Design of New DASPK for Sensitivity Analysis *

Shengtai Li and Linda Petzold

Abstract

A new version of DASPK, DASPK3.0, with capability for sensitivity analysis is presented in this report. DASPK3.0 differs from the sensitivity code DASPKSO, described in [12], in several ways. DASPK3.0 has all the features, which were not available in DASPKSO, of the previous version DASPK2.0. One of these features is an improved algorithm for calculation of consistent initial conditions for index-zero or index-one systems. DASPK3.0 also incorporates a mechanism for initialization and solution of index-2 systems. Other improvements in DASPK3.0 include a more accurate error and convergence test, particularly for the sensitivity analysis. We implemented the Krylov method for sensitivity computation with a different strategy from DASPKSO, and made it more efficient and easier for parallel computing. We also added the staggered corrector method [7] for both the direct and Krylov method. We implemented the sensitivity analysis with an internal parallel mode, which is easy to use for both serial and parallel computation with message passing interface (MPI). We also incorporated automatic differentiation into DASPK3.0 to evaluate the Jacobian matrix and sensitivity equations. The goal of our design has been to be compatible as much as possible with DASPK2.0, to minimize memory and storage requirements for sensitivity analysis, and to speed up the computation for a large number of sensitivity parameters.

*This work was partially supported by DOE contract number DE-FG03-98ER25354, NSF grant CCR-9896198, NSF/DARPA grant DMS-9615858, and LLNL ISCR 99-006.

1 Introduction

This paper is concerned with the solution and sensitivity analysis of initial value problems for differential-algebraic equation systems (DAEs) in the general form

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

where F, y, and y' are N-dimensional vectors. Two software packages have been written for solving initial value problems for the DAE system (1) —DASSL [13], and an extension of it called DASPK [3]. Both use variable-order variable-stepsize backward differentiation formulas. DASSL solves the linear systems that arise at each time step by dense or banded direct linear system methods. In DASPK, the linear systems that arise at each time step are solved with either direct linear system methods, or with a preconditioned Krylov iterative method, namely GMRES [16]. For large-scale systems, the iterative method combined with a suitable preconditioner can be quite effective.

Sensitivity analysis of DAEs is important in many engineering and scientific applications. The information contained in the sensitivity trajectories is useful for parameter estimation, optimization, model reduction and experimental design. The sensitivity equations for the DAEs have many good properties which can be taken advantage of. First they are linear with respect to the sensitivity variables. Second, the Jacobian matrix for the sensitivities is the same as for the original DAEs. Several methods and codes have been designed in the last decade to compute sensitivities for DAEs [12, 5]. DASPKSO [12] is one of them. It solves for the state variables and the sensitivity variables simultaneously in the nonlinear corrector step. This implementation is called the *simultaneous corrector* method. Since the DASPKSO code was designed based on the first version, DASPK1.0, of DASPK, it did not include the new mechanism [4] to calculate consistent initial conditions for index-0 or 1 DAEs. We rewrote it and incorporated the features of the newer version, DASPK2.0, of DASPK.

The Krylov method in DASPKSO solves a large linear system, which consists of both the state variables and sensitivity variables, at each Newton iteration. Sensitivity equations are evaluated at each linear iteration in the Krylov method. This implementation was difficult to parallelize and to incorporate the *staggered corrector method*, which we will talk about later. In DASPK3.0, we implement the Krylov method in a different way: the large linear system is split into several smaller linear systems with the same Jacobian matrix derived from the state variables. This method not only reduces the storage requirement for the Krylov iteration but also improves the efficiency.

When we solve for the sensitivity and state variables together, it is easy to take them as one whole vector in the error and convergence control. However, because DASPK2.0 uses the root-mean-square (RMS) norm, the accuracy of the state variables can be affected by errors in the sensitivity variables. In our new implementation, we evaluate the RMS norm separately for the state variables and for the sensitivities with respect to each parameter. We use the maximum of these norms in making decisions in the code.

More recently, the *staggered corrector* method [7] has been proposed. This method does not solve for the state variables and sensitivities simultaneously in the corrector step. Instead, on each time step, it solves for the state variables first and then solves for the sensitivity variables. This method has proven to be more efficient than the simultaneous corrector method for most problems. We implemented this method in DASPK3.0 to compute the sensitivities for both the direct method and the Krylov method.

In either of the solvers DASSL, DASPK or their sensitivity analysis solvers DASSLSO and DASPKSO, finite differencing (forward or central) is chosen as the default method to evaluate the Jacobian matrix for the direct method, the matrix-vector product in the Krylov iteration, and the residuals for the sensitivity equations in case exact input is not available. For most well-scaled and smooth problems, the results using the finite differencing are nearly as good as for exact input. However, for some badly-scaled problems, finite-differencing cannot get accurate results for the sensitivities. For some strongly nonlinear problems, exact input of the Jacobian in the direct method can greatly improve the accuracy and efficiency. The automatic differentiation tool ADIFOR [1] can generate a subroutine to compute the Jacobian matrix with accuracy up to round-off error. In our experience, ADIFOR-generated derivative code usually outperforms divided-difference approximations. In DASPK3.0, we provide an option to use ADIFOR to compute the derivatives. We embed the ADIFOR-generated routine in such a way that the previous user interface is not altered very much.

Each sensitivity is independent of the others in sensitivity analysis, which is ideal for parallel computation. Several parallel implementations for the sensitivity analysis of DAEs were compared in [15]. The distributed parameter only (DPO) method proved to be the most efficient one in [15]. However, the DPO method as implemented in [15] requires the user to distribute the parameters to each processor and to define a different problem for sensitivity analysis in each processor before calling the DASPK program, which is error-prone and difficult for an inexperienced user. In DASPK3.0, the sensitivities and parameters are distributed automatically to each processor. The problem (residual routine in DASPK) is defined only once and is the same for all the processors. Synchronization at the output time or on the same stepsize can be achieved easily, with minimal communication overhead. We use the portable message passing interface (MPI) in our parallel implementation.

Most of the functionality we added in DASPK3.0 is only for the sensitivity analysis. However, our design is compatible with the previous version of DASPK. We appended two arguments in the argument list of DASPK2.0: one for the sensitivity parameters and one for the ADIFOR-generated routine for the sensitivity equations. They can be removed or treated as dummies if not used. Since variable argument lists are valid on most platforms, codes that use the previous version DASPK2.0 should work in our new version as long as the size of INFO(*) is increased to at least 27.

2 Background

2.1 DASSL and DASPK1.0

DASSL was developed by Petzold [13] and has become one of the most widely used production codes for DAEs at this time. DASSL uses backward differentiation formula (BDF) methods [2] to solve a system of DAEs or ODEs. The methods are variable step-size, variable order. The system of equations in DASSL is written in implicit ODE or DAE form as in (1). Following discretization by the BDF methods, a nonlinear equation

$$F(t, y, \alpha y + \beta) = 0 \tag{2}$$

must be solved at each time step, where $\alpha = \alpha_0/h_n$ is a constant which changes whenever the stepsize or order changes, β is a vector which depends on the solution at past times, and t, y, α, β are evaluated at t_n. To simplify the discussion, we will sometimes refer to the above function as $F(y)$. DASSL solves this equation by a modified version of Newton's method,

$$y^{(m+1)} = y^{(m)} - c\left(\alpha\frac{\partial F}{\partial y'} + \frac{\partial F}{\partial y}\right)^{-1} G(t, y^{(m)}, \alpha y^{(m)} + \beta). \tag{3}$$

The linear system is solved via a dense or banded direct solver in DASSL. The iteration matrix

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

is computed and factored and is then used for as many time steps as possible. The reader can refer to [13] for more implementation details.

DASPK1.0 was developed by Brown, Hindmarsh and Petzold [3] for the solution of large-scale systems of DAEs. It is particularly effective in the method-of-lines solution of time-dependent PDE systems in two and three dimensions. In contrast to DASSL, which is limited in its linear algebra to dense and banded systems, DASPK1.0 has an option to make use of the preconditioned GMRES iterative method for solving the linear system at each Newton iteration. Here we describe some of the basic features of the DASPK algorithm. Further details on the structure and use of DASPK1.0 and on a class of preconditioners for DAE systems for reaction-diffusion type PDE problems can be found in [3].

DASPK1.0 also has an option to use direct methods, which is virtually identical to using DASSL. In the case of DASPK with iterative methods, a preconditioner matrix P, which is an approximation to A that leads to a cheap linear system solution, is computed and factored and used for as many time steps as possible. It is often possible to use a preconditioner over more steps than it would be possible to keep an iteration matrix in the direct methods, because the iterative methods do the rest of the work in solving the system. One of the powerful features of the iterative approach is that it does not need to compute and store the iteration matrix A explicitly in DASPK. This is because the GMRES method never actually needs the matrix explicitly[1]. Instead, it requires only the action of A times a vector v. In DASPK, this matrix-vector product is approximated via a difference of the function F in (2)

$$Av = F'(y)v \simeq \frac{F(t, y + \sigma v, \alpha(y + \sigma v) + \beta) - F(t, y, \alpha y + \beta)}{\sigma}. \tag{4}$$

The GMRES algorithm requires the product Av in which v is a vector of unit length (the norm is a WRMS norm) and y is the current iterate. We note that because y is current in (4), this amounts to taking a full Newton iteration in the iterative option of DASPK (rather than modified Newton, as in DASSL and the direct option of DASPK). In fact, for some highly nonlinear problems we have seen the iterative option in DASPK outperform the direct option in terms of time steps, corrector failures, etc., apparently for this reason.

In DASPK, the iterative option requires the user to provide a preconditioner P. This is in part because the Newton iteration test, and hence ultimately the code reliability, is not justified without a reasonably accurate preconditioner. It is also because any nontrivial DAE needs a preconditioner. Several preconditioners are provided in [4] for method-of-lines solution of PDEs. One of them is the incomplete LU factorization (ILU) preconditioner, which can be used for any sparse linear system.

[1] Depending on the preconditioner, it may need to compute and store a preconditioner matrix explicitly. We hope that this matrix is much cheaper to generate and to store than the actual iteration matrix.

