Solving $0 = F(t, y(t), y'(t))$ in MATLAB

L.F. Shampine
Mathematics Department
Southern Methodist University
Dallas, TX 75275
U.S.A.
lshampin@mail.smu.edu

Abstract

Important algorithms and design decisions of a new code, `ode15i`, for solving $0 = F(t, y(t), y'(t))$ are presented. They were developed to exploit MATLAB, a popular problem solving environment (PSE). Codes widely-used in general scientific computation (GSC) compute consistent initial conditions only for restricted forms of the differential equations. `ode15i` does this for the general problem, a task that is qualitatively different. Unlike popular codes in GSC, `ode15i` saves partial derivatives because this is more efficient in the PSE. A new representation of fixed leading coefficient BDFs is based on the Lagrangian form of polynomial interpolation. Basic computations are both clear and efficient in the PSE when using this form. Some numerical experiments show that `ode15i` is an effective way to solve fully implicit ODEs and DAEs of index 1 in MATLAB.

Keywords: ODEs, DAEs, BDFs, MATLAB

1 Introduction

We investigate the numerical solution of a fully implicit set of first order differential equations

$$0 = F(t, y(t), y'(t)) \tag{1}$$

on an interval $[t_0, t_f]$ with suitable initial conditions at t_0. *Consistent* initial conditions $(t_0, y(t_0), y'(t_0))$ must satisfy the algebraic equations

$$0 = F(t_0, y(t_0), y'(t_0)) \tag{2}$$

For the ordinary differential equations (ODEs) and differential–algebraic equations (DAEs) of index 1 that concern us, an initial value problem (IVP) with smooth function $F(t, y, y')$ and initial conditions that satisfy (2) has a unique solution.

The program `ode15s` [17] solves differential equations of the form

$$M(t, y)y' = f(t, y) \tag{3}$$

It is an effective code that exploits the MATLAB [13] problem solving environment (PSE). The form (3) is quite advantageous when the dependence of the mass matrix $M(t, y)$ on y is weak, but not when it is strong. Also, many problems arise in this form, but others have the general form (1). In this paper we present some of the important algorithms and design decisions of a new program, `ode15i`, for solving problems of the form (1). All were developed with the goal of solving these problems conveniently and efficiently in this widely-used PSE.

DASSL [3] is a popular code for solving differential equations of the form (1) in general scientific computation (GSC). We contrast the approach of this code with the approach that we take to the effective solution of such problems in MATLAB. The first issue is the computation of consistent initial conditions. DASSL has a rather limited capability for this. A later version called DASPK [3] incorporates a substantial improvement in this capability [2]. Even DASPK computes consistent initial conditions for a class of problems that is considerably less general than (1). Wu and White [19] compute consistent initial conditions for the same class of problems with a subroutine for this specific purpose called DAEIS. It is much more flexible than DASPK about which components of the guesses for initial conditions can be fixed. `ode15s` computes consistent initial conditions for problems of the form (3). The user cannot fix any components of the guesses. The algorithms of all these codes make heavy use of the special form they accept. Our aim here is to compute consistent initial conditions for the general problem (1). We let the user fix any components of the guesses that are allowed by the problem itself, but do not require that the user fix any components.

Relative costs in MATLAB are different from those of GSC and the kinds of problems typical of the computing environments are different. After discussing the dominant costs of an integration, we explain why we save partial derivatives in `ode15i` when DASSL does not. Like DASSL we use a fixed leading coefficient implementation of the backward differentiation formulas (BDFs), but DASSL uses a modified divided difference representation and `ode15i` uses a Lagrangian form. After developing this novel form, we argue that it is well suited to the PSE. We further show that the Lagrangian form allows basic computations to be done in a clear way that is efficient in the PSE. It is also convenient for providing capabilities like event location and output in the form of a solution structure that are included in the design of the MATLAB ODE Suite [16].

2 Consistent Initial Conditions

The popular code DASSL [3] integrates ODEs and DAEs of index 1 of the form (1). It has a limited capability for computing consistent initial conditions. Brown et alia [2] made a considerable improvement to this capability in a later version of the code called DASPK [3]. Although it can compute consistent initial conditions for two kinds of DAEs, we give our attention to the more important one of semi-explicit systems. Wu and White [19] also compute consistent initial conditions for semi-explicit DAEs, but they do it in a subroutine separate from the integrator called DAEIS. DAEIS is considerably more flexible than DASPK about which components of the initial conditions can be fixed. `ode15s` computes consistent initial conditions for problems of the form (3). By default this solver asks the user for $y(t_0)$ and determines automatically whether the problem is a DAE. If it is, the code computes a consistent $y'(t_0)$. The user cannot fix any components of the initial conditions. The algorithms of all these codes make heavy use of the special form they accept. Our aim here is to compute consistent initial conditions for the general problem (1). In this we allow users to fix any components of the initial conditions allowed by the problem. This freedom is necessary for the more general task and it is often convenient.

We begin by discussing the initialization problems solved by DASPK and DAEIS because they provide guidance for the general case. Brown et alia [2] consider two types of initialization problem. Initialization Problem I assumes that the d variables $y(t)$ of the problem (1) can be partitioned *a priori* into two sets of variables, p "differential" variables $u(t)$ and $d-p$ "algebraic" variables $v(t)$. After a permutation of the rows, the problem must have the semi-explicit form

$$0 = f(t, u(t), v(t), u'(t)) \tag{4}$$
$$0 = g(t, u(t), v(t)) \tag{5}$$

In the approach of Brown et alia, the user specifies the p differential variables $u(t_0)$. DASPK holds these values fixed and computes consistent $v(t_0)$ and $u'(t_0)$. The derivative $v'(t_0)$ plays no role. Wu and White allow the user to specify algebraic variables when this is more natural and they allow derivatives to be specified. More precisely, the user specifies p components of $u(t_0), v(t_0), u'(t_0)$. DAEIS holds these values fixed and computes the remaining components of a set of consistent initial conditions. Brown et alia also consider an Initialization Problem 2, which is to find $y(t_0)$ in the general DAE (2) when $y'(t_0)$ is given and all its components are fixed.

