1}. First, an initial guess for the solution and its derivative at t_{n+1} is formed by evaluating the *predictor polynomial* and the derivative of the predictor polynomial at t_{n+1}. The predictor polynomial $\omega^P_{n+1}(t)$ is the polynomial which interpolates y_{n-i} at the last $k+1$ times,

$$\omega^P_{n+1}(t_{n-i}) = y_{n-i}, \qquad i = 0, 1, \ldots, k.$$

The predicted values for y and y' at t_{n+1} are obtained by evaluating $\omega^P_{n+1}(t)$

and $\omega'^P_{n+1}(t)$ at t_{n+1}, so that

$$\begin{aligned} y^{(0)}_{n+1} &= \omega^P_{n+1}(t_{n+1}) \\ y'^{(0)}_{n+1} &= \omega'^P_{n+1}(t_{n+1}). \end{aligned}$$

The approximation y_{n+1} to the solution at t_{n+1} which is finally accepted by DASSL is the solution to the *corrector formula.* The formula used is the fixed leading coefficient form of the k^{th} order BDF method. The solution to the corrector formula is the vector y_{n+1} such that the corrector polynomial $\omega^C_{n+1}(t)$ and its derivative satisfy the DAE at t_{n+1}, and the corrector polynomial interpolates the predictor polynomial at k equally spaced points behind t_{n+1},

$$\begin{aligned} \omega^C_{n+1}(t_{n+1}) &= y_{n+1} \\ \omega^C_{n+1}(t_{n+1} - ih_{n+1}) &= \omega^P_{n+1}(t_{n+1} - ih_{n+1}),\ 1 \le i \le k, \\ F(t_{n+1}, \omega^C_{n+1}(t_{n+1}), \omega'^C_{n+1}(t_{n+1})) &= 0. \end{aligned} \tag{5.2.3}$$

In the next subsection we describe how the solution y_{n+1}, implicitly defined by conditions (5.2.3), is determined by solving a system of nonlinear equations by Newton's method.

The values of the predictor $y^{(0)}_{n+1}$, $y'^{(0)}_{n+1}$ and the corrector y_{n+1} at t_{n+1} are defined in terms of polynomials which interpolate the solution at previous time steps. Following the ideas of Krogh [155] and Shampine and Gordon [225], these polynomials are represented in DASSL in terms of modified divided differences of y. More precisely, the quantities which are updated from step to step are given by

$$\begin{aligned} \psi_i(n+1) &= h_{n+1} + h_n + \cdots + h_{n+2-i} = t_{n+1} - t_{n+1-i}, \quad i \ge 1 \\ \alpha_i(n+1) &= h_{n+1}/\psi_i(n+1), \quad i \ge 1 \\ \beta_1(n+1) &= 1 \\ \beta_i(n+1) &= \frac{\psi_1(n+1)\psi_2(n+1)\cdots\psi_{i-1}(n+1)}{\psi_1(n)\psi_2(n)\cdots\psi_{i-1}(n)}, \quad i > 1 \\ \phi_1(n) &= y_n \\ \phi_i(n) &= \psi_1(n)\psi_2(n)\ldots\psi_{i-1}(n)[y_n, y_{n-1}, \ldots, y_{n-i+1}], \quad i > 1 \\ \phi^*_i(n) &= \beta_i(n+1)\phi_i(n), \quad i \ge 1 \\ \sigma_1(n+1) &= 1 \\ \sigma_i(n+1) &= \frac{h^i_{n+1}(i-1)!}{\psi_1(n+1)\psi_2(n+1)\cdots\psi_i(n+1)}, \quad i > 1 \\ \gamma_1(n+1) &= 0 \\ \gamma_i(n+1) &= \gamma_{i-1}(n+1) + \alpha_{i-1}(n+1)/h_{n+1}, \quad i > 1 \\ \alpha_s &= -\sum_{j=1}^{k} \frac{1}{j} \end{aligned} \tag{5.2.4}$$

$$\alpha^0(n+1) = -\sum_{j=1}^{k} \alpha_i(n+1).$$

The divided differences are defined by the recurrence

$$\begin{aligned} [y_n] &= y_n \\ [y_n, y_{n-1}, \ldots, y_{n-k}] &= \frac{[y_n, y_{n-1}, \ldots, y_{n-k+1}] - [y_{n-1}, y_{n-2}, \ldots, y_{n-k}]}{t_n - t_{n-k}}. \end{aligned}$$

The predictor polynomial is given in terms of the divided differences by

$$\begin{aligned} \omega_{n+1}^P(t) = {} & y_n + (t-t_n)[y_n, y_{n-1}] + (t-t_n)(t-t_{n-1})[y_n, y_{n-1}, y_{n-2}] \\ & + \cdots + (t-t_n)(t-t_{n-1})\cdots(t-t_{n-k+1})[y_n, y_{n-1}, \ldots, y_{n-k}]. \end{aligned} \tag{5.2.5}$$

Evaluating ω_{n+1}^P at t_{n+1}, and rewriting (5.2.5) in terms of the notation in (5.2.4), we obtain the predictor formula

$$y_{n+1}^{(0)} = \sum_{i=1}^{k+1} \phi_i^*(n).$$

Differentiating (5.2.5) and evaluating $\omega_{n+1}^{\prime P}$ at t_{n+1}, we find after some manipulation that

$$y_{n+1}^{\prime(0)} = \sum_{i=1}^{k+1} \gamma_i(n+1)\phi_i^*(n),$$

where $\gamma_i(n+1)$ satisfies the recurrence relation in (5.2.4).

To find the corrector formula, we note as in Jackson and Sacks-Davis [147] that (5.2.3) implies

$$\omega_{n+1}^C(t) - \omega_{n+1}^P(t) = b(t)(y_{n+1} - y_{n+1}^{(0)}), \tag{5.2.6}$$

where

$$\begin{aligned} b(t_{n+1} - ih_{n+1}) &= 0, \qquad i = 1, 2, \ldots, k \\ b(t_{n+1}) &= 1. \end{aligned}$$

Differentiating (5.2.6) and evaluating at t_{n+1} gives

$$\alpha_s(y_{n+1} - y_{n+1}^{(0)}) + h_{n+1}(y'_{n+1} - y_{n+1}^{\prime(0)}) = 0, \tag{5.2.7}$$

where the fixed leading coefficient α_s is defined in (5.2.4). Solving (5.2.7) for y'_{n+1}, we find that the corrector iteration must solve

$$F\left(t_{n+1}, y_{n+1}, y_{n+1}^{\prime(0)} - \frac{\alpha_s}{h_{n+1}}(y_{n+1} - y_{n+1}^{(0)})\right) = 0 \tag{5.2.8}$$

for y_{n+1}.

Finally, DASSL employs an interpolant to compute the solution between mesh points. This capability is needed for output purposes, when the user requires a value of the solution at points which are not necessarily at the timesteps chosen by the code. The interpolation is a necessity in the root finding version of DASSL which we will describe later. Starting from the end of a successful step, the interpolant which DASSL uses between t_n and t_{n+1} is given by

$$\begin{aligned} \omega_{n+1}^I(t) = {} & y_{n+1} + (t-t_{n+1})[y_{n+1}, y_n] \\ & + (t-t_{n+1})(t-t_n)[y_{n+1}, y_n, y_{n-1}] + \cdots \\ & + (t-t_{n+1})(t-t_n)\cdots(t-t_{n-k+2})[y_{n+1}, y_n, \ldots, y_{n-k+1}] \end{aligned} \tag{5.2.9}$$

where k is the order of the method which was used to advance the solution from t_n to t_{n+1}. Note that the interpolant is continuous, but that it has a discontinuous derivative at t_n and t_{n+1}. It is possible to define a C^1 interpolant [15,238], which leads to a more robust code for root finding. However, at this time the C^1 interpolant is not implemented in DASSL. It will likely appear in later versions.

It is important to note that algebraic relations (constraints) for DAE's are *not* automatically satisfied at interpolated points. If this is important for the application, for example if DASSL is to be restarted at the interpolated point, then in the present version the user may need to recalculate some of the solution components at the interpolated point so that they are consistent with the constraints.

5.2.2 Nonlinear System Solution

The corrector equation (5.2.8) must be solved for y_{n+1} at each time step. Here, we describe how this is done in DASSL.

To simplify notation, rewrite the corrector equation (5.2.8) as

$$F(t, y, \alpha y + \beta) = 0 \tag{5.2.10}$$

where $\alpha = -\alpha_s/h_{n+1}$ and $\beta = y_{n+1}^{\prime(0)} - \alpha y_{n+1}^{(0)}$. In (5.2.10), all variables are evaluated at t_{n+1}, α is a constant which changes whenever the stepsize or order changes, and β is a vector which remains constant while we are solving the corrector equation.

The corrector equation is solved using a modified Newton iteration, given by

$$y^{(m+1)} = y^{(m)} - cG^{-1}F(t, y^{(m)}, \alpha y^{(m)} + \beta), \tag{5.2.11}$$

where $y^{(0)}$ is given by (5.2.5), c is a scalar constant which will be defined shortly, and G is the iteration matrix,

$$G = \alpha\frac{\partial F}{\partial y'} + \frac{\partial F}{\partial y}.$$

The partial derivatives are evaluated at the predicted y and y'.

To solve (5.2.11), the matrix G is factored into a product of an upper and lower triangular matrix, $G = LU$, and (5.2.11) is then solved by

$$\begin{aligned} Ls^{(m)} &= r^{(m)} \\ U\delta^{(m)} &= s^{(m)}, \end{aligned} \tag{5.2.12}$$

where $\delta^{(m)} = y^{(m+1)} - y^{(m)}$, and $r^{(m)} = -cF(t, y^{(m)}, \alpha y^{(m)} + \beta)$. In DASSL, the matrix G may be dense or have a banded structure. The factorization of G and the solution of the systems in (5.2.12) are performed by routines in the LINPACK [100] software package.