2.2 Consistent initial condition calculation and DASPK2.0

When using either of the solvers DASSL or DASPK, the integration must be started with a consistent set of initial conditions y_0 and y'_0. The present DASSL, DASPKSO, and old version DASPK (DASPK1.0, released before 1996) solvers offer an option for finding consistent y'_0 from a given initial y_0, by taking a small artificial step with the backward Euler method. However, initialization problems do not always arise in this form, and even for the intended problem type, that technique is not always successful. In any case it is unsatisfactory in that it produces values at $t = t_0 + h$ (h=stepsize) rather than at $t = t_0$. In [4], a new algorithm was proposed to calculate the consistent initial conditions for index-one DAEs. Two types of initialization problem have been solved and the algorithm converges nearly as rapidly as the underlying Newton or Newton-Krylov method. The new method is very convenient for the user, because it makes use of the Jacobian or preconditioner matrices which are required in DASSL or DASPK. We call it DASPK2.0 because of its difference from DASPK1.0 in consistent initial condition calculation.

The new initialization technique is applicable to two classes of index-1 initialization problems. Initialization problem I is posed for systems of the form

$$\begin{aligned} f(t,u,v,u') &= 0, \\ g(t,u,v) &= 0, \end{aligned} \tag{5}$$

where $u, f \in R^{N_d}$ and $v, g \in R^{N_a}$, with the matrices $f_{u'} = \partial f/\partial u'$, $g_v = \partial g/\partial v$ square and nonsingular. The problem is to find the initial value v_0 of v when the initial value u_0 for u is specified. Hence it is required for the user to specify which variables are algebraic and which are differential.

In initialization problem II, which is applicable to the general index-1 system (1), the initial derivatives are specified but all of the dependent variables are unknown. That is, we must solve for y_0 given y'_0. For example, beginning the DAE solution at a steady state corresponds to specifying $y'_0 = 0$.

The idea motivating DASPK2.0 is to solve both of these initial condition problems with the help of mechanisms already in place for the solution of the DAE system itself, rather than requiring the user to perform a special computation for it. This idea is also used to develop the initialization algorithm for index-2 systems described in Section 4. A detailed description of the algorithm and convergence theory has been provided in [4].

3 Sensitivity analysis of DAEs

Sensitivity analysis of a DAE model may yield information useful for parameter estimation, optimization, process sensitivity, model reduction, and experimental design. Several approaches have been developed to calculate sensitivity coefficients. A sensitivity analysis capability has been developed by Maly and Petzold [12], and implemented into DASSL and DASPK1.0, yielding two new codes DASSLSO and DASPKSO. Here we summarize the algorithms used by the sensitivity option in DASSLSO and DASPKSO, and other proposed sensitivity analysis methods. Further detail about the codes and some numerical results can be found in [12].

To illustrate the basic approach for sensitivity analysis, consider the general DAE system with parameters,

$$F(t, y, y', p) = 0, \quad y(0) = y_0 \tag{6}$$

where $y \in R^{n_y}, p \in R^{n_p}$. Here n_y is the number of time-dependent variables y as well as the dimension of the DAE system, and n_p is the number of parameters in the original DAE system. Sensitivity analysis entails finding the derivative of the solution y with respect to each parameter. This produces an additional $n_s = n_p \cdot n_y$ sensitivity equations which, together with the original system, yield

$$\begin{aligned} F(t, y, y', p) &= 0, \\ \frac{\partial F}{\partial y} s_i + \frac{\partial F}{\partial y'} s_i' + \frac{\partial F}{\partial p} &= 0, \quad i = 1, ..., n_p, \end{aligned} \tag{7}$$

where $s_i = dy/dp_i$. Defining

$$Y = \begin{bmatrix} y \\ s_1 \\ \vdots \\ s_{n_p} \end{bmatrix}, \quad \mathrm{F} = \begin{bmatrix} F(t, y, p) \\ \frac{\partial F}{\partial y} s_1 + \frac{\partial F}{\partial y'} s_1' + \frac{\partial F}{\partial p_1} \\ \vdots \\ \frac{\partial F}{\partial y} s_{n_p} + \frac{\partial F}{\partial y'} s_{n_p}' + \frac{\partial F}{\partial p_{n_p}} \end{bmatrix}$$

the combined system can be rewritten as

$$\mathrm{F}(t, Y, Y', p) = 0, \quad Y(0) = \begin{bmatrix} y_0 \\ \frac{dy_0}{dp_1} \\ \vdots \\ \frac{dy_0}{dp_{n_p}} \end{bmatrix}.$$

This system can be solved by the k-th order BDF formula with step size h_{n+1} to yield a nonlinear system

$$G(Y_{n+1}) = \mathrm{F}\left(t_{n+1}, Y_{n+1}, {Y'}_{n+1}^{(0)} - \frac{\alpha_s}{h_{n+1}}(Y_{n+1} - Y_{n+1}^{(0)}), p\right) = 0, \tag{8}$$

where $Y_{n+1}^{(0)}$ and $Y'^{(0)}_{n+1}$ are predicted values for Y_{n+1} and Y'_{n+1}, which are obtained via polynomial extrapolation of past values [2]. Also, α_s is the fixed leading coefficient which is defined in [2]. Newton's method for the nonlinear system produces the iteration

$$Y_{n+1}^{(k+1)} = Y_{n+1}^{(k)} - \mathbf{J}^{-1} G(Y_{n+1}^{(k)}),$$

where

$$\mathbf{J} = \begin{bmatrix} J & & & & \\ J_1 & J & & & \\ J_2 & 0 & J & & \\ \vdots & \vdots & \vdots & \ddots & \\ J_{n_p} & 0 & \cdots & 0 & J \end{bmatrix} \tag{9}$$

and

$$J = \alpha \frac{\partial F}{\partial y'} + \frac{\partial F}{\partial y}, \quad J_i = \frac{\partial J}{\partial y} s_i + \frac{\partial J}{\partial p_i}$$

and $\alpha = \alpha_s / h_{n+1}$.

A number of codes for ODEs and DAEs solve the sensitivity system (7), or its special case for ODEs, directly (see [5]). If the partial derivative matrices are not available analytically, they are approximated by finite differences. There are three well-established methods to solve the nonlinear system (8):

- Staggered direct method, described in [5].
- Simultaneous corrector method, described in [12].
- Staggered corrector method, described in [7].

An analysis and comparison of the performance for these three methods has been given in [7]. A detailed comparison of these methods applied to a special problem is also given in [15]. Here we discuss briefly the performance of the three methods.

The staggered direct method first solves equation (8) for the state variables. After the Newton iteration for the state variables has converged, the sensitivity equations in (8) are updated with the most recent values of the state variables. Because equation (8) is linear with a matrix J for the sensitivity equations, it is solved directly without Newton iteration. However, to solve the linear system in this way requires computation and factorization of the Jacobian matrix at each step and also extra storage for the matrix $\partial F / \partial y'$. Since the Jacobian is updated and factorized only when necessary in DASPK, the additional matrix updates and factorizations make the staggered direct method unattractive compared to the other methods. However, if the cost of a function evaluation is more than the cost of factorization of the Jacobian matrix, the staggered direct method is more efficient. We have modified the implementation of [5] to make the staggered direct method more reliable for ill-conditioned problems.

The simultaneous corrector method solves (8) as one whole nonlinear system, where Newton iteration is used. The full Jacobian matrix $\mathbf{J}$ in (9) is actually not computed. Instead, it is approximated by its block diagonal in the Newton iteration. Thus, this method allows the factored corrector matrix to be reused for multiple steps. It has been shown in [12] that the resulting iteration is two-step quadratically convergent for full Newton, and convergent for modified Newton iteration.

The staggered corrector method is similar to the staggered direct method. However, instead of solving the linear sensitivity system directly, a Newton iteration is used

$$s_i^{(k+1)} = s_i^{(k)} - J^{-1} G_{s_i}(s_i^{(k)}), \tag{10}$$

where G_{s_i} is the residual for the i-th sensitivity and J is the factored Jacobian matrix which is used in the Newton iteration for the state variables. Like the simultaneous corrector method, this method does not require the factorization of the Jacobian matrix at each step, and is a significant improvement over the staggered direct method. One of the advantages of the staggered corrector method is that we do not need to evaluate the sensitivity equations during the iteration of solving for the state variables. This can reduce the computation time if the state variables require more iterations than the sensitivity variables. After solving for the state variables in the corrector iteration, only the diagonal part of $\mathbf{J}$ in (9) is left. We can expect the convergence of the Newton iteration will be improved over that of using an approximate iteration matrix in the simultaneous corrector method. This has been observed in our numerical experiments.

Methods for evaluating sensitivity residuals

Several approaches have been developed to calculate the sensitivity residuals that may be used with either the staggered corrector or the simultaneous corrector methods. Maly and Petzold [12] used a directional derivative finite difference approximation. For example, the ith sensitivity equation may be approximated as

$$\frac{F(t, y + \delta_i s_i, y' + \delta_i s_i', p + \delta_i e_i) - F(t, y, y', p)}{\delta_i} = 0, \tag{11}$$

where δ_i is a small scalar quantity, and e_i is the ith unit vector. Proper selection of the scalar δ_i is crucial to maintaining acceptable round-off and truncation error levels, which was discussed in [12]. If $F(t, y, y', p)$ is already available from the state equations, which is the case in the Newton iteration of DASPK, (11) needs only one function evaluation for each sensitivity. The main drawback of this approach is that the accuracy of the method is hard to analyze. The smaller the perturbation δ_i, the lower the truncation error resulting from the omission of higher order terms, but the higher the loss-of-significance errors resulting from subtracting two nearly equal numbers. It is also hard to select a δ_i that is appropriate for all the variables for a badly scaled problem, because y, y' and p may have different scalings.

