SOLVING INDEX 1 DAES IN MATLAB AND SIMULINK*

LAWRENCE F. SHAMPINE†, MARK W. REICHELT‡, AND JACEK A. KIERZENKA§

Abstract. This paper describes mathematical and software developments needed for the effective solution of DAEs of index 1 in the integrated computing environment MATLAB and the dynamic simulation package SIMULINK. The developments are applicable to other problem solving environments and some are applicable to general scientific computation.

Key words. differential-algebraic equations, DAE, ordinary differential equations, ODE, PSE, stiff systems, BDF, Gear method, software

AMS subject classifications. 65L05, 65Y99, 34A99

1. Introduction.

Problem solving environments, PSEs, like MATLAB [11] and SIMULINK [11] need capabilities for the solution of differential-algebraic equations, DAEs. Solving DAEs in PSEs differs in important ways from solving them in general scientific computation. PSEs are in such wide use that it is worth studying algorithms and software appropriate for them. Although we study here the solution of DAEs in MATLAB and SIMULINK, our mathematical and software developments are applicable to other PSEs and some are applicable to general scientific computation.

The `ode15s` code of the MATLAB ODE Suite [16] is based on a variant of the backward differentiation formulas, BDFs, called NDFs. It was developed to integrate stiff ordinary differential equations, ODEs, of the form

$$\boldsymbol{M}(t)\boldsymbol{y}' = \boldsymbol{f}(t, \boldsymbol{y}) \tag{1.1}$$

When the mass matrix $\boldsymbol{M}(t)$ is singular, this is a DAE rather than an ODE. Computation of a new step with an NDF or BDF does not require $\boldsymbol{M}(t)$ to be non-singular, a fact that underlies the "direct approach" to solving DAEs of index 1 seen in the popular codes DASSL [2], LSODI [10], and SPRINT [1]. Solving a DAE is more complicated than solving an ODE because a DAE has a solution only if the initial conditions $\boldsymbol{y}_0$ are consistent in the sense that the equation $\boldsymbol{M}(t_0)\boldsymbol{y}_0' = \boldsymbol{f}(t_0, \boldsymbol{y}_0)$ has a solution $\boldsymbol{y}_0'$ for the initial slope. Most codes for DAEs ask the user to supply consistent $\boldsymbol{y}_0$ and $\boldsymbol{y}_0'$. DASSL and SPRINT have options for the automatic computation of consistent initial conditions. However, Petzold [13] observes that "Probably the biggest complaint of DASSL users has been the lack of a robust and general code for finding a consistent set of initial conditions." Our goal was not merely to extend `ode15s` so that it could solve problems with singular $\boldsymbol{M}(t)$, rather to make solving a DAE of index 1 with this code as much like solving an ODE as possible. With the extended `ode15s`, a user need not make *any* distinction between solving a DAE and an ODE. The code recognizes automatically that the problem is a DAE, computes automatically consistent initial conditions close to the given $\boldsymbol{y}_0$, and goes on to solve the problem with the direct approach. All the capabilities of this powerful ODE solver are available when solving a DAE. We describe how this is done in section 2. Although we do exploit the possibilities of MATLAB in accomplishing this, some of the developments are of general interest. In particular, the scheme for computing

* Draft of February 22, 1999.

† Math. Dept., Southern Methodist University, Dallas, TX 75275. (lshampin@mail.smu.edu)

‡ Peanut Press, 11 Coolidge Road, Wayland, MA 01778. (mark@peanutpress.com)

§ The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760. (jkierzenka@mathworks.com) consistent initial conditions described in section 2.1 can be used in general scientific computation.

SIMULINK is a widely used tool for simulating physical models specified in a directional block diagram language. An "algebraic loop" in a block diagram is a set of blocks connected in a loop so that their outputs affect directly their own inputs. In mathematical terms, such loops correspond to algebraic equations, hence give rise to DAEs. We prove here that DAEs formulated in SIMULINK must have the semi-explicit form

$$\begin{aligned} \boldsymbol{u}' &= \boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{v}) \\ \boldsymbol{0} &= \boldsymbol{f}_2(t, \boldsymbol{u}, \boldsymbol{v}) \end{aligned} \tag{1.2}$$

Previous versions of SIMULINK use an "ODE approach" to solve these semi-explicit DAEs of index 1. These versions had a limited capability for solving models with algebraic loops, so users had to resort to *ad hoc* changes to models in order to solve DAEs beyond the capabilities of the language. Our goal was to extend greatly these capabilities. We retained the ODE approach because it would not be possible to implement the direct approach in SIMULINK without significantly changing the package. Besides, in a simulation context this approach is attractive because it is not closely tied to the method for solving ODEs. We describe in section 3 how to solve effectively semi-explicit DAEs of index 1 in SIMULINK using this approach with both the NDFs and an explicit Runge-Kutta method. It is natural to ask if the ODE approach can compete with the direct approach in MATLAB. After a thorough investigation that we report in section 3.2, we concluded that it cannot.

Source code for the solvers and examples of this paper are available at ftp.mathworks.com. The programs require MATLAB version 5.2 or later.

2. Direct Approach.

In this section we study the direct approach to solving DAEs of the form (1.1) with `ode15s` in MATLAB. As mentioned earlier, our goal was to make solving a DAE of index 1 with `ode15s` as much like solving an ODE as possible. To this end, we considered it essential that the program recognize DAEs automatically. When solving an ODE, the user supplies a vector $\boldsymbol{y}_0$ of initial conditions. When the program recognizes a DAE, it must regard this vector as a guess and compute automatically a set of consistent initial conditions $\tilde{\boldsymbol{y}}$ that are "close" to the $\boldsymbol{y}_0$ input. In accordance with our goal, the program cannot ask for a guessed initial slope. We also considered it essential that all the capabilities available when solving ODEs be available when solving DAEs, a decision with serious implications in the cases of event location and sparse mass matrices. The goal was ambitious, but for problems of the form (1.1), it proved possible to achieve by means that we now describe.

Although it might appear that `ode15s` can be used without change to solve a problem with singular $\boldsymbol{M}(t_0)$, it cannot for two reasons. The first is revealed in the computation of an on-scale initial step size, which is where the original code fails when given a DAE. The fundamental issue is the computation of consistent initial conditions, i.e., finding a $\boldsymbol{y}_0$ for which the system $\boldsymbol{M}(t_0)\boldsymbol{y}_0' = \boldsymbol{f}(t_0, \boldsymbol{y}_0)$ has a solution $\boldsymbol{y}_0'$ that serves as initial slope. This issue is the subject of the next section. For now we note that computation of the initial step size in `ode15s` has several phases. The last phase involves the solution of a linear system involving $\boldsymbol{M}(t_0)$, which is singular for a DAE. By simply dropping this phase, a satisfactory scheme is obtained that involves only $\boldsymbol{y}_0'$.