Like Wu and White, we compute consistent initial conditions in a function `cic` separate from the integrator. We want to solve the algebraic equations (2). Because these equations may not have a unique solution, it is necessary to ask the user for a guess (t_0, y_0, y'_0). In addition to identifying the solution of interest, a good guess can be critical to the success of the numerical scheme for computing the solution. This guess is improved iteratively and we ask the user to specify up to d components of y_0 and y'_0 that are to be held fixed during the iteration. It is often not clear how many components can be held fixed. We do not require that any components of the guess be fixed and recommend that users leave as many free as possible. Correspondingly, the default is that all components are free. In our formulation, the initial conditions are underdetermined, so a fundamental issue is how to resolve this. Our basic principle is to retain as many components of the guess as possible. When there is a choice, our goal is preserve guesses for the components that are customarily fixed when solving DAEs of less general form.

Starting with an initial guess for (t_0, y_0, y_0'), we linearize the equations (2) about the current approximate solution and solve the linear equations for corrections to the guess. In special cases this is Newton's method and convergence is established in a similar way with assumptions usual for Newton's method. In these cases our algorithm is not essentially different from those of [2, 19]. Brown et alia supplement Newton's method with backtracking, Wu and White use damping, and we use a trust region. To reduce the number of linearizations, we iterate twice with a chord method for each iteration with Newton's method (Shamanskii's method [12, §5.4.3]). Rather than discuss further these details, we explain how we resolve a fundamental difference between these other algorithms and our own: For the initialization problems solved by the other algorithms, the linear system for the correction to an iterate has a unique solution. For the general problem that we solve, the corresponding linear system is underdetermined.

We first linearize (2) to obtain

$$F(t_0, y_0, y_0') + F_{y'}\,\delta y_0' + F_y\,\delta y_0 = 0 \tag{6}$$

Unlike DASPK, we assume that both F_y and $F_{y'}$ are available. This is an important design issue that we discuss in §3.1. We eliminate from these equations the components of y_0 and y_0' that are to be held fixed, something easily done with the array operations of Matlab. So as not to complicate the notation unduly, we do not indicate this explicitly in what follows. However, it is important to keep in mind that as a consequence, the matrices of partial derivatives may not be square. Indeed, if all the components of y_0 or y_0' are fixed, one or the other of the matrices is not even present.

We must be alert to the possibility that a user has fixed too many components or fixed a component that cannot be fixed. The discussion earlier of semi-explicit DAEs (4) shows that a user generally cannot fix more than p components. When solving a DAE of the form (1), a user generally cannot fix all d components of y_0. This appears in the program where we try to compute y_0' by solving repeatedly

$$F_{y'}\,\delta y_0' = -F(t_0, y_0, y_0')$$

For a DAE, the matrix $F_{y'}$ does not have full row rank and we are generally not able to solve the linear system. On the other hand, fixing all components of y_0 is typical for ODEs and there is no difficulty then in solving the linear system because $F_{y'}$ is non-singular for an ODE.

If $F_{y'}$ is present, we compute a QR decomposition with column pivoting,

$$F_{y'}\,E = QR$$

With it, we first transform (2) to

$$R\left(E^T \delta y_0'\right) + \left(Q^T F_y\right)\delta y_0 = -Q^T F(t_0, y_0, y_0')$$

and then with some simplification of the notation to

$$Rw' + Sw = d \tag{7}$$

The row rank of R is determined in a way usual for MATLAB, namely by testing which pivots are negligible.

For speed in the PSE it is important to use the built-in functions. The SVD function is an attractive alternative to the QR decomposition with column pivoting. It is slower, but provides a more robust determination of rank. On the other hand, we believe that rank is not ambiguous in the present circumstances because it is a statement about the nature of the DAE. Furthermore, we solve underdetermined systems with the backslash operator. The operator does this computation with a QR decomposition using column pivoting. The integrator `ode15i` is not intended for problems as large as those solved in GSC, but it can solve relatively large problems if the partial derivative matrices are sparse. Because pivoting is necessary for the rank determination, we cannot take advantage of sparsity in the present computation. The same would be true if we were to use an SVD. As a consequence, `cic` cannot compute consistent initial conditions for problems as large those that `ode15i` can integrate. This is unfortunate, but we think that it is of little practical significance.

If R is of full row rank, we set $w = \delta y_0 = 0$. We then solve the triangular system $Rw' = d$ with the fast built-in backslash operator and compute $\delta y_0' = Ew'$. This case corresponds to computing consistent initial conditions for an ODE. It is a case for which it is clear which components of the guess we want to preserve.

When R does not have full row rank, we have a DAE. In this case we write (7) in partitioned form as

$$\begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix} \begin{pmatrix} w_1' \\ w_2' \end{pmatrix} + \begin{pmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \tag{8}$$

The components w_2' play no role in this set of linear equations, so we are free to set them to zero, which we do in order to compute a solution of (2) that retains as many components of y_0' as possible. We then solve the underdetermined block of equations

$$\begin{pmatrix} S_{21} & S_{22} \end{pmatrix} w = d_2 \tag{9}$$

Before actually solving the system, we compute the row rank of the system using a QR decomposition with column pivoting, just as in the first reduction. If the system does not have full row rank, the computation is terminated. If the rank deficiency is no greater than the number of components fixed in the guesses, it is suggested that the user try freeing up this many components. Otherwise, it is suggested that the DAE might have an index greater than 1. We solve (9) with the backslash operator. It is natural to use this built-in function because it is fast, but more important here is that it computes a basic solution. That is, it sets as many components of $w = \delta y_0$ to zero as possible, hence retains as many components of the guess y_0 as possible. (Although this is exactly what we prefer, we did experiment with a minimal norm solution to (9). Computing such solutions is slower and in our limited experiments, convergence of the overall iteration was notably worse.) Once w has been determined, we can use it and $w_2' = 0$ to solve for the components w_1' in the first block of equations

$$R_{11} w_1' = d_1 - S_{11} w_1 - S_{12} w_2$$

using the backslash operator. This completes our solution of the underdetermined system (7).