Recently, Alan Hindmarsh [10] proposed a different method to select the increment δ_i for equation (11). In [12],

$$\delta_i = \Delta \max(|p_i|, ||v_i||_2) \tag{12}$$

where Δ is a scale factor, and

$$v_i = \left(WT^j/WT^{in_y+j} : j = 1, ..., n_y\right).$$

The problem with (12) is that when some s_i^j is large, WT^{in_y+j} will be large and v_i^j will be small. However, $||v_i||_2$ might be large if v_i^k is large for some k. The perturbation $\delta_i s_i$ will be too large for the finite difference approximation. The new method proposed by Hindmarsh is to replace $||v_i||_2$ in (12) with $1/||u_i||_2$, *i.e.*

$$\delta_i = \Delta \max(|p_i|, 1/||u_i||_2) \tag{13}$$

where

$$u_i = \left(WT^{in_y+j}/WT^j : j = 1, ..., n_y\right).$$

If some s_i^j is large, WT^{in_y+j} will be large and u_i^j will be large. Then $1/||u_i||_2$ will definitely be small, leading to a more appropriate $\delta_i s_i$. This is the increment selection strategy that we use in DASPK3.0.

Another approach is to evaluate the sensitivity residuals analytically by ADIFOR or other automatic differentiation methods. If ADIFOR with the seed matrix approach is used, which we will discuss in Section 6, we can avoid evaluating the Jacobians $\partial F/\partial y$, $\partial F/\partial y'$ and $\partial F/\partial p$, and the matrix-vector product in (7). The equations for each sensitivity in (7) can be evaluated as one matrix-vector product in ADIFOR, which can substantially improve the efficiency.

Another approach is to evaluate the sensitivity equations in (7) directly. First the Jacobian ($\partial F/\partial y$, $\partial F/\partial y'$) is evaluated. Then two matrix-vector products and two vector additions in

(7) are performed. This method was suggested as the most desirable method by [7]. However, as we will see in Section 7, it is more efficient than ADIFOR with the seed matrix option only in certain circumstances, which depends on the cost of evaluation of $\partial F/\partial p$. $\partial F/\partial p$ can be evaluated either analytically (including the automatic differentiation method) or by finite difference, e.g.

$$\frac{\partial F}{\partial p} = \frac{F(t, y, y', p + \delta_i e_i) - F(t, y, y', p)}{\delta_i}. \tag{14}$$

Although (14) is similar to (11), the scaling problem is easy to resolve because only one parameter p requires scaling. However, (11) is less costly than (14) plus two matrix-vector products. Hence the directional derivative approach is the preferred method if the scaling of y', y and p is not an issue.

4 Consistent initial condition calculation for sensitivity analysis

In this section, we first extend the initialization algorithm of [4] to the sensitivity equations. Then we propose a simple algorithm to compute the consistent initial conditions for index-2 systems.

4.1 Consistent initial conditions for index-one problems

Suppose there is a parameter p in equation (5). The sensitivity problem becomes

$$\begin{aligned} f(t, u, v, u', p) &= 0, \\ g(t, u, v, p) &= 0, \\ \frac{\partial f}{\partial u} s_u + \frac{\partial f}{\partial v} s_v + \frac{\partial f}{\partial u'} s_{u'} + \frac{\partial f}{\partial p} &= 0, \\ \frac{\partial g}{\partial u} s_u + \frac{\partial g}{\partial v} s_v + \frac{\partial g}{\partial p} &= 0. \end{aligned} \tag{15}$$

The algebraic variables in equation (5) generate algebraic sensitivity variables in equation (15). Equation (15) also has the same index as (5).

Sensitivity has been used in optimization problems that require the derivatives with respect to the constraints, which are given by differential equations. Such an optimization problem can be formulated as

$$y' = F(t, y, p), \quad y(t_0) = y_0,$$

$$\begin{aligned} &\int_{t_0}^{t_{\max}} \Phi(t, y(t), p)dt \quad \text{is minimized,} \\ &g(t, y(t), p) \geq 0. \end{aligned}$$

This problem can be solved by an optimization code, e.g., SNOPT [8], which is based on sequential quadratic programming (SQP) methods. The SQP methods require a gradient and Jacobian matrix which are derivatives of the objective function and of the constraints with respect to the optimization variables. DASPKSO has been used to compute these derivatives [14]. One of the difficulties for the optimization problems is that the solution output from the optimizer does not satisfy the consistent initial conditions required by DASPK. The consistent initial conditions must be computed first before we start the next time step. An initialization algorithm for equation (5) can be used for this purpose.

There are two approaches to compute consistent initial conditions for the sensitivity analysis. One approach is to treat the sensitivity and state variables in the same way, and take them as one whole vector in the initialization computation. In this case, the sensitivity variables and state variables will be solved simultaneously during the initialization. Because algebraic state variables generate algebraic sensitivity variables, and differential state variables generate differential sensitivity variables, the user does not need to indicate which sensitivity variables are algebraic and which are differential. As we have done for the simultaneous corrector method, we use an approximate Jacobian (the diagonal blocks of the exact Jacobian) for this approach. That's why the nonlinear system may not converge for some problems.

The other approach is to compute the consistent initial conditions first for the state variables, and then for the sensitivity variables. This staggered approach usually converges faster than the first approach. Because an exact Jacobian for the sensitivities is used, this approach may solve problems that the first approach fails to solve.

We assume in our implementation that the sensitivity variables fall into the same class of initialization problems as the state variables. Both approaches are included in DASPK3.0. We recommend using the staggered approach if sensitivity is considered.

In DASPK3.0, we made two improvements over the algorithm of [4]. The first improvement is to compute the actual Jacobian for initialization problem I instead of using an approximate one as in [4]. This is because the the approximate Jacobian usually requires the initial artificial time step to be reduced several times to minimize the effect of unwanted part in $\partial F/\partial y$. Evaluation of the actual Jacobian can be easily done with the available information in DASPK, via either finite difference or automatic differentiation. With the actual Jacobian, the initialization problem I becomes solving a nonlinear system which does not require any time-step information. However, we have included as an option the original implementation of [4] in case the Jacobian has been input by the user.

The other improvement is to evaluate the norm of the error only for the unknown variables. DASPK2.0 evaluated the norm of $J^{-1}r$, where $J = \frac{1}{h}\partial F/\partial y' + \partial F/\partial y$ is the Jacobian and r is a vector of the error residuals, in the convergence test. This works fine during integration because y is the unknown variable. For initialization problem I, the error for y' should be scaled by $1/h$ and it may not satisfy the error tolerance. For example, given an equation $y' = 0.1$, the norm of the error in DASPK2.0 is $J^{-1}r = 0.1h$, which might satisfy the error tolerance when h is small, which means that y' might never get updated. However, the actual norm of the error for the unknown variable y' is 0.1. This has been fixed in DASPK3.0 by modifying the norm evaluation in the initializations.

4.2 Consistent initial conditions for index-two problems

With partial error control (excluding the algebraic variables from the error control), DASPK can solve Hessenberg index-2 problems with given consistent initial conditions. However, consistent initial conditions may not be readily available in many cases. The initialization method for index-one systems in DASPK2.0 [4] will not work for index-2 problems. Taking the Hessenberg index-2 problem as an example,

$$\begin{aligned} u' &= f(t,u,v), \\ 0 &= g(u), \end{aligned} \tag{16}$$

given the initial guess $(u_0',\ u_0,\ v_0)$, we cannot fix u_0 and calculate only u_0' and v_0 as for the index-one system if $g(u) = 0$ is not satisfied.

The objective for index-2 initialization is to compute a new triple $(\hat{u'}_0,\ \hat{u}_0,\ \hat{v}_0)$ that satisfies the constraints and consistent initial conditions. Since there are three unknown variables but only two equations, the problem is under-determined. In most cases, we are interested in finding the closest u to u_0, *i.e.*,

$$\min\{||u - u_0||^2 \quad | \quad g(u) = 0\}, \tag{17}$$

which becomes an optimization problem with equality constraints. (17) can be solved easily by many optimization methods. If it is solved by an SQP method, the following iteration is used,

$$u_i - u_0 + \delta u_i + g_u(u_i)^T(\lambda_i + \delta\lambda_i) \quad = \quad 0,$$

$$g(u_i) + g_u(u_i)\delta u_i \;\; = \;\; 0, \tag{18}$$

where u_i, λ_i are the current values for the solution u and Lagrange multiplier λ, and δu_i and $\delta\lambda_i$ are the increments for the next iteration. The iteration matrix for (18) is

$$\begin{pmatrix} I & g_u^T \\ g_u & 0 \end{pmatrix}, \tag{19}$$

which should be evaluated at each iteration. Although this method yields the closest initial conditions which satisfy the constraints, it is not efficient to implement in the DASPK environment. The difficulty is that the iteration matrix (19) is not available in DASPK. Another problem is that the solution computed by solving (17) may not satisfy other hidden constraints which are derived from the equations. In this sense, it may not be optimal.

Following the idea of [4] for index-1 problems, we attempt to solve the consistent initialization problem with the help of mechanisms that already exist in the DASPK solver. We search for the consistent initial conditions in the direction given by the differential equations. This method has a potential advantage that the hidden constraints derived from the equations may also be satisfied. To do that, we should increment the derivative u' by $\frac{1}{h}\delta u$ if the solution u is incremented by δu. Consider a general DAE system,

$$F(t, u, u', v) = 0. \tag{20}$$

After introducing two new variables δu and δv and an artificial time step h, we transform equation (20) into

$$F(t, u_0 + \delta u, u_0' + \frac{1}{h}\delta u, v_0 + \delta v) = 0. \tag{21}$$

δu and δv in (21) can be solved by Newton iteration with initial values of zero. The iteration matrix is

$$J = (\frac{1}{h}F_{u'} + F_u, F_v), \tag{22}$$

which is just the iteration matrix computed in DASPK. In DASPK, h is chosen to be the initial stepsize that satisfies the error tolerance for a zeroth order method.

It is easy to fix some of the differential variables in (21). However, fixing a differential variable u does not imply fixing the derivative u' in our algorithm, and vice versa. For example, if we fix the first element u_{01} of the vector u in (21), the equation becomes

$$F(t, u_{01}, u_{0r} + \delta u_r, u_0' + \frac{1}{h}\delta u, v_0 + \delta v) = 0,$$

where u_{0r} is the rest of u (excluding u_{01}), and δu_r is the rest of δu (excluding δu_1). In the algorithm of [4] for initialization problem I, all of the differential variables are fixed and equation (21) becomes

$$F(t, u_0, u_0' + \frac{1}{h}\delta u, v_0 + \delta v) = 0. \tag{23}$$

The initialization problem II in [4] can also be cast into (21) by fixing all of the derivatives u', which yields

$$F(t, u_0 + \delta u, u_0', v_0 + \delta v) = 0.$$

As mentioned in [4], the iteration matrix for (21) depends on which variables or derivatives have been fixed. The structure of the problem and of the Jacobian determines which variables or derivatives can be fixed. For index-0 and index-1 systems, when h is small in some appropriate sense and no derivatives have been fixed, the matrix computed by DASPK was shown in [4] to be a good approximation to the exact iteration matrix. If some of the derivatives have been fixed, the iteration matrix can change a lot or may even become singular. If all of the derivatives are fixed, as in initialization problem II in [4], the iteration matrix is computed in DASPK by setting CJ=0.