The other reason why `ode15s` gets into trouble appears when solving linear systems of the form $(\boldsymbol{M} - h\gamma \boldsymbol{J})\,\Delta = \boldsymbol{r}$. Here $\boldsymbol{M}$ is the mass matrix evaluated at some

t_m, $\boldsymbol{J}$ is an approximation to the Jacobian of $\boldsymbol{f}$ evaluated at $(t_m, \boldsymbol{y}_m)$, γ is a constant characteristic of the method, and h is the current step size. These matrices are ill-conditioned for stiff ODEs, but in the role they play as iteration matrices when evaluating an implicit formula, only a few accurate digits are required of the Δ. When solving an ODE, reducing h causes the iteration matrix $\boldsymbol{M} - h\gamma\boldsymbol{J}$ to behave more like the non-singular matrix $\boldsymbol{M}$, placing a limit on how ill-conditioned the iteration matrix can be. There is no limit of this kind when the mass matrix is singular. It is shown in [15] that when solving stiff ODEs that degenerate into semi-explicit DAEs as a parameter $\epsilon \to 0$, the condition is $O(h^{-1})$, see also [2], p. 145, case I. Further, it is shown that scaling of the equations corresponding to the algebraic variables by h^{-1} provides a system with a condition that is $O(1)$. The matter is more difficult for mass matrices that are not diagonal because it is easy to construct examples for which row scaling is not helpful. The MATLAB linear equation solvers do not scale, so we supplement them with explicit row scaling. This corrects the scaling of the matrices arising from semi-explicit DAEs and may be helpful in other cases. The linear equation solvers always approximate a condition number and provide a warning message when the matrix is "nearly singular or badly scaled." Such messages serve no useful purpose in the present circumstances, so we take advantage of the possibility in MATLAB 5 of suppressing them.

2.1. Consistent Initial Conditions. DASSL calculates consistent initial conditions by taking a backward Euler, BDF1, step with a "small" step size h and SPRINT takes two such steps. A virtue of the approach is that the computations are much like those of any other step in these BDF codes. However, there are some serious disadvantages. First, the integration starts at $t_0 + h$ instead of t_0. This conflicts with the event location capability of `ode15s` because an event might occur at any time during the integration. Second, something special must be done about assessing the accuracy of these steps. In SPRINT the error is not estimated until the second step. Because the second step is taken from consistent values, the usual estimate is applicable and it is assumed that the error of a first step starting from consistent initial values at t_0 would have been equal to the error estimated at the second step. In DASSL the error test is equivalent to requiring the guess to be no further than the error tolerance from a consistent set of initial conditions. Asking the user to guess consistent initial conditions so accurately is unacceptable in `ode15s`. Indeed, the scheme of DASSL cannot be justified in the mathematical framework of a fixed guess and step size tending to zero because for all sufficiently small h, the guess is not accurate enough for the step to be accepted. We have found a way to compute consistent initial conditions that retains the advantage of the standard approach and avoids its disadvantages. Although the scheme is applicable to more general problems, here we study only index 1 DAEs of the form (1.1). It should be appreciated that these problems are not as general as those accepted by LSODI and SPRINT, and nothing like as general as those accepted by DASSL.

For a guess $\boldsymbol{y}_0$ we compute consistent initial conditions $\tilde{\boldsymbol{y}}$ at the initial point t_0 by solving the system of nonlinear algebraic equations

$$\boldsymbol{M}(t_0)\left(\frac{\tilde{\boldsymbol{y}} - \boldsymbol{y}_0}{h}\right) = \boldsymbol{f}(t_0, \tilde{\boldsymbol{y}}) \tag{2.1}$$

This can be interpreted as a BDF1 step from $t_0 - h$ to t_0. This is possible because BDF1 does not require a function evaluation for t prior to the initial point t_0. As with the standard scheme, the computations are much like those of any other step in `ode15s`. The $\tilde{\boldsymbol{y}}$ computed in this way is a consistent set of initial conditions with $\tilde{\boldsymbol{y}}'$ defined as $(\tilde{\boldsymbol{y}} - \boldsymbol{y}_0)/h$. We prove first that this scheme "works" for linear, constant coefficient problems because the analysis is more transparent and the results are stronger than in the general case.

THEOREM 2.1. *When applied to a linear, constant coefficient DAE of index 1, the initialization scheme is well-defined for any guess $\boldsymbol{y}_0$ for consistent initial conditions and all but a finite number of values of the parameter h.*

Proof. It is shown in [2] that the general linear, constant coefficient DAE

$$\boldsymbol{A}\boldsymbol{y}' + \boldsymbol{B}\boldsymbol{y} = \boldsymbol{f}(t)$$

with real $n \times n$ matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ is solvable if and only if the matrix pencil $\lambda\boldsymbol{A} + \boldsymbol{B}$ is regular. The analysis makes use of a change of variables that for a problem of index 1 results in a semi-explicit system of the form

$$\begin{aligned} \boldsymbol{u}' + \boldsymbol{C}\boldsymbol{u} &= \boldsymbol{q}_1(t) \\ \boldsymbol{v} &= \boldsymbol{q}_2(t) \end{aligned}$$

Here $\boldsymbol{u}$ contains the differential variables and $\boldsymbol{v}$, the algebraic ones. Because our initialization scheme is invariant under the change of variables, we can restrict our attention to such problems without loss of generality. With guess $(\boldsymbol{u}_0^T\ \boldsymbol{v}_0^T)^T$, we solve the system

$$\begin{aligned} \frac{\tilde{\boldsymbol{u}} - \boldsymbol{u}_0}{h} + \boldsymbol{C}\tilde{\boldsymbol{u}} &= \boldsymbol{q}_1(t_0) \\ \tilde{\boldsymbol{v}} &= \boldsymbol{q}_2(t_0) \end{aligned}$$

The algebraic variables $\tilde{\boldsymbol{v}}$ are defined uniquely and the differential variables $\tilde{\boldsymbol{u}}$ are defined uniquely by

$$\tilde{\boldsymbol{u}} = (\boldsymbol{I} + h\boldsymbol{C})^{-1}(\boldsymbol{u}_0 + h\boldsymbol{q}_1(t_0))$$

for all but at most n values of h. $\square$

It is illuminating to note that in the transformed variables, the computed consistent initial conditions converge to $(\boldsymbol{u}_0^T\ \boldsymbol{q}_2(t_0)^T)^T$ as $h \to 0$. These limit values are as close as possible to the guess because any consistent set of algebraic variables must be equal to $\boldsymbol{q}_2(t_0)$ and the differential variables are the same as the guessed values.

In the general case we have to make the usual assumptions about the existence of a solution and the problem being of index 1 for this solution. Correspondingly, the result we prove about the initialization scheme is local in nature. Roughly speaking, the result says that for any guess close to a set of consistent initial conditions, the scheme is well-defined for small h and the computed consistent initial conditions are close to the guess.