For many applications, and especially when the system to be solved is large, the costs of computing and factoring G dominate the cost of the integration. Often the matrices $\partial F/\partial y'$ and $\partial F/\partial y$ change very little over the span of several time steps. However, the constant α changes whenever the stepsize or order of the method being used changes. If the derivative matrices and the constant α have not changed very much since the last iteration matrix was computed, then it is desirable to use the old iteration matrix in (5.2.11), instead of reevaluating the matrix on every step. Thus, if $\hat{G}$ is the iteration matrix that we have saved from some previous time step, $\hat{G} = \hat{\alpha}\partial F/\partial y' + \partial F/\partial y$, where $\hat{\alpha}$ depends on the stepsize and order at some past time step when $\hat{G}$ was last computed, then the iteration we actually use is

$$y^{(m+1)} = y^{(m)} - c\hat{G}^{-1}F(t, y^{(m)}, \alpha y^{(m)} + \beta). \tag{5.2.13}$$

As long as $\hat{G}$ is close enough to G, (5.2.13) will converge at an adequate rate, and we will be able to solve the corrector equation.

The constant c is chosen to speed up the rate of convergence of the iteration when $\alpha \neq \hat{\alpha}$. The idea of using this type of acceleration was introduced in [40,156]. Choosing $c \equiv 1$ corresponds to the usual modified Newton iteration for solving (5.2.10) for y, and choosing $c \equiv \alpha/\hat{\alpha}$ corresponds to the modified Newton iteration for solving $F(t, \hat{\alpha}^{-1}(y' - \beta), y') = 0$, (where $y' = \hat{\alpha}y + \beta$) for y'. To find the optimal value of c, note that the iteration (5.2.13) converges, for a sufficiently good initial guess, when the spectral radius of

$$B = (I - c\hat{G}^{-1}G)$$

is less than one. When (5.2.13) is used to solve the linear ODE, $y' - Ay = 0$, we find that if λ is an eigenvalue of A, then $\bar{\lambda} = 1 - c(\alpha - \lambda)/(\hat{\alpha} - \lambda)$ is an eigenvalue of B. For a general DAE system where the eigenvalues of G are unknown, we take c to minimize the maximum of this expression over all λ in the left half of the complex plane. This minimization gives

$$c = \frac{2\hat{\alpha}}{\alpha + \hat{\alpha}}.$$

When c is chosen in this way, the corresponding $\bar{\lambda}$ is

$$\bar{\lambda} = \frac{\hat{\alpha} - \alpha}{\alpha + \hat{\alpha}}. \tag{5.2.14}$$

The expression (5.2.14) is useful in deciding when to compute a new iteration matrix. Assuming that $\bar{\lambda}$ is an approximation to the rate of convergence of (5.2.13) when $\alpha \neq \hat{\alpha}$, and that we are willing to tolerate a rate of convergence of μ (in DASSL, $\mu = 0.25$), then a new iteration matrix is computed before attempting the iteration whenever the stepsize and/or order changes sufficiently so that $|\bar{\lambda}| > \mu$. Of course, a new iteration matrix is also computed whenever the corrector fails to converge after four iterations.

A decision which is important to the reliability and efficiency of a code like DASSL is when to terminate the corrector iteration (5.2.11). It is well-known that

$$||y^* - y^{(m+1)}|| \leq \frac{\rho}{1-\rho}||y^{(m+1)} - y^{(m)}||$$

where y^* is the solution to the corrector equation and ρ is an estimate of the rate of convergence of the iteration. Following Shampine [221], the iteration is continued until

$$\frac{\rho}{1-\rho}||y^{(m+1)} - y^{(m)}|| < 0.33 \tag{5.2.15}$$

so that the iteration error $||y^* - y^{(m+1)}||$ will be sufficiently small. While condition (5.2.15) appears at first to be independent of any user requested error tolerances, this information is buried in the norm. In particular, the default norm used in DASSL is a weighted root mean square norm, where the weights depend on the relative and absolute error tolerances and on the values of y at the beginning of the step. The constant 0.33 was chosen so that the errors due to terminating the corrector iteration would not adversely affect the integration error estimates. More precisely, if the weights are all equal and proportional to a desired integration error tolerance ϵ, then condition (5.2.15) may be interpreted as restricting the corrector iteration error to be less than $.33\epsilon$. A maximum of four iterations is permitted. Whenever $\rho > 0.9$, the corrector iteration is considered to have failed, and the iteration matrix is reevaluated if it isn't current.

It is possible for the user to define a personalized norm by substituting a routine for computing the norm. This option is important in some applications, for example in the method of lines solution of partial differential equations.

The rate of convergence is estimated whenever two or more corrector iterations have been taken by

$$\rho = \left(\frac{||y^{(m+1)} - y^{(m)}||}{||y^{(1)} - y^{(0)}||}\right)^{1/m}.$$

If the difference between the predictor and the first correction is very small (relative to roundoff error in y), the iteration is terminated, unless α has

changed or a new iteration matrix has just been formed. In these last two cases, a second corrector iteration is forced and a new estimate of ρ will be computed. If the iteration is terminated after only one corrector iteration, ρ is not recomputed. This last exception is needed because the rate may be polluted by roundoff error, and hence be misleading. On other steps, the rate of convergence is taken to be the last rate computed.

If the corrector iteration fails to converge, the iteration matrix is reevaluated and the iteration is attempted again. With a new iteration matrix, if the corrector again fails to converge, the stepsize is reduced by a factor of one quarter. After ten consecutive failures, or if the stepsize becomes so small that $t+h$ is equal to t in computer arithmetic, DASSL returns to the main program with an error flag.

The matrix G is either computed by finite differences, or supplied directly by the user. In finite differencing, if G is banded, the columns of G having nonzero elements in different rows are approximated simultaneously, using an idea of Curtis et al.[85]. The j^{th} column of G is approximated by

$$\frac{F(t, y+\delta_j e_j, \alpha(y+\delta_j e_j)+\beta) - F(t, y, \alpha y+\beta)}{\delta_j}, \tag{5.2.16}$$

where e_j is the j^{th} unit vector, and δ_j is the increment. This formulation of the differencing uses fewer function evaluations than differencing $\partial F/\partial y$ and $\partial F/\partial y'$ separately, and then adding them together to form $G = \alpha\partial F/\partial y' + \partial F/\partial y$.

The differencing procedure can be sensitive to roundoff error. The main source of difficulty in computing the iteration matrix G by finite differencing is the choice of the increment δ_j. The choice

$$\delta_j = \text{sign}(hy'_j)\max(|y_j|, |hy'_j|, \text{WT}_j)\sqrt{u}, \tag{5.2.17}$$

where $\text{WT}_j = \text{RTOL}|y_j| + \text{ATOL}$ is the weight vector used in the norm, is used in DASSL. RTOL and ATOL are the relative and absolute error tolerances specified by the user, and u is the unit roundoff error of the computer. Normally, when $|y_j|$ is not very close to zero, about half of the digits of y will be perturbed by the increment determined by (5.2.17). If $|y_j|$ is near zero, it is quite possible that a nearby value of y_j is not so small, and since $y_j(t+h) \approx y_j + hy'_j$, $|hy'_j|$ was included in the maximum in (5.2.17) to prevent a near zero perturbation from being selected. In the event that $|y_j|$ and $|hy'_j|$ are both near zero, WT_j is included in the maximum because the user has told us implicitly by setting the error tolerances that it is the smallest number which is relevant with respect to y_j. It is important to note that the sign of the increment will be negative if the solution is decreasing. This choice can occasionally be a source of difficulty for problems where F is undefined for $y < 0$, or not differentiable at $y = 0$.

5.2.3 Stepsize and Order Selection Strategies

An important feature of DASSL is that it changes the stepsize and the order of the method to solve problems more efficiently and reliably. In this subsection we describe the criterion used to decide when to accept a step, and the strategies for selecting the stepsize and order for the next step.

One of the most important questions concerning reliability of a DAE code is how it decides to accept or reject a step. There are two sources of errors that we are concerned with. The first is the local truncation error of the method. It is the amount by which the solution to the DAE fails to satisfy the BDF formula. The fixed leading coefficient form of the BDF approximates hy' with the formula

$$h_{n+1}y'_{n+1} = \sum_{i=2}^{k+1} \alpha_{i-1}(n+1)\phi_i(n+1) - (\alpha_s - \alpha^0(n+1))\phi_{k+2}(n+1),$$

where the notation is defined in equation (5.2.4). The true solution $y(t)$ satisfies

$$\begin{aligned} h_{n+1}y'(t_{n+1}) &= \sum_{i=2}^{k+1} \alpha_{i-1}(n+1)\tilde{\phi}_i(n+1) \\ &\quad -(\alpha_s - \alpha^0(n+1))\tilde{\phi}_{k+2}(n+1) + \tau_{n+1}, \end{aligned}$$

where τ_{n+1} is the local truncation error, given by

$$\begin{aligned} \tau_{n+1} &= \left(\alpha_{k+1}(n+1) + \alpha_s - \alpha^0(n+1)\right)\tilde{\phi}_{k+2}(n+1) \\ &\quad + \sum_{i=k+3}^{\infty} \alpha_{i-1}(n+1)\tilde{\phi}_i(n+1), \end{aligned} \tag{5.2.18}$$

and $\tilde{\phi}_i(n+1)$ are the scaled divided differences (i.e, the ϕ_i) defined in (5.2.4), evaluated at the true solution $y(t)$. DASSL estimates the principal term of the local truncation error, which is the first term in (5.2.18).