For the Hessenberg index-2 system (16), if no variables or derivatives are fixed, the iteration matrix is

$$J = \begin{pmatrix} \frac{1}{h}I - f_u & -f_v \\ g_u & 0 \end{pmatrix}, \tag{24}$$

which is just the iteration matrix computed in DASPK. In practice, we usually scale the bottom rows g_u of the matrix (24) by $1/h$ (see [2]), which yields

$$h\hat{J} = \begin{pmatrix} I - hf_u & -hf_v \\ g_u & 0 \end{pmatrix}. \tag{25}$$

This approach is similar to taking a small artificial time step with the backward Euler method except that the time is always fixed during the initialization. The stepsize h may be so small sometimes that it cannot reach the consistent initial conditions. Therefore, if the nonlinear system fails to converge, we should increase h rather than decrease it. This approach is used only when we update both the variables and derivatives. It is not applied to initialization problems I and II, where reducing the time step is used.

If the constraint $g(u) = 0$ is satisfied, all of the differential variables can be fixed, and equation (23) becomes

$$\begin{aligned} \delta u' + u_0' &= f(t, u_0, v_0 + \delta v), \\ 0 &= g(u_0). \end{aligned} \tag{26}$$

where $\delta u' = \frac{1}{h}\delta u$. Since $\delta u'$ and δv are not related to $g(u_0)$, the iteration matrix for (26) is singular, which means the solution is not unique. However, we can replace the constraint equation in (26) with $g_u u' = 0$, which yields

$$\begin{aligned} \delta u' + u_0' &= f(t, u_0, v_0 + \delta v), \\ 0 &= g_u(u_0)(\delta u' + u_0'), \end{aligned} \tag{27}$$

with iteration matrix

$$hJ = \begin{pmatrix} I & -hf_v \\ g_u & 0 \end{pmatrix}. \tag{28}$$

Matrix (28) can be approximated by (25) when h is small in some appropriate sense (see Theorem 4.1 in [4]). However, because the condition number of (28) is proportional to $1/h$, reducing the time step h will increase the round-off error. We observed in some problems that the nonlinear system (27) failed to converge with the approximate Jacobian (25) no matter how small h is. In those cases, the exact Jacobian (28) should be used. We actually use (28) inside DASPK3.0 for the Newton iteration of this initialization problem. The algebraic variables solved by this method also satisfy the derived constraint $g_u f(u, v) = 0$.

It is easy to evaluate the first equation in (27) by function evaluations in DASPK. However, the second equation is not available to DASPK. To evaluate it requires the user to specify which equations are algebraic. We can avoid evaluating $g_u u' = 0$ if $f(t, u, v)$ is a linear system with respect to v. Note that if system (27) is linear with respect to $u' = \delta u' + u_0'$ and $v = v_0 + \delta v$, it has a unique solution for u' and v. The u' and v can be solved via only one iteration for a linear system, independent of the initial values. If we set $u_0' = 0$ and $\delta u' = 0$ in our first guess, the value of the second equation in (27) is zero, which is also the result of $g(u_0)$. Therefore, the residual evaluations in DASPK can be used without modification. If $f(t, u, v)$ is not linear with respect to v, then it might take more than one iteration to solve for u' and v. Since $g_u u'$ might not be zero during the intermediate iterations, $g_u u'$ must be evaluated in addition to the residual evaluations of DASPK.

In most applications from mechanical systems, v is a Lagrange multiplier and $f(t, u, v)$ is linear in v. The method without evaluating $g_u u'$ can be used with a fixed initial value $u_0' = 0$. If $f(t, u, v)$ is not linear with respect to v, the user can either evaluate the $g_u u'$ in the residual routine or specify which equations are algebraic and DASPK will compute $g_u u'$ automatically, via finite difference approximation or automatic differentiation.

When solving the sensitivity equations with DAEs, we note that the sensitivity equations are always linear with respect to the sensitivity variables. Therefore, we can evaluate $g_u u' = 0$

only for the state equations if the staggered method is used. The sensitivities can be solved in one Newton iteration with initial guess $u' = 0$. In DASPK3.0, we have provided several options, which correspond to the options for evaluation of the sensitivity residuals, to calculate $g_u u'$. Even if there are no sensitivities in the system, the flag corresponding to the evaluation methods must be set if the evaluation of $g_u u'$ is necessary. Note that for the simultaneous method, $g_u u' = 0$ must be evaluated for each sensitivity to get the correct solution.

Which variables should be fixed is problem-dependent. For some mechanical systems, such as *trajectory prescribed path control* (TPPC) problems [2], it is desirable to fix the differential variables related to the path constraints in the computation of consistent initial conditions. If only a subset of the differential variables are fixed in (16), the iteration matrix (22), which is required in DASPK, may not be a good approximation to the exact iteration matrix. This is unlike the index-1 system where we can always use (22) no matter which variables are fixed. For an index-2 system like (16), where a subset of the variables are fixed, the iteration matrix (25) becomes

$$hJ = \begin{pmatrix} I & I - hf_{u_2} & -hf_v \\ 0 & g_{u_2} & 0 \end{pmatrix}. \tag{29}$$

where $u = (u_1, u_2)$ and u_1 is fixed. Note that (29) cannot be approximated by (25) no matter how small h is. We observed in numerical experiments that the Newton iteration can fail with the approximate iteration matrix (25). In DASPK3.0, we evaluate the Jacobian (29) instead of (25), via finite difference approximation or automatic differentiation, if only a subset of the differential variables are fixed. If the Jacobian is provided by the user, (29) should also be calculated.

To sum up, the initialization algorithm for Hessenberg index-2 problems is similar to that for index-1 problems. Which variables should be fixed is determined by both the structure of the system and the available information. Although our algorithm for index-2 problems is also valid for index-1 problems, we recommend fixing all of the differential variables and using the initialization algorithm of [4] for most index-1 problems. For an index-2 problem, if the constraints are satisfied, we recommend fixing all of the differential variables and using (27) to compute the derivatives and algebraic variables. For a mixed index-1 and index-2 problem, if the constraints related to the index-2 variables are satisfied, we recommend fixing all of the differential variables. Otherwise we can fix only a subset of the differential variables, or do no fixing at all. Note that this will alter the initial values of the differential variables which may not be desirable depending on the problem.

If only a subset or none of the differential variables are fixed for an index-2 system, we use a two-step process to improve the results for the consistent initial conditions: a predictor step followed by a corrector step. In the predictor step, we calculate a consistent u', u and v with no fixing or partial fixing. After the predictor step, u is often close to u_0 and satisfies the constraint $g(u) = 0$. However, v is usually far away from v_0 because of a small value of h, and the derived constraint $g_u(u)f(t, u, v)$ is also severely violated. In the corrector step, we fix all of the differential variables u, and use (27) to compute u' and v again. In the corrector step, we have reset the values of the derivatives either to zeros or to their original values (the initial guess before the predictor step), depending on whether the system is linear or nonlinear with respect to the algebraic variables.

In our implementation, a linesearch backtracking algorithm [4] has been used to improve the robustness of the Newton algorithm for the initial condition calculation.

4.3 How to compute consistent initial conditions in DASPK3.0

We have modified the DASPK2.0 [4] code to include initialization for index-2 problems. We assume here that the reader is familiar with the use of DASPK2.0 [4]. To be compatible, we have retained all the options from DASPK2.0. By specifying the input parameter INFO(11), DASPK3.0 will solve initialization problems with different approaches.

If the initial values are already consistent, set INFO(11) = 0. This is the default.

For most problems (index-0, index-1, or index-2 where it is not essential to satisfy the derived constraints), use one of the following options depending on which initial conditions have been specified in the problem:

- INFO(11) = 1. Solve initialization problem I: Given Y_d, calculate Y_a and Y'_d. Y_d is fixed during the initial condition calculation. This option is applicable for index-1 problems, and for Hessenberg index-2 systems if the original constraints are satisfied. For Hessenberg index-2 systems, it will yield consistent initial conditions that satisfy the derived differentiated constraint if the system is linear with respect to the algebraic variables, and the initial guess for y' is set to 0. If this option is specified, the user must identify for DASPK the differential and algebraic components of Y (for state variables only). This is done by setting (for I=1,...N)
  - IWORK(40+I) = +1 or +2 or +3 if Y(I) is a differential variable, and
  - IWORK(40+I) = −1 if Y(I) is an algebraic variable.

- INFO(11) = 2. Given Y', calculate Y. Y' is fixed during the calculation. This option is applicable for any problem as long as the resulting initialization problem is well-posed.

- INFO(11) = 3. Given Y and Y', calculate a new Y and Y'. This option is applicable for any problem. INFO(11)=1 and INFO(12)=2 can be transformed into this option by setting IWORK properly. Algebraic variables are never fixed for this option. A subset of differential variables or their derivatives can be fixed by setting:

  - IWORK(40+I) = +1 if Y(I) is a differential variable, and both Y(I) and Y$'$(I) are free;
  - IWORK(40+I) = +2 if Y(I) is a differential variable, and Y(I) is fixed but Y$'$(I) is free;
  - IWORK(40+I) = +3 if Y(I) is a differential variable, and Y$'$(I) is fixed but Y(I) is free;
  - IWORK(40+I) = −1 or −2 if Y(I) is an algebraic variable.

For index-2 problems where it is important that not only the original constraint but also the derived constraint be satisfied at the initial values, the user must specify which equations are index-2 constraints so that Jacobian (28) can be used. This can be done by setting

- IWORK(40+N+I) = 1, if the I-th equation is an index-2 constraint;

- IWORK(40+N+I) = 0, otherwise.