THEOREM 2.2. *Suppose that $\boldsymbol{M}(t)\boldsymbol{y}' = \boldsymbol{f}(t, \boldsymbol{y})$ has a solution $\boldsymbol{y}(t)$ and that near this solution, the functions $\boldsymbol{M}$ and $\boldsymbol{f}$ are smooth and the problem is of index 1. For all guesses $\boldsymbol{y}_0$ sufficiently close to $\boldsymbol{y}(t_0)$ and all sufficiently small h, the initialization scheme (2.1) is well-defined. The computed consistent initial conditions $\tilde{\boldsymbol{y}}$ are continuous in both $\boldsymbol{y}_0$ and h. As $h \to 0$, $\tilde{\boldsymbol{y}} \to \boldsymbol{y}(t_0)$.*

Proof. Let $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^T$ be a singular value decomposition of $\boldsymbol{M}(t_0)$. Here $\boldsymbol{U}$ and $\boldsymbol{V}$ are orthogonal matrices, $\boldsymbol{\Sigma} = diag\{\boldsymbol{D}\,\boldsymbol{0}\}$, and $\boldsymbol{D}$ is a diagonal matrix of non-zero singular values of $\boldsymbol{M}(t_0)$. With the time-independent change of variables $\boldsymbol{Y} = \boldsymbol{V}^T\boldsymbol{y}$, (2.1) is equivalent to

$$\boldsymbol{\Sigma}\left(\frac{\tilde{\boldsymbol{Y}} - \boldsymbol{Y}_0}{h}\right) = \boldsymbol{U}^T\boldsymbol{F}\left(t_0, \boldsymbol{V}\tilde{\boldsymbol{Y}}\right) = \boldsymbol{F}_1\left(t_0, \tilde{\boldsymbol{Y}}\right)$$

The structure of $\boldsymbol{\Sigma}$ yields a natural partition of $\boldsymbol{Y}$ into $(\boldsymbol{u}^T \ \boldsymbol{v}^T)^T$ and $\boldsymbol{F}_1$ into $(\widehat{\boldsymbol{f}}_1^T \ \boldsymbol{f}_2^T)^T$. If we let $\boldsymbol{f}_1(t_0, \tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}}) = \boldsymbol{D}^{-1}\widehat{\boldsymbol{f}}_1(t_0, \tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}})$, the equation is equivalent to

$$\begin{aligned} \tilde{\boldsymbol{u}} - \boldsymbol{u}_0 - h\boldsymbol{f}_1(t_0, \tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}}) &= \boldsymbol{0} \\ \boldsymbol{f}_2(t_0, \tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}}) &= \boldsymbol{0} \end{aligned} \tag{2.2}$$

We write this system as $\boldsymbol{G}(\boldsymbol{u}_0, h; \tilde{\boldsymbol{u}}, \tilde{\boldsymbol{v}}) = \boldsymbol{0}$. After transformation, $\boldsymbol{y}(t_0)$ and $\boldsymbol{y}_0$ correspond to $(\boldsymbol{u}(t_0)^T \ \boldsymbol{v}(t_0)^T)^T$ and $(\boldsymbol{u}_0^T \ \boldsymbol{v}_0^T)^T$, respectively. Accordingly, when $\boldsymbol{u}_0 = \boldsymbol{u}(t_0)$ and $h = 0$, (2.2) has the solution $\tilde{\boldsymbol{u}} = \boldsymbol{u}(t_0)$, $\tilde{\boldsymbol{v}} = \boldsymbol{v}(t_0)$. When evaluated at $(\boldsymbol{u}(t_0), 0; \boldsymbol{u}(t_0), \boldsymbol{v}(t_0))$, the Jacobian of (2.2) with respect to the variables $\tilde{\boldsymbol{u}}$ and $\tilde{\boldsymbol{v}}$ has the form

$$\begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \partial\boldsymbol{f}_2/\partial\boldsymbol{u} & \partial\boldsymbol{f}_2/\partial\boldsymbol{v} \end{pmatrix}$$

Because the DAE is of index one, the block $\partial\boldsymbol{f}_2/\partial\boldsymbol{v}$ is non-singular at $(\boldsymbol{u}(t_0)^T \ \boldsymbol{v}(t_0)^T)^T$, hence this Jacobian is non-singular. For smooth $\boldsymbol{f}_1$, $\boldsymbol{f}_2$, both the function $\boldsymbol{G}$ and its Jacobian are continuous in a neighborhood of $(\boldsymbol{u}(t_0), 0; \boldsymbol{u}(t_0), \boldsymbol{v}(t_0))$. The implicit function theorem implies that for all $\boldsymbol{u}_0$ sufficiently close to $\boldsymbol{u}(t_0)$ and all sufficiently small h, (2.2) has a solution $(\tilde{\boldsymbol{u}}^T \ \tilde{\boldsymbol{v}}^T)^T$. This solution is continuous with respect to $\boldsymbol{u}_0$ and h. As $h \to 0$, it converges to $(\boldsymbol{u}(t_0)^T \ \boldsymbol{v}(t_0)^T)^T$. These conclusions are equivalent to those stated in the theorem. $\square$

2.2. Implementation Issues. Theorems 2.1 and 2.2 show the initialization scheme to be quite satisfactory in principle. However, there are some difficulties of a practical nature and there is an important issue not yet raised. The analysis suggests that a "small" h be used. However, as pointed out earlier, this leads to an ill-conditioned iteration matrix $\boldsymbol{M} - h\gamma\boldsymbol{J}$ and row scaling may not be able to correct this, so h should not be "too" small. If no initial step size is specified, we take h to be the smaller of $10^{-4}t_0$ and the maximum step size allowed. In the rather common situation of $t_0 = 0$, the final point of the integration, t_f, is used instead. However, it is also not unusual that $t_0 = 0$ and t_f is "large", leading to a trial h that is much too big. To deal with this, the Jacobian of $\boldsymbol{f}$ at $(t_0, \boldsymbol{y}_0)$ is approximated by $\boldsymbol{J}$ and h is reduced if necessary so that $\|\boldsymbol{M}\| = h\,\|\boldsymbol{J}\|$ in the Frobenius norm, the aim being to balance the contributions of M and J to the iteration matrix.

A simplified Newton (chord) method is used to solve the algebraic equations for $\tilde{\boldsymbol{y}}$. A weak line search with affine invariant test is used to enhance the robustness of the solver with respect to the quality of the guess $\boldsymbol{y}_0$, c.f. [7], p. 225 ff. and [4], p. 99 ff. Nonetheless, it is assumed that the guess is good enough that each iterate will be rather more accurate and if it is not, a new Jacobian is formed. The last Jacobian of the initialization can be used as the first of the integration itself because `ode15s` saves Jacobians and forms a new Jacobian only when it appears advantageous. Two kinds of convergence test are employed, both requiring the algebraic equations to be solved about as well as possible. The residual test is passed when $\left\|\boldsymbol{M}(t_0)\tilde{\boldsymbol{y}}' - \boldsymbol{f}(t_0, \tilde{\boldsymbol{y}})\right\| \leq 1000 * eps * \max\left(\left\|\boldsymbol{M}(t_0)\tilde{\boldsymbol{y}}'\right\|, \|\boldsymbol{f}(t_0, \tilde{\boldsymbol{y}})\|\right)$ where *eps* is the unit roundoff. The other test is passed when the estimated error in $\tilde{\boldsymbol{y}}$ is no greater than $1000 * eps * ||\tilde{\boldsymbol{y}}||$. If convergence cannot be achieved in 15 iterations, h is reduced by an order of magnitude for another try and up to three tries are allowed.