Brown et alia [2] give a simple example of (1) that is not included in the class of problems for which they compute consistent initial conditions. In slightly different notation the example is

$$\begin{aligned} y_1' + y_2' + g(t, y_1) &= 0 \\ y_2 + h(t) &= 0 \end{aligned}$$

The linearization of these equations is already in the form (8):

$$\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} w_1' \\ w_2' \end{pmatrix} + \begin{pmatrix} g_{y_1} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \end{pmatrix}$$

It is easy to see from the form of these equations that if the user did not fix any components of the guesses, our algorithm would retain the guessed value of $y_2'(0)$ and solve the linear equation for $y_2(0)$ to get $-h(0)$. It would retain the guessed value of $y_1(0)$ and iteratively compute a consistent value for $y_1'(0)$. On the other hand, if the user fixed $y_1(0)$, it would be eliminated from this system and there would be no iteration in the computation of $y_1'(0)$. Finally, if the user fixed $y_2(0)$, the algorithm would use the guessed values for $y_1(0)$ and $y_2'(0)$ and compute a value for $y_1'(0)$ that is consistent in the ODE. Whether the resulting values are a consistent set of initial conditions depends on whether the user assigned the correct value to the fixed component $y_2(0)$. The algorithm seems to be responding in a satisfactory way to each of these tasks.

Wu and White [19] consider the semi-explicit DAE of index 1

$$\begin{aligned} \frac{\rho V}{W} y_1' - \frac{j_1}{F} &= 0 \\ j_1 + j_2 - i_{app} &= 0 \end{aligned}$$

Here

$$\begin{aligned} j_1 &= i_{01} \left[2(1 - y_1) \exp\left(\frac{0.5F}{RT}(y_2 - \phi_{eq,1}) \right) - 2 y_1 \exp\left(-\frac{0.5F}{RT}(y_2 - \phi_{eq,1}) \right) \right] \\ j_2 &= i_{02} \left[\exp\left(\frac{F}{RT}(y_2 - \phi_{eq,2}) \right) - \exp\left(-\frac{F}{RT}(y_2 - \phi_{eq,2}) \right) \right] \end{aligned}$$

and the parameters have the values $F = 96487$, $R = 8.314$, $T = 298.15$, $\phi_{eq,1} = 0.420$, $\phi_{eq,2} = 0.303$, $\rho = 3.4$, $W = 92.7$, $V = 10^{-5}$, $i_{01} = 10^{-4}$, $i_{02} = 10^{-10}$, and $i_{app} = 10^{-5}$. Wu and White state that it is not easy to choose consistent initial values. They estimate that $y_1(0) = 0.05$ and $y_2(0) = 0.38$. They observe that these values are not consistent and many DAE solvers fail when they are used as guesses. Table 1 of their paper presents some results for three popular solvers, DASSL, RADAU5, and LIMEX. Their scheme requires that one of the components be fixed. Table 1 reports a range of values for the other component for which their algorithm produces consistent initial conditions. Our function `cic` succeeds in producing consistent initial conditions without specifying a fixed value. As it turns out, `cic` retains the guessed value of 0.05 for the differential variable $y_1(0)$ and computes the consistent value $y_2(0) = 0.35024$ reported by Wu and White. Because we allow $y_1(0)$ to be fixed, we can experiment with the guess for $y_2(0)$ as Wu and White did. Their scheme has a much bigger interval of convergence than the other codes, namely $-0.974 \leq y_2(0) \leq 1.663$. Our scheme also has a much bigger interval, but not as big as that of Wu and White, namely $-0.3 \leq y_2(0) \leq 0.9$. However, this comparison is reversed when $y_2(0)$ is fixed and the consistent value $y_1(0) = 0.15512$ is computed. This case distinguishes the algorithm of DAEIS from that of DASPK because DASPK does not allow an algebraic variable to have fixed value. Wu and White get convergence for $0.0 \leq y_1(0) \leq 1.0$ and we get convergence for $y_1(0) = 0, \pm 1, \pm 2, \ldots, \pm 10$. When applied to the same kind of problem and used in the same way, our scheme is much like that of Wu and White, so it is not surprising that it would have similar behavior. The important difference in the schemes is that ours applies to the general problem (1) and does not require that any components of the guess be fixed.

Computing consistent initial conditions for problems as general as (1) is difficult. Whether one of the algorithms succeeds depends on the initial guesses and the components that are fixed. Different algorithms can result in different initial conditions. Indeed, the example of Wu and White shows that the same algorithm can result in different initial conditions when applied in a different way. Fig. 1 of their paper shows that the solutions determined by these two sets of initial conditions differ considerably. It is not clear how accurately we should compute the initial conditions and it is not easy to be sure that the ones computed are this accurate. As a supplement to its convergence test, `cic` returns the Euclidean norm of the residual of the computed initial conditions in (2). We have preferred a separate function for the computation of consistent initial conditions because it facilitates interaction with the user, both in securing convergence and in deciding whether the initial conditions found are acceptable.

3 BDFs in Matlab

DASPK and `ode15i` differ in important ways that are explained in this section. We begin with a discussion of evaluating implicit formulas. In this we explain why we chose to save partial derivatives in `ode15i` when DASPK does not. At the same time we discuss a major difference in the designs of the two solvers. Subsequently we discuss a fixed leading coefficient formulation of the backward differentiation formulas. DASPK uses a modified divided difference representation, but we have preferred a Lagrangian form. After developing this novel form, we argue that it is well suited to the MATLAB PSE. In particular, we show that it can be implemented in a simple and efficient way. It is well-known that BDF codes have stability difficulties for certain kinds of problems. An improvement found in the order selection algorithm of DASSL allows it to solve such problems effectively. We have followed DASSL in this regard. An example shows that despite other important differences in the algorithms of the two solvers, we were able to obtain comparable performance from `ode15i` for such problems.

3.1 Implicit Formulas

In this section we study the efficient evaluation of backward differentiation formulas (BDFs) when solving (1). The details are simple for the backward Euler formula (BDF1) and they show what happens with all the BDFs. On reaching $y_n \approx y(t_n)$, BDF1 defines $y_{n+1} \approx y(t_n + h)$ as the solution of

$$0 = F\left(t_{n+1}, y_{n+1}, \frac{y_{n+1} - y_n}{h}\right) \tag{10}$$

These algebraic equations are solved iteratively. An iterate $y_{n+1}^{[m]}$ is improved by writing the next iterate as $y_{n+1}^{[m+1]} = y_{n+1}^{[m]} + \delta$, approximating (10) by the linear system of equations

$$F\left(t_{n+1}, y_{n+1}^{[m]}, \frac{y_{n+1}^{[m]} - y_n}{h}\right) + \left(\frac{1}{h} F_{y'} + F_y\right)\delta = 0$$

and solving them for the correction δ. In general the iteration matrix has the form

$$\frac{\alpha}{h} F_{y'} + F_y \tag{11}$$

where α is a constant characteristic of the formula. Forming, factoring, and storing this matrix is a very important part of the cost of integrating (1). Generally the partial derivatives vary slowly, a fact that is key to the efficient evaluation of the BDFs. The partial derivatives are evaluated at most once in the computation of y_{n+1}, making the iteration a simplified Newton (chord) method instead of a Newton method. This reduces the rate of convergence to linear, but it reduces the number of partial derivatives and factorizations so much that it is a bargain. Step size and order selection algorithms are biased towards constant step size and order so that the solvers can use the same factored iteration matrix for several steps. All the popular codes proceed as outlined. Where they differ is in what they do when the step size and/or order is changed, i.e., when the coefficient α/h of (11) is changed.