The divided difference $\tilde{\phi}_{k+2}(n+1)$ in (5.2.18) is approximated by $\phi_{k+2}(n+1)$, and $\phi_{k+2}(n+1)$ is the same as the difference between the predictor and the corrector. To see this, let $\omega^*_{n+1}(t)$ be the unique polynomial that interpolates y_{n+1-i}, $i = 0, 1, \ldots, k+1$. Then recalling that the predictor interpolates y_{n-i}, $i = 0, 1, \ldots, k$, it follows directly from the equations for the interpolating polynomials that $\omega^*_{n+1}(t) - \omega^P_{n+1}(t) = \phi_{k+2}(n+1)$ at $t = t_{n+1}$. The desired result follows immediately since $\omega^*_{n+1}(t_{n+1}) = y_{n+1}$ and $\omega^P_{n+1}(t_{n+1}) = y^{(0)}_{n+1}$ by definition.

This estimate of $\tilde{\phi}_{k+2}(n+1)$ is asymptotically correct in the case of constant stepsize or under slightly more general conditions [115,147]. The stepsize and order selection strategies in DASSL tend to favor sequences of constant stepsize and order, and the estimate has worked quite well in practice. Thus, the principal term of the local truncation error is estimated by

$$\left(\alpha_{k+1}(n+1) + \alpha_s - \alpha^0(n+1)\right)\|y_{n+1} - y^{(0)}_{n+1}\|. \tag{5.2.19}$$

The other source of error that is important is the error in interpolating to find the solution between mesh points for output purposes. In DASSL the solution at a point t^* between mesh points, $t_n \le t^* \le t_{n+1}$, is found by evaluating the polynomial of degree k (where k is the order of the method which was used to find y_{n+1}) which interpolates y at the last $k+1$ points $y_{n+1}, y_n, \ldots, y_{n-k+1}$, at t^*. The error in this interpolated value is composed of two parts. The first part is the *interpolation error*, due to interpolating the true solution $y(t)$ at $t_{n+1}, t_n, \ldots, t_{n-k+1}$ by a polynomial. The principal term of the interpolation error is bounded by

$$\alpha_{k+1}(n+1)||\phi_{k+2}(n+1)||. \tag{5.2.20}$$

The second part of the error is due to the fact that we are interpolating the computed solution instead of the true solution. This error is proportional to the errors in the solution at the mesh points. We are already trying to control these errors in (5.2.19), so we do not approximate them here.

Taking (5.2.19) and (5.2.20) together, at every step we require that

$$\text{ERR} = M||y_{n+1} - y_{n+1}^{(0)}|| \le 1.0, \tag{5.2.21}$$

where

$$M = \max\left(\alpha_{k+1}(n+1), |\alpha_{k+1}(n+1) + \alpha_s - \alpha^0(n+1)|\right).$$

If this condition is not satisfied, then the step is rejected. Note that condition (5.2.21) requires that the estimated integration error be less than the requested integration error tolerances, as reflected in the weighted norm of the predictor-corrector difference.

Whether or not a step is rejected, the code must decide which order method is to be used on the next step. Since the predictor (5.2.5) and the corrector (5.2.7) are of the same order, the leading order terms in their corresponding Taylor series expansions are identical aside from the error constants. DASSL estimates the leading order term in the remainder of the Taylor series expansion, independent of the error constants, at orders $k-2$, $k-1$, and k. If the last $k+1$ steps have been taken at constant stepsize and order, then the leading error term for order $k+1$ is also estimated. These estimates are

$$\begin{aligned}
\text{TERKM2} &= ||(k-1)\sigma_{k-1}(n+1)\phi_k(n+1)|| \approx ||h^{k-1}y^{(k-1)}|| \\
\text{TERKM1} &= ||k\sigma_k(n+1)\phi_{k+1}(n+1)|| \approx ||h^k y^{(k)}|| \\
\text{TERK} &= ||(k+1)\sigma_{k+1}(n+1)\phi_{k+2}(n+1)|| \approx ||h^{k+1}y^{(k+1)}|| \\
\text{TERKP1} &= ||(k+2)\sigma_{k+2}(n+1)\phi_{k+3}(n+1)|| \approx ||h^{k+2}y^{(k+2)}||.
\end{aligned} \tag{5.2.22}$$

Since the higher order estimate TERKP1 is formed only after $k+1$ steps at constant stepsize, $\sigma_{k+2}(n+1) = 1/(k+2)$. Note that these estimates are scaled to be independent of the error constants for each order. The strategies for raising or lowering the order are nearly identical with those described in Shampine and Gordon [225]. Briefly, there is an initial phase where the order and stepsize are increased on every step, and after that the order is raised or lowered depending on whether TERKM2, TERKM1, TERK and TERKP1 form an increasing or decreasing sequence. The philosophy behind this type of order selection strategy is that the Taylor series expansion is behaving as expected for higher orders only if the magnitudes of successive higher order terms form a decreasing sequence. Otherwise, it is safer to use a lower order method.

It is interesting to note that the order selection strategy in DASSL differs from that used in the LSODE family of codes [144] in several important aspects. In LSODE and its variants, error estimates are formed for the order $k-1$, k and $k+1$ methods, as if the last $k+1$ steps were taken at constant stepsize. The order is chosen which gives the largest stepsize based on these estimates. Thus, the order is chosen based solely on efficiency considerations. In DASSL, the terms $||h^{k-1}y^{(k-1)}||$, $||h^k y^k||$, etc. in the Taylor series expansion, rather than the error estimates for those orders, are compared. Several successive terms are examined, and the order is chosen based on maintaining a decreasing sequence. While the terms that DASSL compares are related to the error estimates, they are not the same.

The advantage of the order selection strategy which is implemented in DASSL is that it effectively solves a problem that variable order BDF codes can have because the higher order BDF methods are unstable for some ODE's. Consider the model problem $y' = Ay$, where A has eigenvalues with large imaginary parts but small negative real parts. The higher order BDF methods are unstable for this type of problem unless the stepsize is chosen to be quite small. If these small stepsizes are not required for the resolution of the solution, then the BDF code can take larger stepsizes with the more stable, lower order formulas than with the higher order formulas. The presence of instability is evidenced by rapid oscillations, or more simply by $||h^p y_n^{(p)}|| > ||h^q y_n^{(q)}||$ for $p > q$ [231]. When DASSL is solving a problem of this type and the higher order formulas begin to have a problem with instability, the estimates TERKM2, TERKM1, TERK, and TERKP1 fail to form a decreasing sequence. Then, the order is lowered until these terms (recomputed for the new order) are again decreasing. Thus the code is forced to use a stable, lower order method, thereby allowing larger stepsizes. The same sort of phenomena occurs with the error estimates in LSODE, but in DASSL testing for monotonicity and comparing the terms in (5.2.22) rather than the error estimates favors the lower order methods and force a lowering of the order sooner. The strategy used in the LSODE codes sometimes fails to lower the order, and in those cases the code can be quite inefficient because it continues to use the higher order BDF and needs to take very small stepsizes to maintain stability. It is important to note that for all modern ODE solvers, the stepsize selection strategy reduces the stepsize as a response to instability. Thus the codes do not give a faulty solution because of the instability for large stepsizes, but instead reduce the

stepsize and become very inefficient. Because of these differences in the order selection strategies between DASSL and LSODE, DASSL is more robust than the LSODE codes for this type of problem. Order selection strategies to control instability in higher order BDF methods are discussed in detail in [231].

After the step has been accepted or rejected and the order of the method to be used on the next step has been determined, DASSL decides what stepsize to use on the next step. The most effective strategy, based on our experiences and experimentation, is the one used by Shampine and Gordon [225]. The error at the new order k is estimated as if the last $k+1$ steps were taken at the new stepsize, and the new stepsize is chosen so that the error estimate satisfies (5.2.21). More precisely, DASSL chooses the new stepsize rh_{n+1} conservatively so that the error is roughly one half of the desired integration error tolerance. Thus r is given by

$$r = (2.0\text{EST})^{-1/(k+1)}, \tag{5.2.23}$$

where EST is the error estimate for the order k method which was selected for the next step and is given by the term estimated in (5.2.22) for the new order k, but scaled now by the error constant $1/(k+1)$. After a successful step, the stepsize is increased only if it can be doubled, and then it is doubled. If a decrease is called for after a successful step, the stepsize is decreased by at least a factor $r = 0.9$, and at most $r = .5$. After an unsuccessful step, if this is the first failure since the last successful step, then r is determined by (5.2.23) and multiplied by 0.9. The stepsize is reduced by at least $r = 0.9$ and at most $r = 0.25$. After the second failure, the stepsize is reduced by a factor of one quarter, based on the philosophy that the error estimates can no longer be trusted.

DASSL keeps a count of the number of integration error test failures since the last successful step. After a large stepsize decrease, the fixed leading coefficient BDF formulas are most accurate at order one, in contrast to the variable coefficient formulation where the accuracy does not deteriorate so much for higher order formulas when the stepsize is decreased drastically. It is likely that the first order formula is best anyway in this situation because the past values of y will not yield much useful information if the error test has already failed several times. After three consecutive error test failures, the order is reduced to one, and the stepsize is reduced by a factor of one quarter on every failure thereafter. When the stepsize is so small that $t + h \approx t$, an error return is made. In DASSL this minimum stepsize is computed as $h_{\min} = 4u\max(|t_n|, |TOUT|)$ where u is the unit roundoff error, t_n is the current mesh point, and TOUT is the user requested output point.

As a final issue in this subsection, we discuss how DASSL selects the initial stepsize. Even for ODE codes, this issue is somewhat tricky. Numerous strategies have been proposed. See for example [116,225,237]. DASSL uses

$$h_0 = \text{sign}(\text{TOUT} - \text{T})\min\left(10^{-3}|\text{TOUT} - \text{T}|, \frac{1}{2}||y'||^{-1}\right).$$

This strategy is designed to yield a successful step for a zeroth order method and is quite conservative. If $||y'||$ is very small, then it is hoped that TOUT−T gives some information on the scale of the solution interval. It is not terribly uncommon in applications for $||y'||$ to be zero or very small at the initial time, as frequently the derivative values are unknown for the algebraic variables. In these cases, the value of TOUT influences the initial stepsize. Hence, different values of TOUT can change the performance of the code. This aspect of DASSL's initial stepsize algorithm is similar to the strategy proposed by Söderlind in his code DASP3 [233] for the numerical solution of coupled stiff ODE's and DAE's. There is an option in DASSL for the user to select the initial stepsize.