There are two options for solving this type of initialization problem, depending on whether the system is linear with respect to the algebraic variables:

- INFO(11) = 4. This option is applicable for systems that are linear with respect to the index-2 algebraic variables.

- INFO(11) = 5. This option is applicable whether or not the system is linear with respect to the index-2 algebraic variables. The user must specify which method is used to evaluate $g_u u'$ by setting INFO(20) correctly. We will discuss how to set INFO(20) in Section 7.

Option INFO(11)=5 is available only for the staggered method, where the state variables are solved first and then the sensitivity variables. It might not work effectively for the simultaneous method.

5 New features of DASPK3.0

Apart from the initialization algorithm, we have incorporated several other new features for sensitivity analysis into DASPK3.0.

5.1 Error test and convergence test

In DASPKSO [12], the state variables and sensitivity variables are considered as one whole vector when the error or residual norm is computed. The norm that DASPK uses is a weighted root mean square norm (WRMS), given by

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

where NEQ is the number of equations and $WT_i = \mathrm{RTOL}_i|Y_i| + \mathrm{ATOL}_i$. If the number of state variables is NY and the number of sensitivity variables is NP, NEQ will be (NP+1)·NY. When NP is large and partial error control is used (excluding the sensitivity variables), the norm for the state variables will be reduced by a factor of 1/(NP+1), which is inappropriate for the error and convergence tests. Even with full control (the sensitivity variables are included), the norm of the state variables will be affected by the norm of the sensitivity variables. Each sensitivity component also affects the others.

In DASPK3.0, we evaluate the norm separately for the state variables and for the sensitivities with respect to each parameter. This does not increase the computational work because the sum is over only NY variables instead of NEQ variables. For the error and/or convergence tests, we choose the largest one among all the norms. This also makes it easier to do partial error control.

For most problems we have tested, the results of partial error control, which excludes the sensitivities from the error tests, and full error control, which includes the sensitivities in the error tests, are almost the same. However, partial error control can speed up the computation. If you have no idea on the error tolerance of the sensitivity variables or the accuracy of the sensitivity variables is less than that of the state variables, it may be advantageous to use partial error control. In the convergence tests of the Newton iterations, all of the variables must be included in the test to get a reliable result.

5.2 New implementation of Krylov method

In DASPKSO [12], the iteration matrix is approximated by its block diagonal in the Newton iteration for the direct method, because the error matrix is nilpotent. In the Newton-Krylov iteration, the matrix-vector product in (4) is approximated via a difference quotient on the functions. Therefore, the iteration matrix $\mathbf{J}$ of (9) can be used instead of its block diagonal part. For example, for the ith sensitivity equation, we have

$$(Av)_i = J_i v_y + J v_{s_i} \simeq \frac{F_i(t, Y + \sigma v, \alpha(Y + \sigma v) + \beta, p) - F_i(t, Y, \alpha Y + \beta, p)}{\sigma}, \qquad (30)$$

where $Y = (y, s_1, ..., s_{n_p})$, $J = \alpha F_{y'} + F_y$, $J_i = \frac{\partial J}{\partial y} s_i + \frac{\partial J}{\partial p_i}$, $F_i = F_{y'} s_i' + F_y s_i + F_{p_i}$ is the ith sensitivity equation, and $v = (v_y, v_{s_1}, ..., v_{s_{n_p}})$. This has been implemented in DASPKSO [12]. The matrix-vector products in (30) require the evaluation of the sensitivity equations, which is usually more expensive than the evaluation of the state equations. Since the state variables and sensitivity variables are solved simultaneously in DASPKSO, the length of an orthonormal basis of the Krylov subspace for the coupled system is NEQ=(NP+1)*NY. Therefore the storage of the orthonormal basis requires more space than for the state variables. Although this scheme may be better for vector computation, it may not be advantageous for parallel computation because we must parallelize the GMRES iterative method. It is also difficult to implement the staggered corrector method.

For the staggered corrector method, only $J v_{s_i}$ is left in (30) for Newton iteration of the sensitivity equations. Although (30) can still be used to evaluate $J v_{s_i}$, it takes no advantage of the reduced structure. We can evaluate $J v_{s_i}$ directly via directional derivative finite difference approximation

$$J v_{s_i} = (\alpha F_{y'} + F_y) v_{s_i} \simeq \frac{F(t, y + \sigma v_{s_i}, \alpha(y + \sigma v_{s_i}) + \beta, p) - F(t, y, \alpha y + \beta, p)}{\sigma}, \qquad (31)$$

where $F(t, y', y, p)$ are the state equations. The function evaluations in (31) involve only the state equations, and no evaluations of sensitivity equations are required during the linear iteration for sensitivity variables. Because there is no coupling between different sensitivity variables, the linear iteration for each sensitivity equation can be done separately, which allows us to split the large linear system in (30) into several small ones and reduce the length of each orthonormal basis to NY.

For the simultaneous corrector method, we can approximate the Newton-Krylov iteration matrix by its block diagonal as for the direct method. Then (31) can be used to calculate the matrix-vector product. This implementation not only sharply reduces the storage needed for the orthonormal basis in the case of a large number of sensitivity parameters, but also improves the computational efficiency. Parallel computation is also easy to implement.

In the above discussion, we assume that evaluation of the sensitivity equations is more costly than evaluation of the state equations, which is true for most problems because the evaluation of the sensitivity equations requires the Jacobian information from the state variables. However, for a large number of sensitivity parameters, the Jacobian is evaluated only once for all the sensitivities, and the average cost for each sensitivity may be less than for the state equations. This inspires us to evaluate the Jv_{s_i} directly by matrix times vector instead of by finite difference. For problems with a large number of sensitivity parameters (larger than the average number of non-zero elements in a row of the Jacobian) and where the state equations are very costly to evaluate, the computation time can be sharply reduced by the matrix times vector method.

5.3 Implementation of staggered direct method

The staggered direct method has been implemented in the DASAC code by Caracotsios and Stewart [5]. In DASAC, system (8) is transformed into

$$Js_{i_{(n+1)}} = \left(-\frac{\partial F}{\partial y'_{n+1}}\beta - \frac{\partial F}{\partial p_i}\right), \tag{32}$$

where $\beta_i = {s'}^{(0)}_{i_{(n+1)}} - \alpha s^{(0)}_{i_{(n+1)}}$. To solve a linear system in this way requires extra storage for the matrix $\partial F/\partial y'_{n+1}$. Moreover, this implementation often fails when the matrix J is ill-conditioned. This is because the right-hand side of equation (32) can be very large and can introduce large round-off errors when J is ill-conditioned [11].

In DASPK3.0, the following linear system is solved for the sensitivities:

$$J\delta = Js^{(0)}_{i_{(n+1)}} + \frac{\partial F}{\partial y'_{n+1}}\beta + \frac{\partial F}{\partial p_i}, \tag{33}$$

where $\delta = s^{(0)}_{i_{(n+1)}} - s_{i_{(n+1)}}$. The right-hand side of (33) is easy to obtain in DASPK3.0 by the function evaluations of the sensitivity equations. It does not require any extra storage or special handling. What is important is that it works well for ill-conditioned problems. The reason is because the right-hand side of equation (33) is usually much smaller than that of equation (32) for a successful step (which means the predictor value $s^{(0)}$ is close enough).

5.4 Implementation of staggered corrector method

The staggered corrector method has been implemented in DASSL in [7] and compared with the simultaneous corrector method. In the following, we discuss how it is implemented in DASPK3.0.

The direct method in DASPK is virtually identical to DASSL. Therefore, it is not difficult to implement the staggered corrector method. We use almost the same algorithm as Feehery et al. have done for the DASSL code [7]. However, it is more difficult to modify DASPK for the staggered corrector method because the DASSL routine has been split into several smaller ones in DASPK. Several features are designed to minimize wasted computations due to corrector convergence failure or error test failure. An error test is performed on the state variables before the sensitivity corrector iteration is started, because an error failure on the state variables will definitely cause an error test failure in the whole system. Because we use the separate evaluation of the norms (see Section 5.1), the error test for the state variables is performed only once during a one step integration. Our new implementation of the Krylov method for the sensitivity analysis in Section 5.2 is similar to that of the direct method. Therefore the implementation for the staggered corrector is almost a copy from the direct method.

We also used different methods from [7] to evaluate the sensitivity residuals. The directional derivative option in [12] is retained as one of the methods. Where ADIFOR is available, ADIFOR with the seed matrix option or with the matrix-vector product only option is used to compute the sensitivity residuals. The matrix times vector method is also available as an option: first the matrix $(\partial F/\partial y', \partial F/\partial y, \partial F/\partial p)$ is computed after convergence of the corrector iteration for the state variables; then the matrix times vector is used to evaluate the sensitivity residuals during the iteration for the sensitivity variables.

The overall algorithm for the staggered corrector method is as follows. First we do the corrector iteration only for the state variables. The iteration is continued until it converges or it reaches a maximum number of iterations. If it does not converge within the maximum number of iterations, the iteration matrix is reevaluated and refactored if the iteration returns with a recoverable error. Otherwise the iteration will return with a convergence error and the integration may start again with a reduced time step. If the iteration for the state variables converges, the error test is performed on the state variables. If the error test fails, the iteration will return with an error test failure. After the state variables pass the convergence test and error test, we compute residuals for the sensitivity equations. The residuals for the state variables are also updated for the Krylov method or if finite-differencing is used to compute the sensitivity equations. Then the sensitivity corrector equation is solved in the same manner as the state variable corrector equation. Provision is made to update and refactor the iteration matrix if the corrector iteration is not converging. After the sensitivity variables have passed the corrector convergence test, an error test may be performed on the sensitivity system if full-error control is selected. If this test fails, the step size is reduced and/or the corrector matrix refactored and the step is attempted again.

For the Krylov method, the nonlinear system (8) is solved by full Newton iteration. For a linear system, it should yield the results in one iteration. Therefore, the staggered direct method (or staggered Krylov method) is essentially the same as the staggered corrector method in this case.