By default DASSL approximates the iteration matrix by finite differences. If it is banded, the code can use this fact to reduce greatly the storage, the cost of approximating the matrix, and the cost of factoring the matrix. A user can provide a subroutine that evaluates (11) for given (t, y, y') and α. DASSL was designed to solve large problems. Accordingly, it tries to minimize the storage it uses. A large reduction in storage is obtained by overwriting the iteration matrix with its factorization. However, this means that the solver must compute new partial derivatives whenever a new iteration matrix is formed. Often they are unnecessary because the change in the iteration matrix is mostly due to the change in α/h. Recognizing this, if the change in α/h is not too big, DASSL continues to use the factorization of the old iteration matrix and compensates for the change in the coefficient by relaxation [4]. In §2 we assumed that both F_y and $F_{y'}$ are available for computing consistent initial conditions. They are available in DAEIS, which approximates them numerically. They are not available in DASPK, which either approximates numerically the iteration matrix or asks the user to supply a subroutine for evaluating it. Brown et alia [2] manage to work around this for the special kinds of problems for which they compute initial conditions, but it would clearly be better to have both partial derivatives available.

The context and the computing environment for `ode15i` are quite different. For one thing, linear algebra is relatively fast. For another, the typical problem is of only modest size, so `ode15i` does not aim to solve problems as big as those solved by DASSL. Because we compute consistent initial conditions for problems of the general form (1), we ask users for both F_y and $F_{y'}$. As a convenience for users, by default we approximate partial derivatives numerically. This is relatively expensive in the PSE. To evaluate the BDFs efficiently in these circumstances, `ode15i` saves partial derivatives. When the coefficient α/h changes, an iteration matrix is formed using the saved partial derivatives and the new coefficient. It is then factored. New partial derivatives are formed only when the rate of convergence is inadequate. We experimented with relaxation as in DASSL, but found it to be counterproductive because the partial derivatives are generally not current and linear algebra is fast in the PSE. Generally the partial derivatives change slowly and forming them is sufficiently expensive in MATLAB that reusing them reduces considerably the cost of solving the IVP. However, we show by example in §4 that this is not always the case.

By default `ode15i` approximates partial derivatives numerically. Optionally the user can supply a function for evaluating F_y and $F_{y'}$. If it returns an empty array for a partial derivative, the solver interprets this as an instruction to approximate that partial derivative numerically. We have made it convenient to evaluate analytically just one partial derivative because often an analytical expression for one is readily available. For instance, if the problem has the form (3), the partial derivative $F_{y'}$ is just the mass matrix $M(t, y)$.

Our approach spends storage to buy generality in the computation of consistent initial conditions and efficiency in the integration. It increases the storage considerably, but this is not very important for the problems that `ode15i` aims to solve. Besides, storage is handled differently for relatively large problems. To deal with large problems, DASSL provides for banded iteration matrices. `ode15i` goes much further by providing for general sparse matrices. In particular, it uses a generalization [5] to general sparse matrices of the scheme DASSL uses to reduce the cost of forming partial derivatives for banded matrices. `ode15i` does not require that F_y and $F_{y'}$ have the same structure. An equation of the form $F(t, y, y') = y' - f(t, y) = 0$ makes clear the advantages of saving $F_{y'} = I$ as a sparse matrix and allowing a different structure for $F_y = -f_y$.

3.2 Fixed Leading Coefficient Formulas

We begin by deriving the BDF of order k for solving $y' = f(t, y)$ when the step size is a constant h. We interpolate approximations y^*_{n+1-j} to $y(t_{n+1} - jh)$ for $j = 0, 1, \ldots, k$ with a polynomial $P(t)$ and then require that

$$P'(t_{n+1}) = f(t_{n+1}, P(t_{n+1})) \tag{12}$$

In general the Lagrangian form of a polynomial $R(t)$ interpolating values Y_{n+1-j} at nodes t_{n+1-j} for $j = 0, 1, \ldots, k$ is

$$R(t) = \sum_{j=0}^{k} Y_{n+1-j} \prod_{\substack{i=0 \\ i \neq j}}^{k} \left(\frac{t - t_{n+1-i}}{t_{n+1-j} - t_{n+1-i}} \right) \tag{13}$$

If we multiply the collocation equation (12) by h and use the representation (13) for $P(t)$, we obtain the BDF of order k as an implicit linear multistep formula,

$$\sum_{j=0}^{k} \alpha_j \, y^*_{n+1-j} - h f(t_{n+1}, y^*_{n+1}) = 0 \tag{14}$$

The local truncation error τ of the formula is the amount by which a smooth solution of the ODE fails to satisfy the formula. To determine it, suppose that the polynomial interpolates solution values $y(t_{n+1-j})$ rather than approximations y^*_{n+1-j}. Then

$$\tau = h\, P'(t_{n+1}) - h\, f(t_{n+1}, P(t_{n+1})) = h\,(P'(t_{n+1}) - y'(t_{n+1}))$$