5.3 Obtaining Numerical Solutions

5.3.1 Getting Started with DASSL

DASSL is designed to be as easy to use as possible, while providing enough flexibility and control for solving a wide variety of problems. It is extensively documented in the source code. The user interface for DASSL is based on the user interface for ODE solvers proposed by Shampine and Watts in [226], with a few changes which are necessary to accommodate the more general DAE systems. In this subsection we outline what a user must do to solve a problem with DASSL.

We emphasize that DASSL is designed for solving index zero and index one problems of the form

$$\begin{aligned} F(t, y, y') &= 0 \\ y(t_0) &= y_0 \\ y'(t_0) &= y'_0, \end{aligned} \tag{5.3.1}$$

where F, y and y' are N-dimensional vectors. DASSL makes use of a subroutine RES which is written by the user to define the function F in (5.3.1). RES takes as input the time T and the vectors Y and YPRIME, and produces as output the vector DELTA, where DELTA = F(T,Y,YPRIME) is the amount by which the function F fails to be zero for the input values of T, Y and YPRIME. The subroutine has the form

```
SUBROUTINE RES(T,Y,YPRIME,DELTA,IRES,RPAR,IPAR)
```

The parameter IRES is an integer flag which DASSL always sets to zero prior to calling RES. It is used to flag situations where an illegal value of Y or a stop condition has been encountered. For example, in some applications, if a component of Y becomes even slightly negative, then the function F cannot be evaluated. In this case, the user would check for a negative component of Y upon entering RES. If one is found, then the user would set IRES = −1

and return without evaluating the function. DASSL then cuts the stepsize and attempts the step again. RPAR and IPAR are real and integer vectors, respectively, and are at the user's disposal to use for communication purposes. They are never altered by DASSL or any of its subroutines.

To get started, DASSL needs a consistent set of initial values T, Y and YPRIME. A necessary but not always sufficient condition for consistent initialization is that F(T,Y,YPRIME) = 0 at the initial time. At the time of this writing, we are not aware of any general codes for computing consistent initial values of Y and YPRIME, given enough information about Y and YPRIME to specify a unique solution to the analytical problem. An algorithm for accomplishing this task in general has recently been developed and will be discussed in Section 5.3.4. For problems in which the initial values of Y are given, there is an option in DASSL to compute the initial value of YPRIME, given a starting guess for YPRIME. In this case, DASSL takes a small implicit Euler step for its first step, and uses a damped Newton iteration to solve the nonlinear system. The error estimate on this step is different from the estimate which DASSL usually uses because the initial derivatives are not available for use in an error estimate. It is sometimes possible to start DASSL without consistent values of YPRIME, but it should be noted that the error estimates on the first step are not correct in this case unless the user has specified the option for the code to compute the initial values of YPRIME.

The call to DASSL is

```
CALL DASSL(RES,NEQ,T,Y,YPRIME,TOUT,INFO,RTOL,ATOL,
IDID,RWORK,LRW,IWORK,LIW,RPAR,IPAR,JAC)
```

The parameters are described in detail in the documentation to DASSL, so we will only discuss a few features here. Many of these features are activated by setting an element of the option vector INFO to one. The user subroutines RES and JAC must be declared external in the user's main program. The variable NEQ equals N, the number of equations in the system (5.3.1). In case DASSL fails, the scalar variable IDID should be examined to see what specific error caused the difficulty. The documentation of DASSL gives a detailed explanation of all the error messages.

Frequently a user desires the numerical solution at a set of output times, so the usual way to call DASSL is in a loop which increments the output time TOUT until the end of the interval has been reached. The value of the solution which DASSL returns is the value of the interpolant (5.2.9) at those times. To obtain more detailed information about the solution at the internal timesteps that DASSL selects, there is an option to have DASSL return after each internal time step.

As we have described in Section 5.2.2, the algorithms in DASSL require an estimate of the iteration matrix, $G = \alpha\partial F/\partial y' + \partial F/\partial y$. There is an option to specify this matrix directly which requires the user to provide a subroutine JAC. The default is for DASSL to approximate this matrix via finite

differences. The advantage of the finite difference option is that it is simpler. We strongly recommend, especially for novice users and new problems, to let DASSL compute the matrix. The JAC subroutine is usually not simple to write and debug. The consequence of a very poor Jacobian matrix approximation is a serious deterioration of efficiency or possibly even complete failure. For cases in which DASSL is inefficient because of a poor finite difference Jacobian approximation, providing a subroutine JAC will improve performance. In Section 5.3.3, we give some suggestions on how to recognize this situation. Which option is most efficient is quite problem dependent.

For many problems in applications, the iteration matrix is banded. If the code is informed of this matrix structure by an appropriate input, then the storage needed will be greatly reduced, the numerical differencing will be performed much more cheaply, and a number of important algorithms will execute much faster. In the case that the user computes a banded matrix via subroutine JAC, the elements must be stored in a special format which is explained in the documentation.

For a very few problems in applications whose analytic solutions are always positive, we have found that it is crucial to avoid small negative solutions which can arise due to truncation and roundoff errors. There is an option in DASSL to enforce nonnegativity of the solution. However, we strongly recommend against using this option except when the code fails or has excessive difficulty without it.

Finally, a comment on the relative and absolute error tolerances RTOL and ATOL. These variables can be specified either as vectors or as scalars. Most of the important decisions in the code make use of these tolerances to compute weights for the norm, and the finite difference Jacobian approximation makes use of them directly when there is no other information available about the scale of the solution. *We cannot emphasize strongly enough the importance of carefully selecting these tolerances to accurately reflect the scale of the problem.* In particular, for problems whose solution components are scaled very differently from each other, it is advisable to provide the code with vector valued tolerances. For users who are not sure how to set the tolerances RTOL and ATOL, we recommend starting with the following rule of thumb. Let m be the number of significant digits required for solution component y_i. Set $\mathrm{RTOL}_i = 10^{-(m+1)}$. Set ATOL_i to the value at which $|y_i|$ is essentially insignificant.

The norm which DASSL uses is a weighted root mean square norm, given by

$$\|\mathrm{v}\| = \sqrt{(1/\mathrm{NEQ})\sum_{i=1}^{\mathrm{NEQ}}(v_i/\mathrm{WT}_i)^2}$$

where

$$\mathrm{WT}_i = \mathrm{RTOL}_i|\mathrm{Y}_i| + \mathrm{ATOL}_i.$$

These are the best choices for the majority of problems. However, for some

problems, particularly method of lines solutions to partial differential equations, it is better to make use of additional information on the structure of Y to define the weights and/or the norm. It is possible for the user to replace these subroutines, but we generally recommend against this. Details are given in the DASSL documentation.

The FORTRAN source code for DASSL is available by way of the National Energy Software Center (NESC). The address is Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, U.S.A. Single and double precision versions are available. The accession number for DASSL at NESC is 9918. Prospective users from Western Europe or Japan can obtain DASSL from the NEA Data Bank. The address is B.P. No. 9 (Bat. 45) F-91190, GIF-sur-YVETTE, France.

5.3.2 Troubleshooting DASSL

In our experience, DASSL solves most DAE systems from applications without difficulty. However, we have occasionally encountered systems for which DASSL has trouble. In this subsection we will describe some of the possible sources of trouble and give some advice on how to recognize the problem, along with possible remedies.

In diagnosing problems, it is helpful to be aware of the existence of information about the internal operations of DASSL. Information which is accessible through the work arrays RWORK and IWORK includes: the stepsize to be attempted on the next step, the stepsize used on the last successful step, the order of the method to be attempted on the next step, the order of the method used on the last step, and running totals of the number of steps taken, the number of calls to RES, the number of evaluations of the iteration matrix needed, the number of integration error test failures, and the number of convergence test failures. We have found the information on integration error test failures and convergence test failures to be particularly useful. The locations of this information are given in the DASSL documentation.

Before proceeding, we wish to emphasize that the DASSL code has successfully solved a wide variety of scientific problems and has had a large number of users. A user who suspects a bug should first very carefully examine his own code and reread the DASSL documentation. *In our experience, bugs in user code and incorrect use of DASSL are by far the most likely causes of difficulties.*

Inaccurate Numerical Jacobian

The finite difference Jacobian calculation is, in our opinion, the weakest part of the DASSL code. It is a difficult problem to devise an algorithm for calculating the increments for the differencing which is both cheap and robust. We have seen cases of extremely poorly scaled systems from applications, where some elements of the finite difference Jacobian were computed to be zero when they should have been quite large. To see how this can happen, suppose there is a term in the system which looks like $\lambda(y_1 - y_2)$, where $y_1 >> y_2$. In computing the partial derivative of this term with respect to y_2, the difference is based on the size of y_2. Thus if y_1 is large enough compared to y_2, we will have $y_1 - (y_2 + \delta y_2) = y_1 - y_2$ in machine arithmetic and the code will think that the partial derivative is zero. It is likely that in future versions of DASSL a more robust finite difference Jacobian algorithm such as the one proposed by Salane [218] will be considered.

Usually, the symptoms of an inaccurate numerical Jacobian are a stepsize that is much smaller than appears to be warranted from accuracy considerations, and high ratios of convergence test failures both to error test failures and to steps taken so far.