The initialization procedure provides initial conditions $\tilde{\boldsymbol{y}}$ and slope $\tilde{\boldsymbol{y}}' = (\tilde{\boldsymbol{y}} - \boldsymbol{y}_0)/h$ that are consistent, but the slope tends to be large because $\boldsymbol{y}_0$ is inconsistent and as $h \to 0$, $\tilde{\boldsymbol{y}} \to \boldsymbol{y}(t_0) \neq \boldsymbol{y}_0$. We would prefer consistent initial conditions with a smaller slope, so we make a second pass through the initialization procedure with $\tilde{\boldsymbol{y}}$ replacing $\boldsymbol{y}_0$,

$$\boldsymbol{M}(t_0)\left(\frac{\hat{\boldsymbol{y}} - \tilde{\boldsymbol{y}}}{h}\right) = \boldsymbol{f}(t_0, \hat{\boldsymbol{y}}),$$

With a consistent guess, the procedure converges quickly to nearby consistent initial conditions $\hat{\boldsymbol{y}}$ and a corresponding slope $(\hat{\boldsymbol{y}} - \tilde{\boldsymbol{y}})/h$ that is of modest size because it approximates the slope of the solution with the consistent initial values $\tilde{\boldsymbol{y}}$. For two of our example problems the norm of the initial slope is reduced in this second pass by more than two orders of magnitude, quite enough to have important effects in the code.

Now that we understand what `ode15s` does when presented a DAE, let us consider how to recognize a DAE and the consequences of a mistake. The only DAEs that the code attempts to solve are those arising from a singular mass matrix. If a mass matrix is present, the user has the option of answering the question, "Is this a DAE?" Possible answers are `'yes'`, `'no'`, and the default of `'maybe'`. Without a definitive answer the code tests `eps*nnz(Mt)*condest(Mt) > 1`. Here `eps` is the unit roundoff, `Mt` is $\boldsymbol{M}(t_0)$, the `nnz` function counts the number of non-zero entries of a matrix, and the `condest` function is Higham's modification of Hager's method for estimating a condition number of a matrix. The latter function produces a lower bound for the condition, so if this test is passed, the matrix is surely very ill-conditioned for the precision available. We assume that the mass matrix is clearly singular for a DAE and that this test will diagnose the problem as a DAE. It is conceivable that the mass matrix of an ODE is so close to singular that the code diagnoses it as a DAE. A mistake of this kind carries with it little penalty. The automatic selection of the initial step size is a little less efficient. The integration itself differs only in that explicit row scaling is done. The unnecessary computation of consistent initial conditions is not costly. Although it might cause the code to integrate a problem with initial conditions that are not exactly those input, the initial conditions will be quite close and so have little effect on the solution.

We investigated two other schemes for recognizing singularity. A scheme based on the singular value decomposition worked well, but we could not use it because it was not compatible with the sparse matrix capability of `ode15s`. A scheme based on the QR decomposition with column pivoting was compatible and fast, but it was not nearly as reliable as the scheme based on `condest` that we adopted.

2.3. Semi-explicit DAEs. In MATLAB it is easy to recognize DAEs with diagonal mass matrices, i.e., semi-explicit problems. Some of the difficulties of the more general case are not present then, so `ode15s` tests for a semi-explicit problem and uses different algorithms for some of the computations when it recognizes one. It is easy to diagnose a semi-explicit problem as a DAE by testing for zero entries on the diagonal. The zero entries identify the algebraic variables and the others are differential variables. In the computation of consistent initial conditions, only the algebraic variables of the guess are altered. It would be good in any event to preserve the components of the guess that correspond to differential variables, but this has the important practical advantage of reducing the size of the system of nonlinear algebraic equations that is solved. Using `condest` we test the Jacobian of this reduced system for singularity because if it is singular, the problem is of index greater than 1. The components of the initial slope corresponding to algebraic variables can simply be set to zero, making unnecessary the second pass of the general scheme for computing consistent initial conditions. As pointed out earlier, explicit row scaling corrects the scaling difficulties with the iteration matrix when the problem is semi-explicit. Semi-explicit problems are common and these advantages are quite important, so it is well worth the complication of recognizing these problems and treating them in a special way. It may not be difficult to reformulate a more general problem in semi-explicit form and the user of `ode15s` should appreciate the advantages of doing so.

2.4. Examples. The `ode15s` code is accompanied by more than a dozen example programs that solve problems taken from the literature. The program is able to integrate the equations without difficulty, so here we make a few remarks about how the code performs when initializing some illustrative problems. The last Jacobian formed during initialization is used to begin the integration, so we report only the number of Jacobian evaluations required for the initialization itself. We also report the number of function evaluations required, not counting those used in forming the Jacobians.

One example solves three variations on the Robertson problem, a classic test problem for codes that solve stiff ODEs:

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= +0.04y_1 - 10^4 y_2 y_3 - 3\times 10^7 y_2^2 \\ y_3' &= 3\times 10^7 y_2^2 \end{aligned}$$

The problem is to be integrated from initial values $\boldsymbol{y}(0) = (1,0,0)^T$ to steady state. The equations admit a linear conservation law that can be used to replace one of the differential equations by an algebraic equation. This is done as an example in the prolog to LSODI [10]:

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ y_2' &= +0.04y_1 - 10^4 y_2 y_3 - 3\times 10^7 y_2^2 \\ 0 &= y_1 + y_2 + y_3 - 1 \end{aligned}$$

Obviously the algebraic variable y_3 is uniquely determined by the differential variables so this problem is of index 1 and consistent initial values are obvious. This DAE is equivalent to the ODE and is also stiff. Before effective codes for stiff ODEs were widely available, chemists coped with stiffness by resorting to steady state approximations. Edsberg [6] treats the Robertson problem as an example:

$$\begin{aligned} y_1' &= -0.04y_1 + 10^4 y_2 y_3 \\ 0 &= +0.04y_1 - 10^4 y_2 y_3 - 3\times 10^7 y_2^2 \\ y_3' &= 3\times 10^7 {y_2}^2 \end{aligned}$$

Among the disadvantages of the approach listed by Edsberg is the need to find consistent initial conditions. With the initial values $y_1(0) = 1$, $y_3(0) = 0$, it is easily seen that there are two consistent choices for $y_2(0)$, namely $\pm 3.65 \times 10^{-5}$. The physical requirement that $y_2(0) \geq 0$ determines the value that should be used here, but it is to be appreciated that with a suitable guess, `ode15s` will compute the other value and then compute a non-physical solution. For both DAEs the example program `ex4` guesses the algebraic variable to be 10^{-3}. The algebraic equation of the first form is linear and the code forms only one approximate Jacobian during the initialization. This Jacobian is "free" because it is used for the subsequent integration. Two additional function evaluations were required for initialization. The nonlinear algebraic equation of the second form was surprisingly difficult, requiring 6 extra Jacobians and 11 function evaluations. In a separate computation we solved the quadratic with a simplified Newton method and found that with the same guess, 10 iterations were required to obtain equivalent accuracy. Evidently the guess is not close enough to a root for this method to exhibit rapid convergence. It appears, then, that the initialization did perform in an acceptable manner.