Using a standard result about the error of numerical differentiation [9, p. 289], we first obtain

$$\tau = -h \, \frac{y^{(k+1)}(\eta)}{(k+1)!} \prod_{j=1}^{k} (t_{n+1} - (t_{n+1} - j\, h))$$

and then

$$\tau = -\frac{1}{k+1} \, y^{(k+1)}(t_{n+1}) \, h^{k+1} + h.o.t. \tag{15}$$

where *h.o.t.* is "higher order terms". Put differently, $y(t)$ satisfies

$$\sum_{j=0}^{k} \alpha_j \, y(t_{n+1} - jh) - h y'(t_{n+1}) = \tau$$

The same approach can be used for a general mesh to get a formula of the same form with $h = t_{n+1} - t_n$ and coefficients α_j that depend on the relative mesh spacing. The step size and order algorithms are biased towards holding the step size and order constant. The difficulty with a general mesh is that after changing the step size or order on one step, the coefficient α_0/h changes for each of the succeeding steps until the mesh spacing is constant in the span of the formula. The codes of Gear and Krogh cited earlier avoid these costs by a quasi-constant step size implementation. On changing the step size from h to H, these codes use interpolation to compute approximate solutions at $t_n - H, t_n - 2H, \ldots$ and then apply the constant step size formula to this new data. In this way the effects of a change of step size on the iteration matrix are confined to a single step. Unfortunately, this approach is not as stable as a fully variable step size implementation. There are a few solvers that accept the cost of the more stable implementation. A notable example is the solver that Brayton et alia [1] develop using a Lagrangian representation of the underlying polynomial. The fixed leading coefficient implementation of Jackson and Sacks–Davis [10] is a compromise. DASSL uses a fixed leading coefficient implementation based on a modified divided difference representation of the underlying polynomial. We now develop the equivalent using a Lagrangian representation.

When the step size varies, we do not have approximate solutions at the equally spaced mesh points of the formula (14). In a fixed leading coefficient implementation, these approximate solutions are obtained by interpolating the accepted numerical solutions $y_{n+1-i} \approx y(t_{n+1-i})$ for $i = 1, \ldots, k+1$ with a polynomial $Q(t)$. The implicit formula (14) for y_{n+1} is then evaluated with approximate solutions $y^*_{n+1-j} = Q(t_{n+1} - jh)$ for $j = 1, \ldots, k$. Just as with the quasi-constant step size implementation, the coefficient of y_{n+1} does not depend on the mesh spacing, hence the name "fixed leading coefficient." On the other hand, interpolated values are formed at every step using the approximations on the actual mesh, not just the step where the step size is changed. To work out the local truncation error lte of this formula, suppose that $Q(t)$ interpolates $y(t_{n+1-i})$ for $i = 1, \ldots, k+1$. Then

$$\begin{aligned} lte &= \left(\alpha_0 y(t_{n+1}) + \sum_{j=1}^{k} \alpha_j \, Q(t_{n+1} - jh) \right) - hy'(t_{n+1}) \\ &= \tau + \sum_{j=1}^{k} \alpha_j \left[Q(t_{n+1} - jh) - y(t_{n+1} - jh) \right] \end{aligned}$$

The term in brackets is an interpolation error. Because we interpolate $y(t_n)$, the first term is zero. For the same reason, more terms become zero as we take more steps of constant size h. After sufficiently many steps of size h, we are working with the constant step size formula and $lte = \tau$. A standard result about the error of interpolation [9, p. 190] states that

$$\begin{aligned} Q(t_{n+1} - jh) - y(t_{n+1} - jh) &= -\frac{y^{(k+1)}(\xi_j)}{(k+1)!} \prod_{i=1}^{k+1} ((t_{n+1} - jh) - t_{n+1-i}) \\ &= -y^{(k+1)}(t_{n+1})\, h^{k+1}\, r_j + h.o.t \end{aligned}$$

where

$$r_j = \prod_{i=1}^{k+1} \frac{(t_{n+1} - jh) - t_{n+1-i}}{ih} \tag{16}$$

A little manipulation then provides an expression for the local truncation error,

$$lte = -\left[\frac{1}{k+1} + \sum_{j=2}^{k} \alpha_j\, r_j\right] y^{(k+1)}(t_{n+1})\, h^{k+1} + h.o.t. \tag{17}$$

that shows clearly the effect of varying the step size.

3.3 Implementation

Now we consider the efficient implementation of fixed leading coefficient BDFs for the fully implicit system (1). First let us recall how the method is formulated in terms of interpolating polynomials and then see how to apply it to the solution of a fully implicit system. The polynomial $Q(t)$ that interpolates y_{n+1-j} for $j = 1, \ldots, k+1$ is evaluated to obtain approximate solutions y^*_{n+1-j} at $t_{n+1} - jh$ for $j = 2, \ldots, k+1$. The polynomial $P(t)$ interpolates y_{n+1}, y_n, and y^*_{n+1-j} for $j = 2, \ldots, k$. If we let $y'_{n+1} = P'(t_{n+1})$, the collocation equation (12) that determines y_{n+1} is $y'_{n+1} = f(t_{n+1}, y_{n+1})$. The corresponding equation for the fully implicit system (1) is clearly

$$0 = F(t_{n+1}, y_{n+1}, y'_{n+1}) \tag{18}$$

This is a nonlinear algebraic equation for y_{n+1}, which is made clear by the relationship

$$y'_{n+1} = \frac{\alpha_0}{h}\, y_{n+1} + \frac{\alpha_1}{h}\, y_n + \sum_{j=2}^{k} \frac{\alpha_j}{h}\, y^*_{n+1-j} \tag{19}$$

that is immediate from (14).

The algebraic equations (18) are solved iteratively as described in §3.1. An iterate $y^{[m+1]}_{n+1}$ is computed as a correction δ to the previous iterate

$$y^{[m+1]}_{n+1} = y^{[m]}_{n+1} + \delta$$

From (19) we see that this implies the correction

$$y'^{[m+1]}_{n+1} = y'^{[m]}_{n+1} + \frac{\alpha_0}{h}\, \delta$$

to the approximate derivative. Linearizing about the previous iterate, the correction is computed as the solution of the system of linear algebraic equations

$$\left(F_y + \frac{\alpha_0}{h} F_{y'}\right) \delta = -F(t_{n+1}, y_{n+1}^{[m]}, y_{n+1}^{\prime\,[m]}) \tag{20}$$

It is important for the efficiency of this process to make a good guess for y_{n+1}. It is natural and effective to use

$$y_{n+1}^{[0]} = Q(t_{n+1}), \qquad y_{n+1}^{\prime[0]} = Q'(t_{n+1})$$

for this purpose. There is another reason for this choice. If we proceed as outlined, it is not necessary actually to compute the approximations y_{n+1-j}^* nor to use the constant coefficient formula (14). Instead we can predict the solution and its derivative using $Q(t)$ and correct these values by solving (20). All we need from the constant step formula is a little table of the values α_0 for the orders $1, \ldots, 5$ used by the solver.