There are several possible remedies to difficulties with obtaining good finite difference approximations to the Jacobian matrix. The function subroutine RES can often be rewritten or a change of variables made to avoid cancellations and poorly scaled variables. A user defined analytic JAC routine can be written, or a user defined finite difference Jacobian routine can be substituted. Sometimes a more careful look at the error tolerances helps. In particular, increasing ATOL for exceedingly small components which do not need to be computed accurately will often remedy this problem. This is because of the way that ATOL is used in the calculation of the increment δ_j for finite difference Jacobians, as we have explained in Section 5.2.2.

Finally, we note that if there is an iteration inside the user RES routine (for example, to solve a nonlinear system), then the increment for the numerical Jacobian should be based on the error of this iteration, rather than on the unit roundoff error. Because such an iteration can also cause difficulties with the stepsize selection and error estimates in DASSL, we strongly encourage DASSL users to avoid such situations whenever possible by reformulating the problem.

Inconsistent Initial Conditions/Discontinuities

If the user-supplied initial conditions for y and y' are inconsistent, then DASSL may fail on the first step. The symptom of the failure can be either successive error test failures or, for nonlinear problems, the Newton iteration may fail to converge due to a poor initial guess. For index one systems with inconsistent initial conditions, the error estimate tends to a constant as the stepsize approaches zero. To see why this is true, consider the simple algebraic equation $y = g(t)$, with $y_0 = a \neq g(t_0)$. DASSL always starts with the implicit Euler method because it requires no memory of past steps. An implicit Euler step for this system gives $y_1 = g(t_1)$. The error estimate is $(1/2)||y_1 - y_1^{(0)}||$, which tends to $(g(t_0) - a)$ as $t_1 \to t_0$. A more extensive discussion of the difficulties encountered by codes based on the BDF methods when applied to DAE's is given in [200].

It is important to note that initial guesses which are only slightly inconsistent can cause DASSL to fail to complete the first step. Failure occurs because the errors, and therefore the inconsistencies, are measured in the weighted norm which is determined by RTOL and ATOL. Hence it is important for the user to compute these initial conditions very accurately. If a discontinuity is introduced, perhaps inadvertently by a user input function, into an algebraic variable of the system, this same behavior can also occur in later steps.

Higher Index Systems

DASSL cannot solve higher index systems without modifications to the error estimates and other strategies in the code. We will discuss the possibilities for solving higher index systems in Section 5.4. Here we focus on recognizing these types of systems when they have inadvertently been given to DASSL. In general, the error estimates used in DASSL do not tend to zero as the current stepsize tends to zero for systems whose index is greater than one – in fact, they may even diverge! As a response to a large integration error estimate, the code drastically (and repeatedly) reduces the stepsize until the iteration matrix becomes ill-conditioned. For a system of index ν, we show in Section 5.4.2 that the iteration matrix has condition $O(h^{-\nu})$. For small enough stepsizes, this conditioning problem causes the Newton iteration not to converge, and the code eventually fails due to multiple convergence test failures. There is an error return flag in DASSL which signals multiple integration error test failures followed by multiple convergence test failures. This error return is usually a signal of a higher index problem, or possibly of an index one problem with inconsistent initial conditions or a discontinuity. However, it is also quite possible for these types of problems to fail solely because of multiple convergence test failures, particularly for nonlinear problems with index greater than two. Sometimes DASSL can start solving a smooth higher index system and fail after a stepsize or order change or a rapid change in the solution, so the failures will not necessarily occur on the first step.

Singular Iteration Matrix

While it is possible to define a solvable DAE system whose iteration matrix is singular for all possible values of the stepsize, we have never seen such a problem occur in an application and it would necessarily have to have an index which was greater than one. DASSL gives an error return flag for a singular iteration matrix only after the code has tried to compute the iteration matrix for several quite disparate stepsizes and has always produced a matrix which is singular. It is possible that a singular matrix can be created by a very inaccurate finite differencing of the matrix within DASSL or by a bug in a user supplied JAC routine. Singularity can also occur because there are redundant equations in the system to be solved (that is, redundant equations in RES). In this case the user needs to reformulate the problem to include

some extra information and remove the redundant information. Otherwise there is no possibility for obtaining a unique solution. In our experience, this bug is a fairly common occurrence in codes using DASSL in the method of lines solution of partial differential equations when the boundary conditions have been improperly specified.

Poorly Scaled System/Inappropriate Error Tolerances

The error tolerances RTOL and ATOL are the means by which the user communicates to DASSL the accuracy requirements of the solution and the scale of the problem. It is possible for DASSL to miss important changes in the solution because these error tolerances are too loose. On the other hand, if the tolerances are too tight, then DASSL will work very hard to get accurate solutions for variables which may be very small and insignificant. In particular, we do not recommend using pure relative error control (ATOL = 0) for solution components which may be small or become small during the integration. The code tends to function most efficiently if it does not need to take unnecessarily small stepsizes because the iteration matrix becomes more singular as the stepsize tends to zero. On the other hand, the error estimates in DASSL are valid only for sufficiently small stepsizes. As with many ODE codes, DASSL may not perform well with exceedingly loose tolerances. We recommend asking for at least two digits of accuracy except possibly for exceedingly small and unimportant solution components.

5.3.3 Extensions to DASSL

Although DASSL is a powerful code which can handle a wide variety of problems, some problems require capabilities which are not implemented in the standard version of DASSL or which can be performed more efficiently with a code based on DASSL but oriented towards a more specific task. In this subsection we will discuss DASSLRT, a root finding version of DASSL, DASSAC, a version of DASSL for sensitivity analysis problems, and several other extensions to DASSL which are at this time under development.

The standard version of DASSL solves a DAE from time T to time TOUT, where TOUT is specified by the user. In some problems it is more natural to stop the code at the root of some function $g(t, y)$. For example, in computing the trajectory of an object, it does not make sense to continue the solution past the time when the object hits the ground. In other problems, it is necessary to stop the code at the root of g because the function F changes at the roots of g. For example, in computing the flow of a fluid, the flow is 'choked' if the pressure ratio exceeds a critical value, and unchoked otherwise. Choked flows obey a different set of DAE's than unchoked flows, so it is necessary to check for the root of the function defined by the pressure ratio minus the critical value, and define the function F in RES according to whether the flow is currently choked or not.

In our experience, some users have attempted to implement these types of root finding problems by first inserting an IF statement in the RES routine and switching the function based on the sign of g. We do not, in general, recommend this approach. Since linear multistep methods utilize function and derivative information over several integration steps, it does not make sense to switch the function (effectively introducing a discontinuity in the derivatives) without restarting the integration. The sudden change in the derivatives of F can wreak havoc with the numerical Jacobian approximation, Newton iteration, error estimates, and stepsize and order selection algorithms in DASSL. It can be grossly inefficient and even lead to failure for some problems. Instead, we recommend setting a flag in the main program which determines which case of the function is to be evaluated. The flag is communicated to RES via IPAR. Then, DASSLRT should be called. When it returns at a root, the flag is changed in the main program. It may also be necessary at this time to enforce consistency of the constraint equations. Then DASSLRT is restarted. For complex problems which involve several cases of F, it may be necessary to use several flags.

It is helpful in formulating problems to understand roughly how DASSLRT works. First, whenever DASSL completes a successful step to t_{n+1}, it evaluates the root function $g(t_{n+1}, y_{n+1})$. If the sign of $g(t_{n+1}, y_{n+1})$ is different from the sign of $g(t_n, y_n)$, then a root must have occurred in the interval $[t_n, t_{n+1}]$. DASSL then calls a root solver to find the root of $g(t, \omega^I_{n+1}(t)) = 0$ in $[t_n, t_{n+1}]$. The root solver in DASSLRT is based on ideas developed by K. Hiebert and L. F. Shampine [142]. The 'Illinois algorithm' [86] is used for actually locating the root.

In general, a vector of functions $g_i(t, y)$, $i = 1, 2, \ldots, \mathrm{NG}$, may be supplied to DASSLRT such that the root of any of the NG functions g_i is desired. Of course, there may be several such roots in a given output interval. DASSLRT returns them one at a time, in the order in which they occur along the solution. An integer array tells the user which g_i, if any, were found to have a root on any given return. We note that because DASSLRT detects roots by looking for sign changes in g, it is conceivable for it to skip intervals where there are multiple roots. We have not found this to be a problem in the applications that we have seen.

At the time of this writing, DASSLRT exists and is routinely used to solve application problems at Lawrence Livermore National Laboratory and at Sandia National Laboratories, Livermore. A version for outside release is planned.

A second extension of DASSL has been developed to handle sensitivity analysis for DAE's. Suppose we are given a DAE

$$F(t, y, y', p) = 0, \tag{5.3.2a}$$

$$y(t_0, p) = y_0(p) \tag{5.3.2b}$$

where $y, y' \in \mathcal{R}^N$, whose solution depends on a vector of M time independent parameters p. We would like to compute the (N, M) matrix $W(t)$ of sensitivity functions

$$W(t) = \frac{\partial y(t)}{\partial p} \tag{5.3.3}$$

which describes how the solution components change as a result of changes in the parameters.

The sensitivity matrix satisfies a system of DAE's which can be derived by partial differentiation of equations (5.3.2) with respect to the parameter vector p

$$\begin{aligned} \frac{\partial F}{\partial y'} W'(t) + \frac{\partial F}{\partial y} W(t) + \frac{\partial F}{\partial t} + \frac{\partial F}{\partial p} &= 0 \\ W(t_0) &= \frac{\partial y_0}{\partial p}. \end{aligned} \tag{5.3.4}$$

Computationally, the important observation with respect to the sensitivity equations (5.3.4) is that they are linear and that they have the same iteration matrix as the original system (5.3.2) if they are solved using the same sequence of methods and stepsizes. Caracotsios and Stewart [66] have written a code DASSAC for solving the sensitivity system which makes use of these observations and is based on DASSL. The code works as follows. The sensitivity equations (5.3.4) are appended to the original system. The system is solved using a version of DASSL which has been modified to evaluate the iteration matrix on every step, require only one Newton iteration for the linear system (5.3.4), and make use of the repetitive block diagonal structure of the iteration matrix to save storage and operations in forming the matrix and solving the linear system. The error estimates and stepsize adjustment are based on the entire system (5.3.4) appended to (5.3.2). It is to be noted that the sensitivities may not be accurate if the iteration matrix is computed via finite differencing.