Dew and Walsh [5] present a program for solving elliptic-parabolic partial differential equations, PDEs, in one space variable by the method of lines. The semi-discrete equations are ODEs unless there are elliptic PDEs in the system, in which case they are DAEs in semi-explicit form. Interestingly, initial conditions obtained by discretization of the initial condition for the PDEs are generally not consistent for the DAE. Fortunately, as the spatial mesh is refined and the number of equations increases, initial conditions obtained in this way are closer to being consistent. Dew and Walsh initialize in a way much like that used by `ode15s` for semi-explicit problems. Specifically, they use a Newton iteration to solve the algebraic equations for consistent initial values. Also, their BDF code requires the initial slope and like us, they set to zero the components of the slope corresponding to algebraic variables. The example program `ex12` solves Problem 1 of [5]. With the discretization used, the semi-explicit DAE has 19 differential variables and 21 algebraic variables. The PDEs are nonlinear with the consequence that all the algebraic equations are nonlinear. To solve this problem, `ode15s` required no extra Jacobian evaluations and only 3 extra function evaluations for the initialization. In the program of Dew and Walsh, DAEs arise only in the relatively unusual case of both elliptic and parabolic equations being present in the system. We have drafted a program for the solution of PDEs by the method of lines that is based on `ode15s` and the semi-discretization of Skeel and Berzins [17]. In this discretization, Dirichlet boundary conditions give rise to algebraic equations, so when solving systems of parabolic equations with our draft program, DAEs involving one or two algebraic equations are common. Just as with the discretization of Dew and Walsh, elliptic equations give rise to many algebraic equations. Obviously the automatic computation of consistent initial conditions in `ode15s` is of great value in this application of the solver. Although the basis of their discretization is different, Skeel and Berzins use lumping in such a way that they also obtain only semi-explicit DAEs, a very desirable characteristic that we exploit with `ode15s`.

A particularly interesting example of initialization is the DAS 2 problem of Cameron [3] that we solve in the example program `ex10`. In the course of the simulation a number of events occur, such as the opening or closing of a valve, at which times the differential equations may change discontinuously. At such times a new integration must be started with initial value equal to the solution at the end of the previous integration. Because the equations may change, this initial value may not be a consistent initial condition. For the first integration `ode15s` requires one extra Jacobian and 6 extra function evaluations to compute consistent initial conditions. The code is able to recognize that the initial conditions are consistent for three of

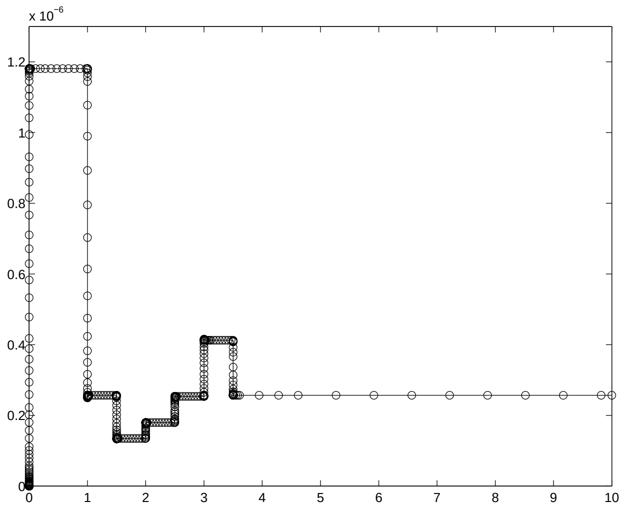

FIG. 2.1. *Height of liquid for DAS 2 problem.*

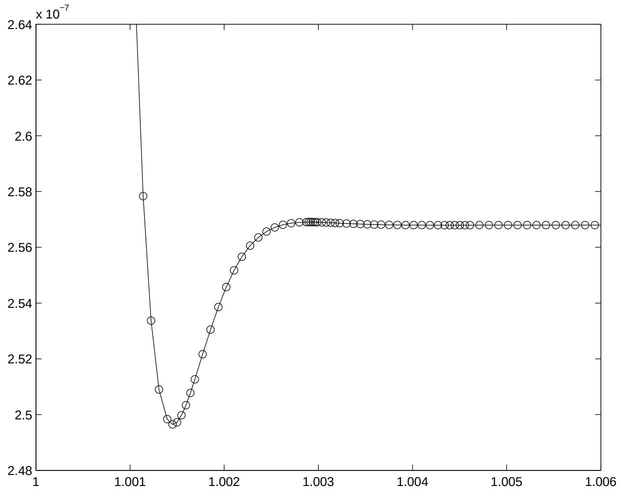

FIG. 2.2. *Height of liquid near first event.*

the remaining six integrations. Each of the other three integrations required only one extra function evaluation to initialize. The plot of one solution component shown in Figure 2.1 shows what appear to be vertical lines and discontinuities at events. As Figure 2.2 shows, this is a consequence of the solution changing on vastly different time scales.

Hairer, Lubich, and Roche [8], p. 106 ff. discuss the solution of a two phase plug flow problem due to Byrne and Hindmarsh. After some preparation they obtain a DAE in semi-explicit form involving one differential equation and two algebraic equations. There are two sets of parameter values that lead to quite different solutions. If the consistent initial conditions of [8] are used as guess, `ode15s` does one simplified Newton iteration at a cost of one extra function evaluation. The example programs `ex5` and `ex6` use half the correct values of the algebraic values in $\boldsymbol{y}_0$. For

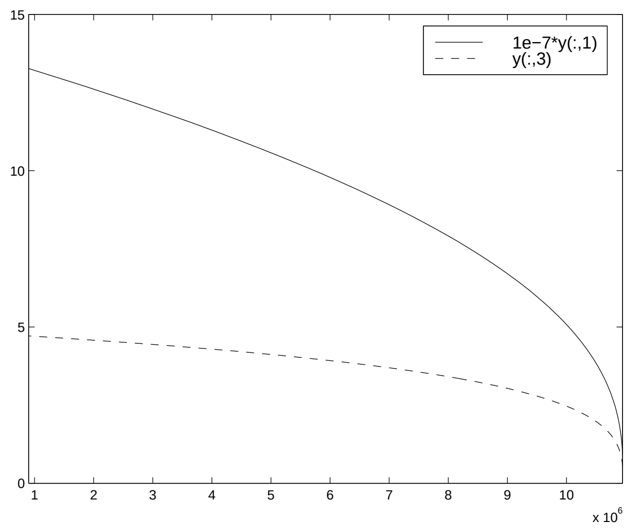

FIG. 2.3. *Choked two phase plug flow. The first component is multiplied by* 10^{-7}.

both problems two additional Jacobians and 9 extra function evaluations are required for initialization from this guess. The example programs reproduce the solutions displayed in [8] as Figures 9.1 and 9.2. The solution to the second problem, shown here in Figure 2.3, is especially interesting because it corresponds to a choked flow, meaning that there is a vertical tangent to the graph of the solution. The `ode15s` code terminates the computation at time 1.095789×10^7 with a message that there appears to be a singularity. The location of the singularity agrees with the value reported by other authors. Stopping short of the singularity by integrating to 1.095×10^7 furnishes a graph that has the same appearance, but runs in about one third the time; evidently the bulk of the cost occurs as the step size goes to zero on approach to the singularity.