Evaluating an interpolating polynomial is a basic computation in `ode15i`: We predict y_{n+1} in this way. All the solvers of the MATLAB ODE Suite, and correspondingly `ode15i`, provide for output at specific points. This is done efficiently by stepping past an output point and obtaining an approximate solution there by interpolation. The solvers also provide for locating events, a capability that depends on interpolation. Typically solutions computed with `ode15i` are studied graphically. Standard output from the MATLAB solvers is a mesh and the approximate solution on this mesh. If the mesh chosen by the solver does not provide a smooth graph, interpolation can be used to get the additional approximations needed for this purpose. All the solvers provide for output in the form of a structure. An auxiliary function `deval` is used to evaluate the solution inexpensively anywhere in the interval of integration. The solution structure contains the information that `deval` needs for interpolating the solution.

We have preferred the Lagrangian form of the interpolating polynomial because it is attractive from a conceptual point of view and is well-suited to the MATLAB PSE. In this PSE array operations are relatively inexpensive. `ode15i` holds the local mesh and solution on this mesh in arrays. Specifically, on reaching t_n, the mesh point t_{n-j} is held in `mesh(j+1)` for $j = 0, \ldots, k$ and the vector y_{n-j} is held in `meshsol(:,j+1)`. To interpolate at each point of an array `tintrp`, it is both convenient and fast to calculate each coefficient of (13) for all these arguments at one time using array operations. All the interpolated values are then computed in a single multiplication of the matrix `meshsol` and the matrix of coefficients. For output in the form of a solution structure, we just need the usual mesh and solution on the mesh plus an integer for each step that tells `deval` how many points to interpolate. Other convenient aspects of the Lagrangian form will be seen in the next section.

3.4 Stability and Order Selection

There are two main approaches to order and step size selection pioneered by Gear's DIFSUB [7] and Krogh's DVDQ [11]. DIFSUB estimates the step size that might be used at several orders and selects the order with the largest step size. This scheme is used by `ode15s`. DVDQ first selects the order and then the step size for this order. Shampine and Gordon [15] provide details of the approach for their Adams code that appears in MATLAB as `ode113`. Brenan et alia [3, p. 126] state that the strategy for selecting the order in DASSL is nearly identical to that of [15]. There is one important difference. The idea of Krogh's approach is to work with a step size and order for which successive terms in a Taylor series expansion of the solution decrease in size. A solver tests this by estimating the error that would occur if the current step size were used with several different orders. The innovation of DASSL is to estimate and compare the norms of successive scaled derivatives

$$y^{(k)}(t_{n+1})\, h^k \tag{21}$$

The error of the formula of order $k-1$ is a multiple of the norm of the scaled derivative (21), so the order selection schemes are closely related. With one exception, they behave much the same in practice. Brenan et alia [3, p. 127] point out a well-known difficulty with order selection algorithms when solving certain kinds of stiff IVPs with BDFs. If the local Jacobian has eigenvalues that are quite close to the imaginary axis, the BDFs of low order are stable for all step sizes h, but the higher order formulas are not. The difficulty is that the order selection algorithm might choose an order that suffers from a stability restriction when a lower order would not. DASSL's order selection scheme exploits the observation of Skelboe [18] that the instability of the higher order BDFs for these problems is revealed in differences of the numerical solution. More specifically, instability causes the differences to increase in size with the order of the difference. Because of this phenomenon an order selection scheme based on comparing scaled derivatives can recognize the advantages of a lower order formula and lower the order to the point that a stable formula is used.

A discussion of the difficulty due to eigenvalues near the imaginary axis and a numerical example are found in [14, p. 387]. Because `ode15s` uses the Gear approach to order selection, we can see the difficulty by solving the example problem with this code. The problem has the form $y' = Jy$ for a constant matrix J. The BDFs are an option in `ode15s` that we use for this experiment. The maximum order is an option and we begin with its default of 5. It is natural to supply an analytical Jacobian for this problem. With these options and default tolerances, `ode15s` solves the problem with 7238 successful steps and 93 failed attempts. This solver evaluates Jacobians only when necessary and here is successful in recognizing that once is enough. When the maximum order is reduced to 3, the code solves the problem with 725 successful steps and no failed attempts, again evaluating only one Jacobian. If the order selection algorithm were performing properly, reducing the maximum order allowed would not increase dramatically the efficiency of the integration. With the same options and a maximum order of 5, `ode15i` solves this problem posed as a fully implicit system in 657 successful steps and 3 failed attempts. This code also recognizes that one set of partial derivatives is enough. Clearly the approach to selecting order in DASSL is much more effective for this problem. DASSL and `ode15i` can be compared only in a rough way. For one thing, DASSL uses an RMS norm, which is a Euclidean norm divided by the square root of the number of equations. It is not important that `ode15i` uses a maximum norm, but it is necessary to adjust the tolerances to account for the factor of the RMS norm. When this is done, DASSL solves the problem in 662 successful steps and 8 error failures. In this it evaluates the partial derivatives 8 times.

The order and step size selection schemes of `ode15i` are nearly identical to those of DASSL. On the other hand, the way these schemes are implemented differ greatly, a matter we take up now. Fornberg [6] has developed efficient recursions for the computation of the coefficients of the Lagrangian form of the interpolating polynomial (13), as well those for all derivatives up to a specified order. `ode15i` uses a translation of Fornberg's WEIGHTS1 program for this computation called `weights`. It is important to appreciate that the solvers use orders that range only from 1 to 5. In Lagrangian form, the coefficients depend only on the mesh points in the span of the formula and there are few of them, so this computation is inexpensive. The number of equations could be large, but we can evaluate the formulas with matrix-vector multiplications that are fast in the PSE. After computing a tentative solution y_{n+1} at t_{n+1} using the formula of order k, we need to compute a weighted norm of an approximation to the scaled derivative (21). In the solver we call the new mesh point `tnew` and the tentative solution there, `ynew`. The coefficients for computing all derivatives through k of the polynomial interpolating at t_{n+1-j} for $j = 0, \ldots, k$ are obtained by

```
c = weights([tnew mesh(1:k)],tnew,k);
```

The coefficients for the derivative of order k are returned as `c(:,k+1)`. Notice that we extend temporarily the array holding the mesh points to include the tentative mesh point `tnew`. We similarly extend temporarily the array holding the solution and approximate the derivative of order k by a matrix–vector multiplication:

```
sderk = norm(([ynew meshsol(:,1:k)] * c(:,k+1)) .* invwt,inf)...
        * absh^k;
```

In this we also multiply the vector of approximate derivatives by the weight vector `invwt`, compute the maximum norm, and multiply the expression by the appropriate power of the magnitude of the step size. This is an efficient computation in the PSE because array operations are fast and the built-in function `norm` is fast. In a *very* rough way, each line of MATLAB code costs the same, so this is a clear and quite efficient computation of this quantity. The step size selection algorithm has a strong bias towards constant step size, so it is often the case that the step size is constant in the span of the local mesh. Indeed, the order selection algorithm considers raising the order only when this is true. A small reduction in overhead can be obtained by computing in advance and storing in the solver the coefficients for this special, but common, situation.