One of the obvious deficiencies of the standard DASSL is the lack of options for dealing with Jacobian matrices which do not fit naturally into a dense or banded category. This is particularly apparent in the case of large-scale problems from chemical engineering applications (see for example [110]) and in the method of lines solution of partial differential equations in more than one dimension. Marquardt [180] has recently implemented a version of DASSL which makes use of the HARWELL sparse direct linear system solver. A version of DASSL which uses the preconditioned GMRES method to solve the linear system at each Newton iteration is currently under development by Brown, Hindmarsh and Petzold, and is expected to be released soon. This code implements many of the strategies described for ODE's in [36]. Some initial experiences with the code are described in [204].

Finally, we note that Skjellum et al.[232] have recently implemented a version of DASSL for use on massively parallel computers, called Concurrent DASSL.

5.3.4 Determining Consistent Initial Conditions

Often the most difficult part of solving a DAE system in applications is to determine a consistent set of initial conditions with which to start the computation. More precisely, we formulate this problem as follows. Given information about the initial state of the system which is sufficient, in a mathematical sense, to specify a unique solution to the DAE, determine the complete initial state $(y(t_0), y'(t_0))$ corresponding to this unique solution. For example, the user may specify information about the solution and/or its derivatives at the initial time, and the problem is to determine the remaining values of the solution and its derivatives. In this subsection we describe a newly developed general algorithm for finding consistent initial conditions for index one and semi-explicit index two DAE systems. This work is described in more detail in Leimkuhler et al.[162].

For a simple algorithm, one might consider substituting $y = y_0$, $y' = y'_0$ and solving the resulting system, together with the user defined initial data, for y_0 and y'_0. For example, for an ODE $y' = Ay$ together with user defined information $Cy'_0 + Dy_0 = q_0$ at the initial point, we have

$$\begin{bmatrix} -A & I \\ D & C \end{bmatrix} \begin{bmatrix} y_0 \\ y'_0 \end{bmatrix} = \begin{bmatrix} 0 \\ q_0 \end{bmatrix}$$

which we can solve uniquely for y_0 and y'_0 provided the left hand matrix is of full rank and q_0 is in the range of the matrix $CA + D$. In this case, these are the same conditions which the matrices D and C must satisfy to ensure that the user has given enough information about the initial state to specify a unique solution to the mathematical problem.

On the other hand, consider the index one DAE given by

$$\begin{aligned} Y'_1 + Y'_2 + Y_1 &= g_1(t) \\ Y_2 &= g_2(t). \end{aligned} \tag{5.3.5}$$

To specify a unique solution to the DAE, it is sufficient to give the value of either Y_1 or Y'_1 at t_0, because Y_2 and Y'_2 must satisfy the constraint and the derivative of the constraint. However, evaluating Y_1, Y'_1, Y_2, and Y'_2 at time t_0 in (5.3.5), it is apparent that it is not possible to obtain $Y_1(t_0)$ uniquely if only $Y'_1(t_0)$ is specified since $Y'_2(t_0)$ is unknown. Thus this simple algorithm fails to give the solution, although the user has given enough information to specify a unique solution to the original problem.

Clearly the difficulty with the above procedure for equations (5.3.5) is that the simple algorithm has no way of obtaining the information about the derivative of the constraint which is inherent in the system. In the fully-implicit index one problem it is not in general possible to isolate the constraints and differentiate them. Thus we are led to consider the following algorithm, motivated by the definition of global index given in Chapter 2. We assume that the DAE system (5.3.1) has index ν.

Solve

$$\begin{aligned} F(t_0, y_0, y'_0) &= 0 \\ \frac{dF}{dt}(t_0, y_0, y'_0, y''_0) &= 0 \\ &\vdots \\ \frac{d^\nu F}{dt^\nu}(t_0, y_0, y'_0, \ldots, y_0^{(\nu+1)}) &= 0 \end{aligned} \tag{5.3.6}$$

coupled with the user defined information

$$B(t_0, y_0, y'_0) = 0. \tag{5.3.7}$$

It is easy to show that if the user defined information (5.3.7) is sufficient to determine a unique solution to the DAE, then (5.3.6), (5.3.7) have a solution $(y_0, y'_0, \ldots, y_0^{(\nu+1)})$, in which the first two components y_0, y'_0 are uniquely determined and are the solution to the DAE at t_0. Note that the higher derivatives of y are not determined uniquely by this algorithm.

Since it is frequently not practical to obtain the derivatives of F analytically, we are led to consider approximating the derivatives of F. Because F is possibly only defined for $t \geq t_0$, it is natural to consider using one sided differences. For example, the simplest one sided difference is given by

$$D_h F = \frac{F(t_0 + h, y_0 + hy'_0, y'_0 + hy''_0) - F(t_0, y_0, y'_0)}{h}.$$

We can define higher order difference approximations by

$$D_h F = \frac{1}{h}\left[\sum_{i=1}^{s} \alpha_i F(t_0 + hc_i, y_0 + hc_i y'_0, y'_0 + hc_i y''_0) - \left(\sum_{i=1}^{s} \alpha_i\right) F(t_0, y_0, y'_0)\right] \tag{5.3.8}$$

by choosing the constants α_i, c_i appropriately. The higher derivatives of F can be approximated similarly.

The algorithm obtained by replacing the consistency equations (5.3.6) by their approximations (5.3.8), coupled with the user defined information (5.3.7), produces a rank deficient over-determined nonlinear system. Unlike the analytic consistency equations, the approximate system may not have an exact solution because some of the derivative approximations to F may be approximating user defined information. Thus the approximate system is solved in a least squares sense. However, the solution of this system is complicated by the rank deficiency of the Jacobian. Using the structure of the system, it is possible to show that the minimum norm solution to this nonlinear least squares problem converges to the correct solution as the approximations D_h become more accurate. One can also formulate a scheme for replacing the system with a full rank system which has the same solution for y_0, y'_0. Numerical results for this technique appear to be promising.

In other approaches to the consistent initialization problem for DAE's, Berzins, Dew and Furzeland [16], in their code SPRINT, use the initialization method in DASSL to approximate the derivatives, and then improve these by also attempting to satisfy a difference approximation to the first time derivative of the DAE. The algorithm that we have described in this section can be considered to be a generalization of this idea. Pantelides [199] uses a graph theoretic algorithm to determine the minimal set of equations to be differentiated in order to solve for the consistent initial values. Mrziglod [190] gives an algorithm which is based on a thorough decomposition of the system structure. These latter two algorithms require knowledge of analytical expressions for various derivatives of the problem, which is a drawback for large scientific problems.

To our knowledge, no general purpose software for solving the consistent initialization problem is available at this time.

5.4 Solving Higher Index Systems

5.4.1 Alternate Forms for DAE Systems

In this subsection we consider some techniques for rewriting a DAE system in an alternative form that may be easier to solve numerically. All of the different forms of the equations that we consider are equivalent in the sense that, given a consistent set of initial conditions, different forms of a system have the same analytical solution. Computationally, however, some forms of the equations may have much different properties than others. We discuss some of the advantages and disadvantages of rewriting a high index DAE system in a different form.

As an example, let us consider some different ways to solve the constrained mechanical system given by

$$\begin{aligned} M(q)q'' &= f(q,q',t) + G(q)\lambda, \\ \Phi(q) &= 0. \end{aligned} \tag{5.4.1}$$

Here, the mass matrix M is nonsingular almost everywhere, λ is the Lagrange multiplier vector, and $\partial\Phi/\partial q = G^T$.

We can attempt to solve the system in its original, index three form, using an implicit numerical method such as BDF. This technique is actually used in some codes [197] for solving mechanical systems. Solving the problem in this way has the advantages that it is easy to formulate the system, as we do not have to differentiate the constraints or rewrite the system in any way, the sparsity of the system is preserved, and the constraints are satisfied exactly on every step. The disadvantage is that there are several difficulties in using a variable stepsize BDF code for solving systems in this form, which we describe in detail in Section 5.4.2. Many of the difficulties can be circumvented, but in general it is not a simple matter to obtain an accurate numerical solution for

the Lagrange multipliers λ. For BDF methods, λ and q' must be filtered out of the error estimate. A code based on this strategy does not always give reliable results for the variables which are filtered out of the estimate. In particular, results are likely to be unreliable in situations where there are steep gradients or discontinuities in the velocities. This type of error control strategy can also give incorrect solutions when the initial conditions are improperly specified in the sense that the algebraic constraint is satisfied but the derivative of the constraint is not. It may be possible to reliably solve systems in the index three form with methods other than BDF, such as extrapolation or defect correction, by controlling errors on the velocities as well as the positions, but we are not aware of any further work in this area.

A second way of solving (5.4.1) is to differentiate the constraint one or more times and solve the resulting lower index system. Except for the index zero system, there are still numerical difficulties, but they are less severe for the lower index systems. It should be noted that this type of lower index formulation of a problem does not force the constraints to be satisfied on every step, and there may be a tendency for the amount by which the constraint is not satisfied to increase from step to step. In Chapter 6 the 'drift' in the algebraic constraints is demonstrated in numerical experiments with DASSL on an index three pendulum problem when it is reformulated as an index one problem. By using small stepsizes, or in an automatic code by keeping the error tolerances fairly stringent, we can usually keep small the amount by which the constraint is not satisfied. Whether this is a serious problem or not depends on the application, although clearly it could be troublesome if the solution is desired over a long interval in time. In a recent paper, Führer and Leimkuhler [108] have shown by example that differentiating a nonlinear constraint can be dangerous in the sense that the stability properties of the new system will not be the same as the stability properties of the original system. Thus this approach should be used only with caution.