Three of the example problems, `ex7`, `ex8`, and `ex13`, are not semi-explicit. The one transistor amplifier model studied in [9], p. 376 ff. arises naturally with a constant mass matrix that is not diagonal. Hairer and Wanner show how to write the problem in semi-explicit form, but we use the original form to test the initialization of problems that are not semi-explicit. The example program `ex7` adds 0.1 to the consistent values 6 and 0 of the two algebraic variables. No extra Jacobians and only 6 extra function evaluations are needed for the initialization. The value of the parameter h selected by the code suffices. The computed $\hat{\boldsymbol{y}}$ differs in a relative sense from the consistent initial conditions of [9] by 2×10^{-4}. Nevertheless, it satisfies the consistency condition to high accuracy because $\left\| \boldsymbol{M}(t_0)\hat{\boldsymbol{y}}' - \boldsymbol{f}(t_0, \hat{\boldsymbol{y}}) \right\| = 5 \times 10^{-17}$ and $\left\| \boldsymbol{M}(t_0)\hat{\boldsymbol{y}}' \right\| = 3 \times 10^{-4}$. A two transistor amplifier model studied in [8] is solved in the example program `ex8`. When the consistent initial conditions provided by Hairer, Lubich, and Roche are used, `ode15s` requires no extra Jacobians and 7 extra function evaluations to initialize. The first value of the parameter h tried by the code works. The computed initial conditions differ from the consistent initial guess in a relative sense by 2×10^{-4}. The computed initial conditions are consistent to high accuracy because $\left\| \boldsymbol{M}(t_0)\hat{\boldsymbol{y}}' - \boldsymbol{f}(t_0, \hat{\boldsymbol{y}}) \right\| = 5 \times 10^{-16}$ and $\left\| \boldsymbol{M}(t_0)\hat{\boldsymbol{y}}' \right\| = 5 \times 10^{-4}$. As posed by Preston, Berzins, Dew, and Scales, the discharge pressure control problem studied in [8] is of index 2. In proving this, Hairer, Lubich, and Roche introduce for analytical purposes a variable $s = c + p/15$

and reduce the problem to one of index 1. The example program `ex13` solves this index 1 problem. However, it does not introduce the variable s in order to have a problem with a mass matrix that is not diagonal. When the first few digits of the consistent initial conditions of [8] are used as guess, `ode15s` requires no extra Jacobians and 3 extra function evaluations for the initialization. The first value of h tried by the code works. The computed initial conditions differ from those of [8], but they are consistent to high accuracy.

2.5. Trapezoidal Rule. Although the initialization procedure has been described in terms of steps with BDF1 and exemplified only with the `ode15s` code, it is not restricted to BDF codes. The trapezoidal rule code `ode23t` added to MATLAB at version 5.2 makes the point. Because this implicit Runge-Kutta formula is not damped at infinity, it has not received much attention as a method for the solution of DAEs of index 1. However, for the simulation of important kinds of electrical circuits, this stability behavior is precisely what is desired. It is shown in [8] and [2] that the method converges for DAEs of index 1 and has the same rate of convergence for both differential and algebraic variables. Extending `ode23t` to solve DAEs of index 1 involved nothing more than copying the algorithms of `ode15s` for recognizing DAEs and computing consistent initial conditions. The extended code solves efficiently all the examples supplied with `ode15s` for which the stability properties of the trapezoidal rule are appropriate.

3. ODE Approach. In this section we study the ODE approach to solving semi-explicit DAEs of index 1 in the form (1.2). The approach is based on the usual treatment of existence and uniqueness for such problems, see e.g. [2, 9]. The equations are to be solved on an interval $[t_0, t_f]$ with initial condition $\boldsymbol{u}(t_0) = \boldsymbol{u}_0$ and a guess $\boldsymbol{v}_0$ for $\boldsymbol{v}(t_0)$. Assuming that the algebraic equations $\boldsymbol{0} = \boldsymbol{f}_2(t_0, \boldsymbol{u}_0, \boldsymbol{v})$ have a solution $\boldsymbol{V}$ near $\boldsymbol{v}_0$, the key requirement is that the Jacobian $\partial \boldsymbol{f}_2 / \partial \boldsymbol{v}$ is non-singular in a region containing $(t_0, \boldsymbol{u}_0, \boldsymbol{V})$. This assumption on $\partial \boldsymbol{f}_2 / \partial \boldsymbol{v}$ makes the problem of index 1. The implicit function theorem implies the existence of a function $\boldsymbol{R}(t, \boldsymbol{u})$ such that $\boldsymbol{R}(t_0, \boldsymbol{u}_0) = \boldsymbol{V}$ and $\boldsymbol{0} = \boldsymbol{f}_2(t, \boldsymbol{u}, \boldsymbol{R}(t, \boldsymbol{u}))$. The differential equations are then $\boldsymbol{u}' = \boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{R}(t, \boldsymbol{u})) = \boldsymbol{f}(t, \boldsymbol{u})$ and the task reduces to solving an initial value problem for an ODE. In this approach an ODE is solved with a function $\boldsymbol{f}(t, \boldsymbol{u})$ that happens to be complicated to evaluate because it involves the solution of a system of nonlinear algebraic equations. In the literature this approach is often called the "indirect approach", but we think "ODE approach" is apt. A powerful attraction of the ODE approach is that it can be used with any of the methods implemented in the codes of the ODE Suite and SIMULINK. These include two explicit Runge-Kutta codes and an Adams-Bashforth-Moulton PECE code for non-stiff problems and an NDF code, a modified Rosenbrock code, and a trapezoidal rule code for stiff problems.

3.1. SIMULINK. The ODE approach assumes that the DAE is semi-explicit and of index 1. We prove that the DAEs in SIMULINK are semi-explicit, but it is not hard to construct examples that have index greater than one. We have made no provision for solving such DAEs in SIMULINK.

THEOREM 3.1. *A system of equations can be represented in a SIMULINK block diagram if and only if the system can be written as a DAE in semi-explicit form (1.2).*

Proof. First, if a DAE system has the form (1.2), then the block diagram shown in Figure 3.1 represents it in SIMULINK. In this figure the first two blocks implement $\boldsymbol{u}' = \boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{v})$ and the last two blocks implement $\boldsymbol{v} = \boldsymbol{f}_2(t, \boldsymbol{u}, \boldsymbol{v}) + \boldsymbol{v}$, i.e., $\boldsymbol{0} = \boldsymbol{f}_2(t, \boldsymbol{u}, \boldsymbol{v})$.