An identity provides an alternative way of estimating the scaled derivative that appears in (17). Recall that $Q(t)$ is the polynomial interpolating y_{n+1-j}

for $j = 1, \dots, k+1$. Further, the predicted solution $y^{[0]}_{n+1} = Q(t_{n+1})$. Now let $S(t)$ be the polynomial interpolating the same values as $Q(t)$ and the additional value y_{n+1}. Standard results about interpolation with divided differences [9, §6.1] state first that

$$S(t) = Q(t) + [y_{n+1}, y_n, \dots, y_{n-k}] \prod_{j=1}^{k+1} (t - t_{n+1-j})$$

and then that

$$S(t_{n+1}) = Q(t_{n+1}) + \frac{y^{(k+1)}(\eta)}{(k+1)!} \prod_{j=1}^{k+1} (t_{n+1} - t_{n+1-j})$$

The values of the interpolating polynomials at t_{n+1} and a little manipulation first show that

$$y_{n+1} - y^{[0]}_{n+1} = y^{(k+1)}(t_{n+1})\, h^{k+1} \prod_{j=1}^{k+1} \left(\frac{t_{n+1} - t_{n+1-j}}{jh} \right) + h.o.t.$$

and then that

$$y^{(k+1)}(t_{n+1})\, h^{k+1} = (y_{n+1} - y^{[0]}_{n+1}) \prod_{j=1}^{k+1} \left(\frac{jh}{t_{n+1} - t_{n+1-j}} \right) + h.o.t.$$

The weighted maximum norm of this approximation to the scaled derivative is computed with

```
sderkp1 = norm((ynew - ypred) .* invwt,inf) * ...
          abs(prod((absh * [1:k+1]) ./ [tnew - mesh(1:k+1)]));
```

By using the fast built-in function `prod` and array operations, the whole computation is done efficiently in one line of code.

4 Additional Numerical Examples

The MATLAB demonstration code `batonode` illustrates solving ODEs of the form (3). A natural formulation of the motion of a thrown baton leads to the ODEs

$$\begin{aligned}
0 &= y_1' - y_2 \\
0 &= (m_1 + m_2) y_2' - m_2 L \sin y_5\, y_6' - m_2 L y_6^2 \cos y_5 \\
0 &= y_3' - y_4 \\
0 &= (m_1 + m_2) y_4' + m_2 L \cos y_5\, y_6' - m_2 L y_6^2 \sin y_5 + (m_1 + m_2) g \\
0 &= y_5' - y_6 \\
0 &= -L \sin y_5\, y_2' + L \cos y_5\, y_4' + L^2\, y_6' + g L \cos y_5
\end{aligned}$$

In `batonode` the parameters have the values $m_1 = 0.1, m_2 = 0.1, L = 1, g = 9.81$. The initial values are $y_0 = (0, 4, 2, 20, -\pi/2, 2)$ and the interval of integration is $[0, 4]$. One reason we consider this IVP is that it is not stiff. Another is that as an ODE, it represents a special case for the computation of consistent initial conditions. With all components of the guesses for $y(0)$ and $y'(0)$ left free, we would like `cic` to hold fixed the initial values $y(0)$ and compute consistent values for $y'(0)$. For equations of the form (3), it is natural to supply an analytical expression for $F_{y'} = M(t, y)$ and convenient to approximate F_y numerically. With this, a nominal guess of $y'(0) = 0$, and default tolerances, `cic` held fixed all components of y_0 and computed y_0' accurate to full precision. These consistent initial conditions were used in our experiments.

With default tolerances `ode15i` solved this IVP in 75 successful steps and 31 failed attempts. In this it formed 25 partial derivatives and evaluated the equations 379 times. With tolerances adjusted for its RMS norm, DASPK solved the IVP in 66 successful steps and 2 failed attempts. It formed 26 partial derivatives and evaluated the equations 259 times. As expected, `ode15i` had more failed steps than DASPK because of its scheme for deciding when to form new partial derivatives. For this IVP, saving partial derivatives was not helpful.

We also solved the baton problem with `ode15s`. As with the other solvers, we used its option for specifying a consistent y_0'. The default methods of this solver are the NDFs [16], but there is an option of using the BDFs like `ode15i` and DASPK. We used default tolerances and the default numerical partial derivatives in our experiments. When using the NDFs, this solver took 58 successful steps and had 24 failed attempts. There were 17 partial derivatives formed and the equations were evaluated 260 times. When using the BDFs, the code took 56 successful steps and had 23 failed attempts. There were 18 partial derivatives formed and the equations were evaluated 269 times.

A one transistor amplifier circuit is presented in Fig. 1.3 of E. Hairer and G. Wanner [8] and modeled by five differential equations. The equations arise naturally in the form (3) with a constant mass matrix

$$M = \begin{pmatrix} -C_1 & C_1 & & & \\ C_1 & -C_1 & & & \\ & & -C_2 & & \\ & & & -C_3 & C_3 \\ & & & C_3 & -C_3 \end{pmatrix}$$

Here $C_k = k \times 10^{-6}$ for $k = 1, 2, 3$. This matrix is of rank 3, so the system is a DAE. DASPK and DAEIS do not compute consistent initial conditions for such problems. It is easy enough to change variables to get a system in the (permuted) semi-explicit form required by those codes, but `cic` and `ode15s` can be applied directly to compute consistent initial conditions.

From the form of this problem we see that for any y_0, a consistent y_0' is determined only up to a vector in the null space of M, i.e., up to a vector of the form $(\alpha, \alpha, 0, \beta, \beta)$ for arbitrary α and β. Hairer and Wanner work out consistent initial conditions analytically. For the computations resulting in Fig.