Another alternative is to use the index reduction techniques of Gear [118], which were described in Section 2.5.3, (see also Section 6.2) to rewrite the system as a lower index system. This approach requires differentiating the constraints and adding additional (algebraic) variables which act as Lagrange multipliers. This process can be performed repeatedly, to yield a semi-explicit index two system. The solution of this system satisfies both the constraints and the derivatives of the constraints. Because the solutions for the algebraic variables of this system are zero, the resulting system can be solved by quite general multistep methods, as described in Gear [117]. Another possibility for the constrained mechanical systems is to do the differentiation only once, and add the extra Lagrange multipliers. While the resulting system is a simpler index two system, not all of the algebraic variables are zero now. Therefore BDF methods, or appropriate implicit Runge-Kutta methods, can be used to solve the system. In the next section this promising approach using the BDF methods is pursued further. Another possibility suggested by Gear [118] for

reducing the index in semi-explicit systems $x' = f(x,y,t)$, $0 = g(x,y,t)$, is to replace the undifferentiated variables y by w'. While the resulting system in x, w, t is one index lower, the solution of this new system yields only x and w so it is still necessary to differentiate the computed w to obtain y.

Yet another strategy, which is dependent upon knowing more information about the structure of the problem, and is used in some codes for solving mechanical systems (see Section 6.2), is to eliminate the Lagrange multipliers analytically using methods in analytical mechanics to obtain a standard form ODE system. The system of ODE's is assembled from a data structure describing the mechanical system. If the resulting problem is not stiff, this approach has the advantage that the system can be solved by an explicit numerical method. The number of unknowns after this type of transformation usually is smaller, but the sparsity of the system has decreased, which is an important consideration if the problem happens to be stiff. Again, we must be very careful that the initial conditions satisfy both the constraint and its derivative, or we will obviously obtain a solution which is nonsense. A constraint corresponding to an eliminated Lagrange multiplier is automatically satisfied in the chosen representation of the mechanical system. As an example, consider solving the equations of a pendulum in Cartesian coordinates. The system is written as

$$\begin{aligned} x'' &= \lambda x, \\ y'' &= \lambda y - g, \\ 0 &= \frac{1}{2}(x^2 + y^2 - L^2), \end{aligned} \tag{5.4.2}$$

where g is the gravity constant and L is the length of the bar. Let

$$x = L\sin\phi, \qquad y = -L\cos\phi.$$

Then the algebraic constraint of constant length is fulfilled and the well-known ODE is

$$\phi'' - \frac{g}{L}\sin\phi = 0.$$

This approach may be difficult to implement in general or for very large systems. Finally, we note that Führer and Leimkuhler [108] have recently studied the implications of the various formulations for mechanical systems. See also Section 6.2 for a further discussion on the formulation and solution of systems describing constrained mechanical systems.

A different way of dealing with the high index systems is through the use of penalty functions or regularizations. We loosely define the *regularization* of a DAE to be the introduction of a small parameter into the DAE in such a way that the solution of the perturbed system approaches the solution of the unperturbed system as the parameter tends to zero. Baumgarte [14] discusses a technique for circumventing the problem of 'drifting off' the constraints $\Phi(q)$ by adding to the original equations an equation consisting of a linear

combination of Φ, $d\Phi/dt$ and $d^2\Phi/dt^2$. The linear combination is chosen so that the resulting system damps errors in satisfying the constraint equation. This idea was used successfully by Adjerid and Flaherty [1] to stabilize the constraint in the solution of partial differential equations by adaptive moving mesh methods. The stabilization approach is similar, but not identical, to penalty function methods (Lötstedt [165], Sani et al. [219]). Depending on the choice of the parameters in the linear combination, we may see any of the difficulties discussed earlier. This technique introduces extraneous eigenvalues into the system, which may or may not cause difficulties. Finally, the penalty techniques have the disadvantage that if the initial conditions are not posed correctly, they introduce a nonphysical transient into the problem [219]. März [183] gives a regularization for the semi-explicit index two system so that the regularized system is index one. In recent experiments, Knorrenschild [153] reports that DASSL solves some regularized semi-explicit index two systems much more efficiently than systems which are not regularized. The regularization which Knorrenschild uses is motivated by physical considerations from circuit analysis applications but can be applied to general problems. We feel that at this time the results on regularization as a general technique for solving higher index systems are inconclusive. However, approaches based on some form of regularization have definitely proven successful in some applications. Regularization is discussed in more detail for multistep methods in Section 3.3, and for Runge-Kutta methods in Section 4.5.

Before leaving the subject of alternate forms for DAE systems, there is one more aspect of this problem that we wish to consider. Sometimes there is a choice of which variables to use for solving a problem. For example, in the system

$$\begin{aligned} u' &= v, \\ v' &= f(u,v,t) + G(u)\lambda, \\ G^T v &= 0 \end{aligned} \tag{5.4.3}$$

we could have replaced v in the constraint by u' to obtain

$$\begin{aligned} u' &= v, \\ v' &= f(u,v,t) + G(u)\lambda, \\ G^T u' &= 0. \end{aligned} \tag{5.4.4}$$

In implementing these systems, the question naturally arises, are there any advantages in writing the system in one form over the other? Using BDF methods combined with Newton iteration for linear problems in exact arithmetic, the two forms of the equations give identical solutions. This is because Newton's method is exact in one iteration for linear systems, and because the equations which result from discretizing both systems by BDF are identical. This last fact is obviously true for nonlinear systems too. For nonlinear systems in exact arithmetic we know of no reasons why Newton's method would

be more likely to converge for one form of the equations than the other. Both forms of the system may lead to very poorly conditioned iteration matrices. In the next section we will suggest scaling techniques to overcome this difficulty. These techniques are directly applicable to systems such as (5.4.3) which are in Hessenberg form, but not to systems such as (5.4.4) which are not in any standard recognizable form. Equation (5.4.3) may be preferred for just this reason. This question of which variables to use in writing a system may seem like a minor point, but in adaptive moving mesh calculations for partial differential equations we found that the formulation of the problem in this sense was crucial [203]. Sometimes, as happened in this application, a variation in the problem formulation which is equivalent in exact arithmetic to the original formulation can drastically change the condition of the iteration matrix and dramatically affect the reliability and efficiency of the calculation. We will investigate this adaptive moving mesh example further in Chapter 6. In general, we recommend investing some time into the problem formulation step of the solution process, keeping in mind the standard forms which have so far proven so the most successful.

5.4.2 Solving Higher Index Systems Directly

Although convergence rates are known for BDF and implicit Runge-Kutta methods applied to some forms of semi-explicit higher index systems, the implementation of codes based on these methods for solving higher index systems is not straightforward. Here we examine some of the difficulties and suggest some remedies.

Matrix ill conditioning is a problem for numerical methods when applied to DAE systems, and especially to higher index systems. We specifically study the conditioning of the iteration matrices corresponding to the BDF methods and useful (i.e., efficiently implementable) IRK methods, such as the SIRK's or DIRK's. It is not practical to consider general IRK methods where the stage equations cannot be uncoupled, either naturally or via transformations in the case of SIRK's, due to the high cost of solving large nonlinear systems. Even so, the iteration matrices for general IRK methods will contain blocks with a structure resembling the one studied below. Hence their conditioning is expected to be very similar. From Theorem 2.3.4 we have

Theorem 5.4.1 *The condition number of the iteration matrix for a system with (local) index of* ν is $O(h^{-\nu})$.

It is often possible to reduce the condition number by scaling the DAE system appropriately. This is especially true for semi-explicit systems

$$\begin{aligned} x' &= f(x,y,t) \\ 0 &= g(x,y,t). \end{aligned}$$

For these systems, the iteration matrix is written as

$$hJ_n = \begin{bmatrix} \alpha_0 I - h\frac{\partial f}{\partial x} & -h\frac{\partial f}{\partial y} \\ h\frac{\partial g}{\partial x} & h\frac{\partial g}{\partial y} \end{bmatrix}.$$

We consider two cases: semi-explicit index one systems, and semi-explicit index two systems with the property that $\partial g/\partial y \equiv 0$. Scaling for index three systems arising in constrained mechanical problems is discussed in [205].

Case I. When the index is one, we have that $\partial g/\partial y$ is nonsingular, so hJ_n is nonsingular as $h \to 0$ if we scale the rows corresponding to the algebraic constraints by $1/h$. Since we are not scaling variables, but only equations, the effect of this scaling should be to improve the accuracy of the solution of the linear system, for *all* variables.

Case II. For this case, we will assume that the index is two, and that $\partial g/\partial y \equiv 0$. By explicitly computing $(hJ_n)^{-1}$ we find that the orders of the blocks of the inverse are

$$\begin{bmatrix} 1 & 1/h \\ 1/h & 1/h^2 \end{bmatrix},$$

where the elements in the first row correspond to x and those in the second row to y. Roundoff errors proportional to u/h and u/h^2 are introduced in x and y, respectively, while solving this unscaled linear system. See Lötstedt and Petzold [205] for a more detailed explanation. If we scale the bottom rows of hJ_n (corresponding to the 'algebraic' constraints) by $1/h$, then the scaled matrix can be written as

$$h\hat{J}_n = \begin{bmatrix} \alpha_0 I - h\frac{\partial f}{\partial x} & -h\frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & 0 \end{bmatrix}$$

With this scaling the roundoff errors are $O(u)$ in x and $O(u/h)$ in y. As $h \to 0$ these errors can begin to dominate the solution y. In this case, the error estimates and convergence tests in an automatic code may experience difficulties due to these growing rounding errors. These difficulties, along with what can be done to minimize their effects, will be described in greater detail later. For now, we merely note that the effect of the proposed scaling is to control the size of the roundoff errors in x which are introduced in solving the linear system. At the same time, the algebraic variables y may contain errors proportional to u/h. However, since the values of y do not affect the state of the system directly (that is, how the system will respond at future times), we may be willing to tolerate much larger errors in y than in x. In any case, this scaling is a significant improvement over the original scaling. For the scaled system, the errors are considerably diminished, and the largest errors are confined to the variables which are in some sense the least important. Painter [198] describes difficulties due to ill conditioning for solving incompressible Navier-Stokes equations and employs essentially the same scaling that we have suggested here to solve the problem. These difficulties are most severe when

an automatic code is using a very small stepsize, as in starting a problem or passing over a discontinuity in some derivative.