In practical problems, evaluation of the residuals for the sensitivity variables is usually accompanied by evaluation of the residuals for the state variables. This is especially true when the ADIFOR package is used to compute the sensitivity equations. Therefore, the residuals for the state variables may also be evaluated during the sensitivity corrector iterations, although it is not necessary. This is why the staggered corrector method may be slower than the simultaneous corrector method in some special cases. In most cases, even if the residuals for the state variables are evaluated during the sensitivity corrector iteration, the staggered corrector method has better performance than the simultaneous corrector method. We will provide some examples to demonstrate this in Section 9.

Apart from using the staggered method in regular integration, we also use it in the initialization algorithm of Section 4.

6 ADIFOR and DASPK3.0

ADIFOR is an automatic differentiation tool for FORTRAN programs, developed recently at Argonne National Laboratory and Rice University [1]. It adopts a hybrid approach to computing derivatives that is generally based on the forward mode, but uses the reverse mode to compute the derivatives of assignment statements containing complicated expressions. The forward mode acts similar to the usual application of the chain rule in calculus. Derivatives of the intermediate variables with respect to input variables are computed and are propagated forward through the computational stages. The reverse mode is based on the adjoint quantities representing derivatives of the output with respect to intermediate variables. The adjoints are computed at each node of the computational graph. They are propagated by reversing the flow of the program and by recomputing intermediate values that have a nonlinear impact on the output.

ADIFOR requires the user to supply the FORTRAN source code for the function value and for all lower level subroutines as well as a list of the independent and dependent variables in the form of parameter lists or common blocks. ADIFOR can determine which other variables throughout the programs are to be differentiated, and augments the original code with derivative statements. The augmented code is then optimized by eliminating unnecessary operations and temporary variables. The FORTRAN code generated by ADIFOR requires no run-time support and therefore can be ported between different computing environments. More information can be obtained via the World Wide Web (http://www.mcs.anl.gov/Projects/autodiff).

In this section, we study how to incorporate ADIFOR-generated derivatives into DASPK3.0. First, we list the places that require derivative evaluations

- Jacobian evaluation for the direct method,
- matrix-vector product for the Krylov iteration,
- Jacobian evaluation for incomplete LU factorization (ILU) preconditioner,
- evaluation of sensitivity equations in DASPKSO.

Where ADIFOR is available to the user, it might be better to replace all the finite differencing with the ADIFOR-generated routines. However, there is a trade-off when we consider the efficiency and accuracy of the computations.

The ADIFOR-generated routine not only computes the derivatives but also the original functions. To compute one matrix-vector product in an ADIFOR-generated routine requires at least one evaluation of the original function and possibly more than one evaluation of the derivatives. But the matrix-vector product approximated by first-order finite difference requires only one evaluation of the original functions. Since the finite difference approximation in the matrix-vector product for the Krylov iteration has incorporated the scaling into its implementation, it yields an acceptable result in most cases. Because the matrix-vector product is used in each linear iteration in the Krylov method, if we use the ADIFOR-generated routine to replace it, it will take much more computation time. This is why we do not use it in DASPK3.0.

Although evaluation of the sensitivity equations is also a matrix-vector product in ADIFOR, the scaling problems in (11) are not easily resolved. We tried to incorporate scaling for the finite difference approximation of sensitivity equations but failed. For badly-scaled problems, the finite difference approximation cannot give an acceptable result. We recommend using ADIFOR to evaluate the sensitivity equations. Even for some well-scaled problems, the ADIFOR-generated routine has better performance in terms of efficiency and accuracy than the finite difference approximation.

The ADIFOR-generated Jacobian has proven to be more efficient and accurate than the finite difference method. We use it in DASPK3.0.

There are several options in the ADIFOR script file to generate derivative code:

- matrix-vector product only, invoked with AD_SCALAR_GRADIENTS = true;
- seed matrix input, invoked with AD_FLAVOR = dense;
- sparse linear combination (SparsLinC), invoked with AD_FLAVOR = sparse;

Each option works better than others under certain circumstances. The matrix-vector product option has the best performance if only one matrix-vector product is computed. This option is also chosen for parallel computation if you do not want to modify the ADIFOR-generated routine for parallel use. If more than one matrix-vector product has to be computed, the seed matrix option works better than the matrix-vector product option. For the Jacobian evaluation, the SparsLinC option has the best performance for a large sparse matrix. If the Jacobian is a dense matrix or a banded dense matrix (here a matrix is called dense matrix if it has at least 50% of nonzero elements), the seed matrix option works better than the SparsLinC option.

For the evaluation of the sensitivity equations, we offer three options with ADIFOR. The matrix-vector product only and seed matrix options have been discussed in the last paragraph. Another option provided is direct evaluation of the sensitivity equations by matrix times vector: first we evaluate the Jacobian ($\partial F/\partial y$, $\partial F/\partial y'$, $\partial F/\partial p$) by ADIFOR, usually with the SparsLinC option; then we evaluate

$$\frac{\partial F}{\partial y}s_i + \frac{\partial F}{\partial y'}s_i' + \frac{\partial F}{\partial p}$$

by two matrix-vector products and two vector additions. This option can be more efficient than the other two options for the staggered corrector method when $F(t, y, y', p)$ is expensive to evaluate.

To compare the costs of the different methods for sensitivity evaluations, we consider the problem of evaluating the Jacobian J of a vector function F with respect to an n vector of variables y. The cost of evaluation of the function F is $\mathrm{COST}(F)$. The cost of evaluating J is related to $\mathrm{COST}(F)$ by

$$\mathrm{COST}(J) \simeq a \cdot n \cdot \mathrm{COST}(F),$$

where $a = 3$ for the basic forward mode of automatic differentiation. If only a product of the Jacobian with some vector p is required, the cost of $J \cdot p$ is

$$\mathrm{COST}(J \cdot p) \simeq a \cdot \mathrm{COST}(F).$$

If the sparse forward mode (with SparsLinC option) is used in automatic differentiation, the cost satisfies

$$\mathrm{COST}(J) \simeq a \cdot n' \cdot \mathrm{COST}(F),$$

where n' is the maximum number of nonzero entries in any row of the Jacobian. Suppose the average number of nonlinear iterations is n_i, and the number of sensitivity parameters is n_p. Then the cost of ADIFOR with the seed matrix option includes n_p matrix-vector products for each nonlinear iteration. The total cost can be approximated by

$$n_i n_p \cdot a_1 \cdot \mathrm{COST}(F).$$

The cost of direct evaluation by matrix times vector method includes the cost of evaluation of the Jacobian and the two matrix times vector operations, which is

$$a_2(m' + n_p') \cdot \mathrm{COST}(F) + n_i n_p m' n,$$

where m' and n_p' are the maximum number of nonzero entries in any row of the Jacobian $(\partial F/\partial y,\ \partial F/\partial y')$ and $(\partial F/\partial p)$ respectively. We use a_1 and a_2 here to distinguish the coefficients for two different approaches because a_1 is usually much larger than a_2. The matrix times vector method is better when

$$(m' + n_p')a_2 \cdot \mathrm{COST}(F) + n_i n_p m' n < n_i n_p a_1 \cdot \mathrm{COST}(F),$$

which results in

$$n_p > \frac{(m' + n_p')a_2}{n_i a_1}, \tag{34}$$

and

$$\mathrm{COST}(F) > \frac{n_i n_p m' n}{n_i n_p a_1 - (m' + n_p')a_2}. \tag{35}$$

Equations (34) and (35) imply two conditions for the matrix times vector method to be advantageous over the seed matrix option. First n_p must be large enough. For example, if a_1=a_2, $n'_p = n_p$, and $n_i = 2$, then $n_p > m'$, which means the number of sensitivity parameters must be larger than the maximum number of nonzero entries in any row of the Jacobian ($\partial F/\partial y$, $\partial F/\partial y'$). The second condition is that the evaluation of the function F must be costly enough, which is defined by (35).

For the evaluation of the Jacobian matrix, we assume the user is familiar with DASSL and the previous version of DASPK. The Jacobian matrix is required only for the direct method. INFO(5) in DASPK is an indicator for which method is used:

- INFO(5)=0, finite-differencing is used;
- INFO(5)=1, Jacobian is evaluated by user-supplied routine JAC;
- INFO(5)=2, ADIFOR with the SparsLinC option is used;
- INFO(5)=3, ADIFOR with the seed matrix option is used.

The evaluation method of the Jacobian for the calculation of the consistent initial condition is different from that for the integration inside DASPK. When the Jacobian is supplied by the user, we provide a flag in JAC for the user to determine what kind of Jacobian should be provided. The user can see the documentation of DASPK3.0 for more details. Because the ILU preconditioner for the Krylov method works efficiently only for a sparse Jacobian matrix, we use ADIFOR with the SparsLinC option to evaluate it.

We have provided several script files in our documentation on how to generate the derivative routines via ADIFOR. One problem when using ADIFOR in DASPK3.0 is that the user supplied vector RPAR(*) may sometimes depend on Y or YPRIME, which could generate different argument lists in the derivative routine. We handle this problem by adding an indicator of whether or not there is a data dependence between RPAR and Y or YPRIME. If there is, then IWORK(38) is used to store the dimension of RPAR.

One thing we should consider when we select ADIFOR options is the requirement for work space. Usually, the matrix-vector product only option requires the least extra work space among the three options. The SparsLinC option requires some extra integer work space (about 3*NY) to store the pointers. The seed matrix option, however, requires huge extra temporary work space. Suppose the bandwidth of the Jacobian is L=MU+ML+1. For Jacobian evaluation, it needs L*(3*NY+IWORK(38)); for sensitivity evaluation, it needs

NPAR*(3*NY+IWORK(38)). When L or NPAR is large, this temporary work space dominates the work space for other uses. We must be careful when we use the seed matrix option for machines with less memory.

All of the ADIFOR-generated routines require the support of the ADIFOR library, which is part of the ADIFOR package [1]. Since ADIFOR users must get the license first (free for academic use) from the developer, we do not provide the ADIFOR library as part of the DASPK packages. We have designed DASPK3.0 in such a way that it is stand-alone and independent of other libraries. However, incorporating the ADIFOR features (especially the SparsLinC option) into DASPK must include some routines defined in the ADIFOR library. For convenience of the user, we provide dummy routines for users that do not have access to the ADIFOR library.