1.4 of [8], they take $y_0 = (0, 3, 3, 6, 0)$. As in the preceding example, it is convenient when using `cic` and `ode15i` to supply $F_{y'}$ analytically and have the program approximate F_y numerically. In the absence of other information, we guess $y'_0 = 0$ with the aim of getting a consistent initial derivative that is not large. With all components of the guesses left free and default tolerances, `cic` held fixed all components of y_0 and returned $y'_0 = (0, 0, -500/3, 0, 0)$. The program returns the Euclidean norm of the residual in (3) of the computed initial conditions. In this case the residual was zero. When the guess was changed to $y'_0 = (1, 1, 1, 1, 1)$, `cic` returned $y'_0 = (1, 1, -500/3, 1, 1)$. We see here the effects of the algorithm holding fixed as many components of the guess as possible and the null space of M. `ode15s` uses a different algorithm [17] for computing consistent initial conditions that is tailored to the form (3). It holds fixed all components of y_0 and does not ask for a guess for y'_0. A goal of its algorithm is to find a consistent y'_0 that is not large. The solver does not return this vector to the user, but the source code is available, so it is easy enough to determine that it computed $y'_0 = (0, 0, -500/3, 0, 0)$.

With options set as in computing consistent initial conditions, `ode15i` solved the IVP in 3992 successful steps and 1616 failed attempts. In this it formed 115 partial derivatives and evaluated the equations 10852 times. DASPK solved the IVP in 3142 successful steps with 1141 failed attempts. It formed 2223 partial derivatives and evaluated the equations 18667 times. Saving partial derivatives has reduced the number of partial derivatives formed by a factor of nearly 20. The number of failed steps increased, but this was a small cost for saving so many partial derivatives.

The MATLAB demonstration program `amp1dae` solves this problem on a shorter interval to illustrate the solution of DAEs of index 1 of the form (3) with `ode15s`. The solver takes important advantage of the fact that the mass matrix is constant. As with the other example of this section, we used the option of supplying consistent $y'(0)$. We also used default tolerances and the default of approximating partial derivatives numerically. With the default NDFs, the IVP was solved in 1326 successful steps with 471 failed attempts. In this it formed 184 partial derivatives and evaluated the equations 4608 times. When using the BDFs, the solution required 1749 successful steps and had 542 failed attempts. There were 170 partial derivatives formed and the equations were evaluated 5207 times. With either family of formulas, `ode15s` is significantly more efficient than `ode15i` and DASPK for this particular IVP. Critical to this is the restricted form of the differential equations and especially the fact that the mass matrix is constant.

5 Conclusions

Computing consistent initial conditions for differential equations as general as (1) is difficult, but in our experience `cic` has performed rather well. The capability of fixing selected components of the guesses for consistent initial conditions can be useful. In our experience, fixing too many components and fixing the wrong components are not rare, so we recommend that users fix no more than necessary. Indeed, the capability of leaving free components that other schemes hold fixed can be quite useful.

Our experience is that `ode15i` and DASPK solve a problem with a comparable number of successful steps because they use the same formulas and their step size and order algorithms are nearly the same. `ode15i` does not form as many partial derivatives, often not nearly as many, but it has more step failures because that is the way that it recognizes when it needs new partial derivatives. Basing the formulas of `ode15i` on the Lagrangian form of the underlying interpolating polynomials leads to simple expressions that are evaluated efficiently in MATLAB. We have also found the form to be quite convenient in endowing `ode15i` with the ability to locate events and the ability to evaluate the numerical solution efficiently anywhere in the interval of integration.

6 Acknowledgement

The programs `cic` and `ode15i` were developed in collaboration with J. Kierzenka of The MathWorks, Inc. His advice about the algorithms presented here is much appreciated.

References

[1] R.K. Brayton, F.G. Gustavson, and G.D. Hachtel, A new efficient algorithm for solving differential-algebraic systems using implicit backward differentiation formulas, Proc. of the IEEE, 60 (1972), pp. 98–108.

[2] P.N. Brown, A.C. Hindmarsh, and L.R. Petzold, Consistent initial condition calculation for differential-algebraic systems, SIAM J. Sci. Comput., 19 (1998), pp. 1495–1512.

[3] K.E. Brenan, S.L. Campbell, and L.R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, SIAM Classics in Applied Mathematics **14**, SIAM, Philadelphia, 1996.

[4] K. Burrage, J.C. Butcher, and F.H. Chipman, An implementation of singly implicit Runge-Kutta methods, BIT, 20 (1980), pp. 326–340.

[5] A.R. Curtis, M.J.D. Powell, and J.K. Reid, On the estimation of sparse Jacobian matrices, *J. Inst. Math. Appl.*, 13 (1974), pp. 117–119.

[6] B. Fornberg, *A Practical Guide to Pseudospectral Methods*, Cambridge University Press, New York, 1998.

[7] C.W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[8] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II*, 2nd ed., Springer, Berlin, 1996.

[9] E. Isaacson and H.B. Keller, *Analysis of Numerical Methods*, Wiley, New York, 1966.

[10] K.R. Jackson and R. Sacks–Davis, An alternative implementation of variable step-size multistep formulas for stiff ODEs, ACM Trans. Math. Software, 6 (1980), pp. 295–318.

[11] F.T. Krogh, VODQ/SVDQ/DVDQ – Variable order integrators for the numerical solution of ordinary differential equations, TU Doc. No. CP-2308, NPO-11643, Jet Propulsion Laboratory, Pasadena, CA, 1969.

[12] C.T. Kelley, Iterative Methods for Linear and Nonlinear Equations, SIAM, Philadelphia, 1995.

[13] MATLAB *6*, The MathWorks, Inc., 3 Apple Hill Dr., Natick, MA 01760, 2000.

[14] L.F. Shampine, *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.

[15] L.F. Shampine and M.K. Gordon, *Computer Solution of Ordinary Differential Equations*, W.H. Freeman, San Francisco, 1975.

[16] L.F. Shampine and M.W. Reichelt, The MATLAB ODE suite, *SIAM J. Sci. Comput.*, 18 (1997), pp. 1–22.

[17] L.F. Shampine, M.W. Reichelt, and J.A. Kierzenka, Solving index-1 DAEs in MATLAB and Simulink, *SIAM Review*, 41 (1999), pp. 538–552.

[18] S. Skelboe, The control of order and steplength for backward differentiation methods, BIT, 17 (1977), pp. 91–107.

[19] B. Wu and R.E. White, An initialization subroutine for DAEs solvers: DAEIS, Computers and Chemical Engineering, 25 (2001), pp. 301–311.