We further note that if we are using Gaussian elimination with partial pivoting, we do not need any column scaling. The solution will be the same without this scaling, because it does not affect the choice of pivots [235]. What the analysis shows is that the errors which are due to ill conditioning are concentrated in the algebraic variables y of the system and not in the differential variables x. Thus, we must be particularly careful about using the "algebraic" variables in other tests in the code which might be sensitive to the errors in these variables.

There is a simple way to implement row scaling in a general DAE code, which unfortunately requires a minor change to DASSL. By modifying the argument list for the RES routine to include the stepsize h, the user can scale DELTA inside this subroutine. This way, the scaling costs virtually nothing. An alternative idea is to provide an option to automatically do row equilibration, or to use linear system solvers which perform row scaling, as suggested by Shampine [222] for stiff ODE systems.

Even after scaling, for high index systems there are relatively large errors in some of the algebraic variables. Since these variables do not determine the future state of the system, the errors are in some sense tolerable. For an automatic code we must have some criterion for deciding when to terminate the corrector iteration. It is shown in Petzold and Lötstedt [205] that to maintain the global order of accuracy of the state variables of the system, it is sufficient to terminate the Newton iteration based on the errors of the *scaled* variables for the column scaling which would bring the condition number of the system to $O(1)$. For the index two system where $\partial g/\partial y \equiv 0$, the iteration error of hy, rather than y, would be required to be less than a prescribed tolerance. If y needs to be known very accurately, this strategy is not appropriate.

It remains to examine the integration error tests which are used to control the stepsize in an automatic code. For index one systems and for index two systems where $\partial g/\partial y \equiv 0$, the errors in the algebraic variable y on previous time steps do not directly influence the errors in any of the variables at the current time [205]. Therefore, we can consider deleting these variables from the error estimate in order to promote the smooth operation of a code. Consider first the case of solving semi-explicit index one systems. Suppose, for example, that on one step we make a fairly large mistake by terminating the Newton iteration before it has really converged, and that on this step the value of y that should have been computed has a large error in it, but that based on the incorrect value it passes the error test anyway. If we base the error test on y, then a bigger problem may arise on the next step because the new value of y does not approach the old (incorrect) value of y, so that their difference, and hence the error estimate, does not approach zero as $h \to 0$. For this reason we feel that it is probably wise to leave y out of the error control decisions in semi-explicit index one systems. The main drawback in this strategy is that if we need to know the values of y at interpolated points (between mesh points), then the stepsize should be based in part on the values of y. For index two systems, the stepsize control strategy in an automatic BDF code will fail, for reasons explained in Petzold [200], unless the algebraic variables y are excluded from the integration error test.

Another possibility for BDF codes is to construct an error estimate which automatically 'filters' errors corresponding to the algebraic variables out of the estimate, an idea introduced in [179] and [229] for linear constant coefficient systems of arbitrary index ν. This filtering approach has been studied further for more general index one and two systems in [160,205]. A similar strategy is applied to control the errors in the solution of stiff problems by Sacks-Davis [217,224].

To motivate the estimate of interest here for index one and semi-explicit index two systems, consider applying the implicit Euler method to the system

$$Ax' + Bx = f(t). \tag{5.4.5}$$

Taking one step, we obtain

$$A\left(\frac{x_{n+1} - x_n}{h_{n+1}}\right) + Bx_{n+1} = f(t_{n+1}). \tag{5.4.6}$$

The true solution to (5.4.5) satisfies

$$A\left(\frac{x(t_{n+1}) - x(t_n) - \frac{h_{n+1}^2}{2}x''(\xi)}{h_{n+1}}\right) + Bx(t_{n+1}) = f(t_{n+1}). \tag{5.4.7}$$

Subtracting (5.4.7) from (5.4.6), we obtain

$$A\left(\frac{e_{n+1} - e_n + \frac{h_{n+1}^2}{2}x''(\xi)}{h_{n+1}}\right) + Be_{n+1} = 0, \tag{5.4.8}$$

where $e_{n+1} = x_{n+1} - x(t_{n+1})$. Solving for e_{n+1}, we have

$$e_{n+1} = (A + h_{n+1}B)^{-1}Ae_n - (A + h_{n+1}B)^{-1}A\left(\frac{h_{n+1}^2}{2}\right)x''(\xi).$$

Thus the contribution to the global error from the current step is given by

$$e_{local} = (A + h_{n+1}B)^{-1}A\tau_{lte}, \tag{5.4.9}$$

where τ_{lte} is the local truncation error in the current step. A similar argument for higher order BDF gives (5.4.9) as the contribution to the global error from the current step, where $(A + h_{n+1}B)$ has been replaced by $(A + h_{n+1}\alpha_0 B)$. Here, α_0 is a constant depending on the method. The application of this filtering idea to other methods is not advised without an error expression to

justify it. It can cause asymptotically incorrect estimates if it is carelessly used in combination with other types of methods and local truncation error estimates.

It is useful to reflect on the effects of the filtering for some systems of interest. For index one systems, the filter has the effect of removing the components of the local truncation error corresponding to the algebraic variables from the error estimate because multiplying by the matrix A kills these components. For implicit ODE systems (A nonsingular), the filter tends to the identity as $h_{n+1} \to 0$, giving the usual estimate. For semi-explicit index two systems with $\partial g/\partial y \equiv 0$, the filtering yields an estimate for the kth order BDF which is $O(h^{k+1})$ for the differential variables and $O(h^k)$ for the algebraic variables, thereby correctly reflecting the behavior of the local error. On DAE's with index greater than two, it is quite likely that DASSL (or any other BDF code for DAE's) will fail even when this filter is applied. A stronger filter or a different strategy altogether may be required to control the errors in these high index DAE's. In [205] Petzold and Lötstedt suggest a strategy for error control of index three mechanical systems.

Finally, the estimate (5.4.9) is easily computed in a code like DASSL. The matrix $(A+h_{n+1}B)$ is the iteration matrix used in the solution of the corrector equation. Hence it is already available in factored form. The term $A\tau_{\ell te}$ can be approximated by noting that for $F(t,y,y') = 0$, $A = \partial F/\partial y'$ and $\tau_{\ell te}$ is the usual local truncation error estimate. Then $(\partial F/\partial y')\tau_{\ell te}$ can be approximated by

$$\frac{\partial F}{\partial y'}\tau_{\ell te} \equiv F(t,y,y'+\tau_{\ell te}) - F(t,y,y').$$

Even more simply for semi-explicit systems, the quantity $A\tau_{\ell te}$ is a vector composed of the local truncation errors corresponding to the differential variables x and zeros corresponding to the algebraic variables y. The filter is used in the decision on whether to accept or reject the step. The order selection remains unchanged as it is based on a comparison of relative magnitudes of terms in a Taylor series, rather than on relative magnitudes of local errors. Berzins [17] has recently reported good success in applying part of the filter (namely, the matrix A) also in the order selection process. This approach has the disadvantage that the order selection is not then independent of scalings of the system. This latter point is not an issue for semi-explicit systems.

The filtered estimate is also advantageous for stiff systems which experience frequent steep transients of small magnitude. For example, in the solution of partial differential equations by operator splitting techniques or with adaptive mesh methods, there is a discontinuity and a steep transient after every rezone of the mesh. The filtered estimate allows the code to take relatively large steps through the transient, which would not be allowed using the usual estimate. The filtered estimate has been shown to be quite effective [16,204] in reducing the number of time steps without sacrificing accuracy in experiments with adaptive grid solutions of partial differential equations.

Chapter 6

Applications

6.1 Introduction

In Chapter 1 we briefly discussed a number of scientific disciplines in which DAE's have occurred. Now we consider some of these areas in more depth in order to discuss several issues which are critical to the numerical solution process. For example, determining consistent initial values for a DAE system often requires special problem-dependent techniques. There are often alternative formulations of the DAE for a particular application. The performance of a numerical method can be influenced dramatically by the choice of problem formulation. Our focus in this chapter is to examine these issues as they pertain to specific applications, rather than to give a comprehensive and general description of all the various techniques. Examples arising in the simulation of mechanical systems, electrical systems, trajectory prescribed path control problems, and the discretization of partial differential equations by the method of lines (MOL) will be used to illustrate these ideas and techniques.

Historically, DAE's have been solved indirectly by requiring the system to be reformulated as an explicit ODE system, for which there are many software packages. Often, the reduction of a DAE to an ODE system requires complicated problem-specific manipulations of the equations, incorporation of boundary conditions into the system equations and simplification of the underlying problem. This process slows the study of more complex systems because it often has to be repeated when the model is altered. The ease of setting up and altering DAE models is a major factor in the large amount of current interest in them.

There are often several natural ways to formulate a given application. Different formulations may affect the behavior of the numerical solution process. Sometimes there are choices in both the selection of the equations to be imposed and the variables. In many cases, there is the option of reducing the index of the system being considered. But it is not always apparent whether the reduced index formulation of a problem will yield a better implementation. If the numerical solution is obtained by solving a reduced index problem, the