7 Using DASPK3.0 for sensitivity analysis

Since DASPK2.0 [4] adds some options for consistent initial condition computation and error control, the INFO(*) for DASPK3.0 is totally different from that of DASPKSO. We assume here that the reader is familiar with the use of DASPK2.0 [4]. DASPK2.0 [4] used 18 of the total length of INFO(*). The sensitivity options of DASPK3.0 start from the 19-th.

- INFO(19) is used as the sensitivity toggle.
  - INFO(19)=0. No sensitivity is evaluated (default).
  - INFO(19)=NP. There are a total of of NP sensitivity parameters. There are two types of sensitivity parameters: the parameter appears in RES and the parameter appears only in the initial conditions. To evaluate the sensitivity equation, only the former type of parameters need to be stored for later use. They are stored in SENPAR(*) (this array is added into DASPK3.0). The number of this type of parameter is specified in INFO(22).
- INFO(20) allows several options for computing the sensitivity equations. Options for calculating $g_u u'$ for the initialization of index-2 systems are also available via INFO(20).
  - INFO(20)=0. Central differencing is used(default).
  - INFO(20)=1. One-sided forward differencing is used.

– INFO(20)=2. The residual is computed by the user-supplied routine RES. The parameter IRES in the RES routine is used to determine whether to compute the sensitivities or not. The user must specify in the RES routine as follows

if (ires == 0) then compute residuals for the state equations;

else if (ires == 1) then compute residuals for both state and sensitivity equations;

else if (ires == 2) then compute residuals for the state equations with $g_u u' = 0$ in place of $g(u) = 0$ (only for index-2 initialization).

end if

– INFO(20) = 3. The residuals of state and sensitivity equations are computed by ADIFOR with the seed matrix option.

– INFO(20) = 4. The residuals of state and sensitivity equations are computed by ADIFOR with the matrix-vector product option.

– INFO(20) = 5. The residuals of sensitivity equations are computed by the matrix times vector method. This option is only for the staggered corrector method, *i.e.*, when INFO(25)=1. First the matrices are computed by ADIFOR. Then the residuals are evaluated by the matrix times vector method.

If the ADIFOR option (INFO(20)>2) is chosen, the user must specify IWORK(38) regarding whether RPAR depends on Y, YPRIME or SENPAR. *If RPAR is used to pass values of Y, YPRIME or SENPAR between procedures, set IWORK(38) to be the dimension of RPAR(*).* Otherwise set IWORK(38)=0.

- INFO(21) acts as a perturbation factor option. This option is used only when INFO(20)<2 (finite difference options for sensitivities).

  – INFO(21)=0 (default). The default value 1.0D-3 is used.

  – INFO(21)=1. The user must supply the desired value in RWORK(16).

- INFO(22) stores the number of parameters that appear in the RES routine. This is usually the size of SENPAR(*).

- INFO(23) is used as an error control option for the sensitivity variables.

  – INFO(23)=0. The sensitivities are included in the error test.

  – INFO(23)=1. The sensitivities are excluded from the error test.

Note that all variables will be included in the Newton convergence test.

- INFO(24) is used if the sensitivity of a derived quantity is to be computed. In addition to computing the sensitivities of the solution and its derivative with respect to the parameters, the user may want to compute the sensitivity of a quantity $Q(t, y, y', p)$ with respect to the parameters. To do this, the user can call the auxiliary routine DSENSD

```
      CALL DSENSD(QRES, NEQ, T, Y, YPRIME, QSEN, INFO,
     *     RWORK, IWORK, RPAR, IPAR, SENPAR)
```

  in between calls to DASPK to find the sensitivity of the derived quantity Q at the output points of DASPK. The user can refer to the documentation of DSENSD for more details.

  – INFO(24)=0 (default). DSENSD is not called.
  – INFO(24)=NQ, where NQ is the dimension of the vector function Q if DSENSD is called.

- INFO(25) is used for sensitivity method options. This option applies also to the initial condition calculation.

  – INFO(25)=0 (default). Simultaneous corrector method is used.
  – INFO(25)=1. Staggered corrector method is used.
  – INFO(25)=2. Staggered direct method is used.

The work space required by DASPK3.0 is different from that of DASPK2.0 or DASPKSO if sensitivities are computed or the ADIFOR options are selected. The user can refer to the documentation of DASPK3.0 for details.

In DASSL and DASPK, there is an option for constraints on variables which are always nonnegative (or non-positive). When we combine the code with sensitivity analysis, it is unrealistic for us to tell which sensitivities are always nonnegative (or non-positive). Therefore, the constraints at the initial conditions and each time step are enforced only on the state variables, not on all the variables as in DASPKSO. The user can specify constraints for state variables only.

DASPK3.0 is compatible with the previous version of DASPK, *i.e.*, a previous code that works under DASPK2.0 should work well under DASPK3.0 when the INFO array is increased to 30 and is set to zero after the 18th element. However, DASPK3.0 is not compatible with DASPKSO. DASPK3.0 has a big difference from DASPKSO in storing the sensitivity parameters. We use SENPAR to store the sensitivity parameters separately in DASPK3.0.

8 Parallel implementation of DASPK for sensitivity analysis

Several parallel implementations for sensitivity analysis of DAEs have been compared in [15]. In this section, we describe the parallelization in DASPK3.0. Although all the tests in [15] are for DASSL with the direct method, the comparative results are similar for DASPK3.0 with both the direct method and the Krylov method with the new implementation described in Section 5.2. We have experienced that the distributed parameter only (DPO) approach of [15] is also the fastest for DASPK3.0. Therefore, we use this method in the parallel implementation of DASPK.

The implementation of DPO in [15] requires no modification to the DASPK code. However, it does require the user to distribute the parameters to each processor. The DPO method in [15] requires the user to distribute the parameters to each processor before calling the DASPK program, which is unrealistic or a big burden for an inexperienced user. Because the distribution is done outside of DASSLSO in [15], it is difficult to have centralized control over the computations, which is needed if we require to have the same stepsize for all the processors. Another difficulty in locating the equi-distribution outside of DASPK is that the user must define a different problem (RES in DASPK) for sensitivity analysis in each processor, which is inconvenient and error-prone.

Our implementation distributes the sensitivity parameters inside the DASPK code so as to reduce the burden on the user. To balance the workload between processors, we allocate the parameters randomly to each processor: if we have NP processors and NPAR parameters, N=NPAR/NP, we distribute parameter numbers

$$\begin{cases} j, ..., j + i * \mathrm{NP}, ..., j + N * \mathrm{NP}, & \text{if } j \leq \mathrm{mod(NPAR,NP)} \\ j, ..., j + i * \mathrm{NP}, ..., j + (N-1) * \mathrm{NP}, & \text{if } j > \mathrm{mod(NPAR,NP)}. \end{cases} \tag{36}$$

to the j-th processor. Each processor computes the state variables locally, and the Jacobian matrix is also computed and factorized locally when needed. To minimize the storage and memory requirements in each processor, we assume that each processor has distributed memory, *i.e.*, each processor has a local value of the same variable. Therefore, the work space in each processor can be reduced to approximately 1/NP of the total work space. Since the sensitivities are independent of each other, each processor can work independently without communicating with the others.

In our implementation, we initially parallelized all the loops related to computation of Y(*) and YPRIME(*). The values of Y(*) and YPRIME(*) were stored in their original positions and different processors. This method does not need to redistribute the initial conditions but requires much recoding from the serial version. In the revised version, we have removed the sensitivity variables from Y(*) and YPRIME(*) (they may not be adjacent in each processor), and restored them consecutively right after the state variables in each processor. However, we did not change other public variables such as RPAR(*), IPAR(*) and SENPAR(*) in DASPK. In this way, the main body of the code does not require modification if we set the number of equations NEQ and the number of sensitivity parameters INFO(19) properly inside DASPK. The initial conditions, however, must be redistributed accordingly in the first call of the program.

We have attempted to develop a code for which both parallel and serial computation can run efficiently. We enforce the same stepsize control for all the processors in the parallel implementation. The communication overhead is very small. In each time step, each processor may be using different orders of the BDF formulae. Since this implementation requires an MPI-related routine and the support of the MPI library, which may not be accessible by users doing serial computation, we provide a dummy routine, which can be linked without involving the MPI library, for use with serial computation

How to use DASPK with parallel computation

MPI was developed by researchers at Argonne National Laboratory and Mississippi State University [6]. Programs written with MPI are portable between different machines. A tutorial on how to use MPI to write parallel programs is given in [9]. The MPI packages, including the library and running examples, can be freely downloaded from

```
http://www.mcs.anl.gov/mpi
```

We assume here that the reader has a basic knowledge of MPI.

When MPI starts, it calls set-up routines to assign an identity to each processor. The identities are generated by MPI and form a consecutive integer array that starts from 0.

For example, if we have 10 processors, MPI will generate identities 0,1,...,9 and assign them to each processor randomly. The identity is used in DASPK3.0 to assign different sensitivity parameters to each processor as (36). Usually three MPI set-up routines must be called before calling DASPK3.0 to do parallel computation:

```
CALL MPI_INIT(IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, MYID, IERR)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, NUMPROCS, IERR)
```

where MYID is the identity of the current processor, and NUMPROCS is the number of processors that participate in the parallel computing.

We have added two more parameters in DASPK specifically for parallel computation with MPI methods. The user must set them to a correct value before calling DASPK3.0.

- INFO(26) – identity of the processor. The default value is 0, which is for serial computation.
- INFO(27) – number of processors participating. The default value is 0 and it is altered to 1 by DASPK.

For MPI users, INFO(26) is MYID, and INFO(27) is NUMPROCS, both available from the set-up routines of MPI. DASPK3.0 can run on serial machines by setting INFO(26)=INFO(27)=0.

9 Numerical Experiments

In this section, we test and justify our software with several examples. All tests were run on an SGI O2 workstation. The following quantities are used to compare different methods:

METH	Integration method
NSTP	Number of time steps used
NRES	Number of calls to residual subroutine
NSE	Number of sensitivity evaluations
NJAC	Number of Jacobian evaluations
NNI	Number of nonlinear iterations
NLI	Number of linear iterations (only for Krylov method)
CPU	The total cpu time taken to solve the problem

The integration methods we use include the direct method (D) and Krylov method (K). The integration methods for the sensitivity equations include the staggered corrector method (ST), the staggered direct method (SD) and simultaneous corrector method (SI). Therefore we use STD to represent the staggered corrector direct method, STK to represent the staggered corrector Krylov method, SID to represent the simultaneous corrector direct method, SIK to represent the simultaneous corrector Krylov method, and SDK to represent the staggered direct Krylov method.