Now suppose that we are given a block diagram. Label the output of each integrator block with a unique u_i, and label the output of each non-integrator block with a unique v_i. This labels all lines in the model, except for those that depend solely on t. For each integrator block, write down the equation $u_i' = u_j$ or $u_i' = v_j$, depending on the label of the input to the block. These differential equations are in the form $\boldsymbol{u}' = \boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{v})$. The output v_i of each non-integrator block is a function of its inputs, namely $v_i = g_i(t, \boldsymbol{u}, \boldsymbol{v})$. When this is rewritten as $0 = v_i - g_i(t, \boldsymbol{u}, \boldsymbol{v})$, the algebraic equations are in the form $0 = \boldsymbol{f}_2(t, \boldsymbol{u}, \boldsymbol{v})$. This accounts for all the equations of the system, so we see that it has the semi-explicit form (1.2). $\square$

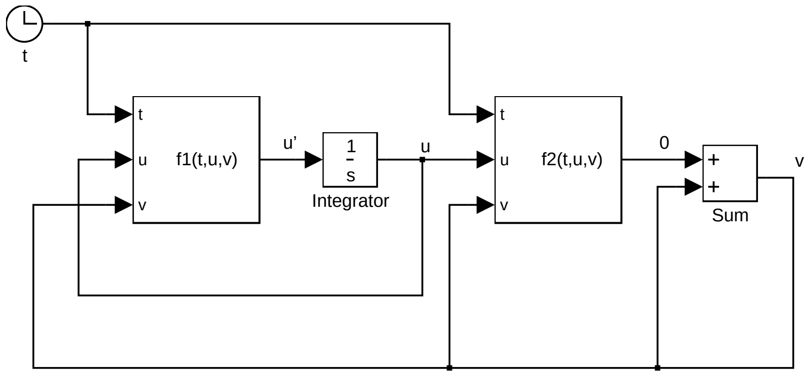

FIG. 3.1. *A block diagram representing the semi-explicit DAE system (1.2).*

In the ODE approach, we rely upon standard ODE solvers. The thing that is different is that each time the integrator needs a value of $\boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{v})$ for given $(t, \boldsymbol{u})$, the algebraic equations $\boldsymbol{0} = \boldsymbol{f}_2(t, \boldsymbol{u}, \boldsymbol{v})$ must first be solved for $\boldsymbol{v}$. The issue is how to solve these equations efficiently, and more important, reliably. The first question is how accurately to solve the algebraic equations. The simplest answer is to solve them about as accurately as possible in the precision available, which proves to be both convenient and practical in the SIMULINK environment. For one thing, there is no provision in this package for specifying tolerances on algebraic variables, though it would be possible to change this.

Because signal flow in a SIMULINK block diagram is directional, the blocks can be topologically ordered. Strongly connected components of the directed graph represent the algebraic constraints. Accordingly, the first step of solving $\boldsymbol{0} = \boldsymbol{f}_2(t, \boldsymbol{u}, \boldsymbol{v})$ for $\boldsymbol{v}$ is to partition the equations into subsets that can be solved independently in topological order, node tearing. Solving the subsets successively is an important advantage of the ODE approach because it is generally faster and more robust. By solving each of the smaller systems about as accurately as possible, we avoid the difficult question of how errors compound in successive solution.

We solve each subsystem of nonlinear algebraic equations with the algorithm of the MINPACK [12] code HYBRD1, customized for the context. This provides a strong foundation for the reliable and efficient solution of algebraic equations. The algorithm makes use of a Jacobian that in the circumstances must be approximated numerically. We found it necessary to strengthen greatly the scheme for doing this. We began by replacing it with the scheme of D.E. Salane [14] used by `ode15s`. However, we found that scaling difficulties sometimes led to columns in the numerical Jacobian that were zero. For DAEs of index 1 the Jacobian cannot have zero columns, so we must recognize such columns and compute them more accurately. We also found it necessary to use central rather than one-sided differences to deal with signals that have discontinuous first derivatives.

A good starting guess is of obvious importance. The popular codes using the direct approach are based on methods that approximate the algebraic variables by polynomial interpolants, just like the differential variables. Although it would require more interaction with the integrator than is convenient, it would be possible to do this in SIMULINK when using `ode15s`. However, prediction of the algebraic variables is fundamentally different for the Runge-Kutta and Rosenbrock methods of the package. Where the true solution $(t, \boldsymbol{u}(t), \boldsymbol{v}(t))$ is smooth, an interpolant could be used to predict the solution to the same order of accuracy as the method for taking a step. There are well-known difficulties with this, but the fundamental difficulty in the present context is that the one-step methods evaluate $\boldsymbol{f}_2(t, \boldsymbol{\theta}, \boldsymbol{v})$ at judiciously chosen arguments $\boldsymbol{\theta}$ and points t in the span of a step of size h. These intermediate arguments are close to $\boldsymbol{u}(t)$, but differ typically by $O(h^2)$. Predicting $\boldsymbol{v}(t)$ to a high order of accuracy for these t is pointless because it corresponds to the argument $(t, \boldsymbol{u}(t))$ and so is not especially close to the $\boldsymbol{v}$ corresponding to the actual argument $(t, \boldsymbol{\theta})$. We have tried a number of schemes for the initial guess and the simplest has worked best. Suppose that we have reached $(t_n, \boldsymbol{u}_n, \boldsymbol{v}_n)$. In the computation of the stages for the step from t_n, we start with the initial guess $\boldsymbol{v}_n$ and thereafter use the result from one computation of $\boldsymbol{v}$ as the initial guess for the next stage. This scheme can be used with any method for ODEs, which is important in the application.

3.2. MATLAB. The ODE approach is the natural one in SIMULINK, but can it compete with the direct approach in the rather different environment of MATLAB? To investigate this we modified the Runge-Kutta code `ode45` and the NDF code `ode15s` to use the approach for semi-explicit problems.

There are a number of questions that arise in connection with the software interface. The structure of SIMULINK makes it natural to access the algebraic equations independently of the differential equations. This would be advantageous in MATLAB, too, but it would complicate the user interface and require the user to code the equations for independent evaluation, so we did not treat these equations differently in the modified codes.

In their discussion of the ODE approach, Thompson and Tuttle [18] say "This frequently proves impractical for several reasons. The first is caused by the interaction of the convergence error in the iteration with error control in the integrator. If the convergence tolerance is too small, the program's execution time is high (due to the iteration); whereas if the tolerance is too large, the integrator error test fails repeatedly, leading to prohibitively small step-sizes and resulting in high execution times as well as possibly erroneous results." The key point is that to compute the differential variables to a specified accuracy, it may be necessary to compute the algebraic variables more accurately because the value of the function $\boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{v})$ is sensitive to $\boldsymbol{v}$. We avoided this in SIMULINK by computing the algebraic variables about as accurately as possible in the precision available, but that is not practical in MATLAB because function evaluation is slower. The matter does not arise in the direct approach because all the variables are computed simultaneously, hence the algebraic variables are automatically computed as accurately as necessary. If the iteration for the algebraic variables is converging quickly to $\boldsymbol{v}^*$, the error of an iterate $\boldsymbol{v}^k$ can be approximated by $\boldsymbol{v}^* - \boldsymbol{v}^k \approx \boldsymbol{v}^{k+1} - \boldsymbol{v}^k$ and the error of the scaled function value $h\boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{v}^k)$ by $h\boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{v}^{k+1}) - h\boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{v}^k)$. Because estimating the error in the scaled function value costs a function evaluation, we first iterate until the algebraic variables are estimated to be as accurate as the tolerance on the differential variables. We then iterate until we estimate the scaled function value to be sufficiently accurate. Two function evaluations are needed to recognize convergence, but because we insist on rapid convergence of the iterations, usually only two were made in our experiments.

In section 3.1 we explained why it is harder to predict the algebraic variables in the ODE approach when using a Runge-Kutta method. There is a related difficulty in both `ode45` and `ode15s` with intermediate output. All the codes of the ODE Suite have continuous extensions of the formulas that are used to obtain approximate solutions between steps for "free". This is basic to the event location capability available in all the codes. This stays the same in the direct approach because there are interpolants for both algebraic and differential variables, but in the ODE approach there is the additional cost of computing algebraic variables that correspond to the interpolated differential variables. Unfortunately, this computation is less effective than when taking a step because the algebraic variables computed at the end of the step and an approximation to $\partial \boldsymbol{f}_1/\partial \boldsymbol{v}$ there are used to predict and compute the variables at intermediate points, and they may not be good approximations near the beginning of the step. It might be thought that this difficulty would not be important except when locating events, but because `ode45` takes relatively large steps, by default it approximates the solution at four points in the course of every step in order to produce a smooth graph. Accordingly, there is a considerable additional cost due to computing algebraic variables at intermediate points for a typical integration with this code.

`ode15s` is based on an implicit formula and a step ends with a final correction of the differential variables. The algebraic variables must also be corrected if they are to correspond to the accepted differential variables. A related issue is the formation of approximate Jacobians by finite differences, the default in `ode15s`. In this it is necessary to evaluate $\boldsymbol{f}_1(t, \boldsymbol{u}, \boldsymbol{v})$ a good many times at perturbed values of the differential variables and the algebraic variables corresponding to $(t, \boldsymbol{u})$ are not exactly the same as those corresponding to the perturbed differential variables. Rather than go to the expense of computing accurately the algebraic variables in this situation, we obtain them by means of a single simplified Newton iteration using the current approximation to $\partial \boldsymbol{f}_2/\partial \boldsymbol{v}$ at $(t, \boldsymbol{u})$. Because only a small perturbation is made to the differential variables, a single iteration should suffice, especially since only an approximate Jacobian is required. The same is done with the final correction to the differential variables. There is an analogy for `ode45` because the formula is first-same-as-last, FSAL. The final evaluation is the first stage of the next step and it seemed better in the situation to compute the algebraic variables in usual fashion so as to verify their accuracy.

Despite our best efforts to deal with the issues raised, and others, the version of `ode15s` based on the ODE approach is not competitive with the one based on the direct approach, quite aside from the fact that it deals with a smaller class of problems. To be fair, the code is an acceptable way to solve semi-explicit DAEs, it is just not as efficient, and this is true even without special provision for semi-explicit problems in the direct approach. The situation is less clear in the case of `ode45`. It might be thought that the implicit method of `ode15s` would place it at a great disadvantage when solving non-stiff problems. However, this code is surprisingly effective for non-stiff problems because it forms few Jacobians then and linear algebra is relatively fast in MATLAB. Also, we have pointed out a number of ways in which the

ODE approach is relatively expensive in an explicit Runge-Kutta code as compared to the direct approach in the extended `ode15s`. The conclusions might be different in another computing environment, but in MATLAB we did not find the ODE approach to be competitive with the direct approach.

REFERENCES

[1] M. Berzins, P. Dew, and R. Furzeland, *Developing software for time-dependent problems using the method of lines and differential-algebraic integrators*, Appl. Numer. Math., 5 (1989), pp. 375–397.

[2] K. Brenan, S. Campbell, and L. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Society for Industrial and Applied Mathematics, Philadelphia, 1996.

[3] I. Cameron, *Solution of differential-algebraic systems using diagonally implicit Runge-Kutta methods*, IMA J. Numer. Anal., 3 (1983), pp. 273–289.

[4] P. Deuflhard and A. Hohmann, *Numerical Analysis*, de Gruyter, Berlin, 1995.

[5] P. Dew and J. Walsh, *A set of library routines for solving parabolic equations in one space variable*, ACM Trans. Math. Software, 7 (1981), pp. 295–314.

[6] L. Edsberg, *Numerical methods for mass action kinetics*, in Numerical Methods for Differential Systems, L. Lapidus and W. Schiesser, eds., Academic, New York, 1976, pp. 181–195.

[7] I. Gladwell, *Shooting methods for boundary value problems*, in Modern Numerical Methods for Ordinary Differential Equations, G. Hall and J. Watt, eds., Clarendon Press, Oxford, 1976, ch. 16.

[8] E. Hairer, C. Lubich, and M. Roche, *The Numerical Solution of Differential-Algebraic Systems by Runge-Kutta Methods*, vol. 1409 of Lecture Notes in Mathematics, Springer, Berlin, 1989.

[9] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations II*, Springer, New York, 1991.

[10] A. Hindmarsh, *LSODE and LSODI, two new initial value ordinary differential equation solvers*, ACM SIGNUM Newsletter, 15 (1980), pp. 10–11.

[11] The MathWorks, Inc., *MATLAB 5.3 and Simulink 2.3*, 24 Prime Park Way, Natick MA.

[12] J. Moré, D. Sorensen, K. Hillstrom, and B. Garbow, *The MINPACK project*, in Sources and Development of Mathematical Software, W. Cowell, ed., Prentice-Hall, Englewood Cliffs, NJ, 1984, ch. 5.

[13] L. Petzold, *Numerical methods for differential-algebraic equations—current status and future directions*, in Computational Ordinary Differential Equations, J. Cash and I. Gladwell, eds., Clarendon Press, Oxford, 1992, pp. 259–273.

[14] D. Salane, *Adaptive routines for forming Jacobians numerically*, Tech. Report SAND86–1319, Sandia National Laboratories, Albuquerque, NM, 1986.

[15] L. Shampine, *Conditioning of matrices arising in the solution of stiff ODEs*, Tech. Report SAND82-0906, Sandia National Laboratories, Albuquerque, NM, 1982.

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

[17] R. Skeel and M. Berzins, *A method for the spatial discretization of parabolic equations in one space variable*, SIAM J. Sci. Stat. Comput., 11 (1990), pp. 1–32.

[18] S. Thompson and P. Tuttle, *The evolution of an ODE solver in an industrial environment*, in Stiff Computation, R. Aiken, ed., Oxford Univ. Press, Oxford, 1985, pp. 180–202.