The first example models a multi-species food web [4], in which mutual competition and/or predator-prey relationships in the spatial domain are simulated. Specifically, the model equations for the concentration vector $c = (c^1, c^2)^T$ are

$$\begin{aligned} c^1_t &= f_1(x, y, t, c) + d_1(c^1_{xx} + c^1_{yy}), \\ 0 &= f_2(x, y, t, c) + d_2(c^2_{xx} + c^2_{yy}), \end{aligned}$$

with

$$f_i(x, y, t, c) = c^i(b_i + \sum_{j=1}^{2} a_{ij} c^j).$$

The coefficients a_{ij}, b_i, d_i are

$$\begin{aligned} &a_{ii} = -1, \quad i = 1, 2; \\ &a_{12} = -0.5 \cdot 10^{-6}, \quad a_{21} = 10^4; \\ &a_{ij} = e, \quad i > p \text{ and } j < p; \\ &b_1 = 1 + \alpha xy + \beta \sin(4\pi x)\sin(4\pi y)) = -b_2, \\ &d_1 = 1, \quad d_2 = 0.05, \\ &\alpha = 50, \quad \beta = 100. \end{aligned}$$

The domain is the unit square $0 \le x, y \le 1$ and $0 \le t \le 10$. The boundary conditions are of Neumann form with normal derivative equal to 0. The PDEs are discretized by central differencing on an M by M mesh. We have taken M=20. Therefore the resulting DAE system has size NEQ = $2M^2$ = 800. The tolerances used were RTOL=ATOL=10^{-5}.

For sensitivity analysis, α and β were taken as the sensitivity parameters. The initial conditions were taken as

$$\begin{aligned} c^1 &= 10 + (16x(1-x)y(1-y))^2, \\ c^2 &= -(b_2 + a_{21}c^1)/a_{22}, \end{aligned}$$

which nearly satisfy the constraint equations. The initial conditions for the sensitivity variables were taken as zero, which are not consistent. We tested our problem with both the direct and Krylov methods. For the Krylov methods, we used the block-grouping preconditioner (which is included in the package DASPK2.0[4]). To eliminate the effect of finite differencing when comparing different methods, we used the ADIFOR option in DASPK3.0 to generate the Jacobian matrix (only for direct method) and sensitivity equations. Without setting INFO(11)=1, the integration failed. After setting INFO(11)=1, the consistent initial conditions were computed quickly for both the direct and Krylov methods. Table 1 shows the results of the staggered corrector method and the simultaneous corrector method. Full error control (including the sensitivity variables) was used. Although there were no convergence

METH	NSTP	NRES	NSE	NJAC	NNI	NLI	NLIS	NETF	CPU
STD	312	770	389	45	381	0	0	4	30.84
SID	335	508	508	42	508	0	0	3	36.56
STK	341	2712	353	36	406	732	0	1	22.98
SIK	505	4262	617	47	617	1532	0	9	39.12
STD	128	377	190	42	205	0	0	0	17.90
SID	128	228	228	40	228	0	0	0	18.91
STK	133	1456	147	38	165	329	425	0	11.36
SIK	131	1888	202	38	202	332	697	0	15.47
SDK	133	1442	133	38	165	329	425	0	11.03
STD	128	3589	190	42	187	0	0	0	24.85
SID	128	3240	228	40	228	0	0	0	26.11
STK	133	1442	147	38	165	329	425	0	10.36
SIK	131	1818	201	38	201	332	700	0	14.37
SDK	133	1442	133	38	165	329	425	0	10.12

Table 1: Results for multi-species food web. The upper part is for ADIFOR option with error control including the algebraic variables. The middle part is for ADIFOR option with error control excluding the algebraic variables. The bottom part is for the finite difference option with error control excluding the algebraic variables.

test failures for this problem, the staggered corrector method (ST) performed better than the simultaneous corrector method (SI).

The finite differencing options for the sensitivity equations were also tested. Because this problem is badly scaled (c^1 has a value of $O(10^6)$ while c^2 has a value of $O(10)$), finite differencing does not work well if we choose δ_i via (12). Selecting δ_i via (13), finite differencing works quite well. We used the central difference and $\Delta = 10^{-3}$ (default value). The results are shown in Table 1.

The next example is the heat equation,

$$\frac{\partial u}{\partial t} = p_1 u_{xx} + p_2 u_{yy},$$

posed on the 2-D unit square with zero Dirichlet boundary conditions. An $M+2$ by $M+2$ mesh is set on the square, with uniform spacing $1/(M+1)$. The spatial derivatives are represented by standard central finite difference approximations. At each interior point of the mesh, the discretized PDE becomes an ODE for the discrete value of u. At each point on the boundary, we pose the equation $u = 0$. The discrete values of u form a vector U, ordered first by x, then by y. The result is a DAE system $G(t, U, U') = 0$ of size $(M+2) \times (M+2)$. Initial conditions are posed as

$$u(t = 0) = 16x(1-x)y(1-y).$$

The problem was solved by DASPK on the time interval $[0, 10.24]$. We took M=40 in our test. To compute the sensitivities, we took 10 sensitivity parameters; p_1 and p_2 were two of them. The other 8 are chosen from the initial conditions. The error tolerances for DASPK are RTOL=ATOL=1.0D-4. For the direct method, we used the ADIFOR option with SparsLinC to generate the Jacobian. For the Krylov method, we used the incomplete LU (ILU) preconditioner, which is part of the DASPK package. The Jacobian for the ILU preconditioner is also evaluated by ADIFOR with SparsLinC. The sensitivity residuals are evaluated by ADIFOR with the seed matrix option. Table 2 gives the results of the staggered corrector and simultaneous corrector methods.

Because this problem is well-scaled, finite-differencing in the Jacobian and/or sensitivity equation evaluation gets a good result. Table 3 shows the results when central differencing (INFO(20)=0) is used for evaluation of the sensitivity equations. The default perturbation factor (10^{-3}) is used in evaluating the sensitivity equations. The Jacobian is also evaluated by finite-differencing. Only the data for full error control are listed.

We tested our parallel code on a cluster of DEC alpha machines in Los Alamos National Laboratory. Each processor is 533MHz with 64MB memory. We tested the heat equation with 24 sensitivity parameters. The staggered corrector method was used. The synchronization to achieve the same stepsize on each processor does not introduce much overhead to the computation.

The next example models a single pendulum

$$y_1' \;=\; y_3,$$

METH	NSTP	NRES	NSE	NJAC	NNI	NLI	NETF	CPU
STD	64	160	65	22	95	0	3	36.20
SID	64	97	97	22	97	0	3	46.35
STK	71	1527	72	18	100	149	1	25.67
SIK	71	1572	102	18	102	184	1	29.40
STD	92	220	103	23	123	0	2	53.58
SID	93	130	130	25	130	0	3	63.63
STK	106	1823	114	24	141	182	2	35.68
SIK	116	1776	155	24	155	213	2	39.06

Table 2: Results for heat equation with ADIFOR evaluation. The upper half is for partial error control (excluding the sensitivity variables). The bottom half is for full error control.

METH	NSTP	NRES	NSE	NJAC	NNI	NLI	NETF	CPU
STD	92	2118	103	23	117	0	2	64.07
SID	93	2175	130	25	130	0	3	75.76
STK	107	3917	114	24	143	187	2	38.39
SIK	116	3695	157	24	157	207	2	44.24

Table 3: Results for heat equation with finite difference approximation for sensitivities and full error control.

$$\begin{aligned} y_2' &= y_4, \\ y_3' &= -y_1 y_5, \\ y_4' &= -y_2 y_5 - g, \\ 0 &= y_1 y_3 + y_2 y_4, \end{aligned}$$

where $g = 1.0$. This is an index-two problem. The initial conditions are y_1=0.5, $y_2 = -\sqrt{p^2 - y_1^2}$, y_3=10.0, y_4=10.0, and $y_5 = 0.0$. The sensitivity parameter is p, which has initial value $p = 1.0$. The initial conditions for the sensitivity variables are (0.0, -1.1547, 0.0, 0.0, 0.0). We set all of the derivatives to 0 initially. The tolerance for DASPK was taken as RTOL=ATOL=10^{-6}. Because

$$g(y) = y_1 y_3 + y_2 y_4 = -3.660254 \neq 0,$$

the initial conditions are inconsistent. DASPK3.0 failed to solve the problem with the given initial conditions. The consistent initial conditions must be computed first. We solved it with the initialization option for index-2 problems. Since the system is linear with respect to

METH	NPROC	NSTP	NRES	NJAC	NNI	NLI	CPU
Direct	1	64	1810	19	91	0	35.33
	2						19.64
	4						12.50
	8						8.78
Krylov	1	71	3410	18	100	181	24.59
	2	71	1935	18	100	159	12.94
	4	71	1109	18	100	160	7.43
	8	71	696	18	100	156	4.75

Table 4: Results for heat equation with finite difference approximation and partial error control. MPI was used in all the parallel computations. The same stepsize control was enforced on all the processors.

the algebraic variable y_5, we use INFO(11)=4. During the computation, we monitored three constraints,

$$\begin{aligned} g_1 &= y_1^2 + y_2^2 - p, \\ g_2 &= y_1 y_3 + y_2 y_4, \\ g_3 &= y_3^2 + y_4^2 - (y_1^2 + y_2^2) y_5 - y_2. \end{aligned}$$

Initially, we have

$$g_1 = 0, \quad g_2 = -3.66, \quad g_3 = 200.866.$$

We also tried to fix y_1, y_2 during the experiments on the initial condition computation. The results are shown in Table 5. Note that if y_1 and y_2 are not fixed, g_1 may be violated. The

METH	Fixed	y_1	y_2	y_3	y_4	g_1	g_2	g_3
STD		0.512	-0.859	11.777	7.016	-1.88e-4	0.0	3.77e-15
y_5=0	y_1, y_2	0.5	-0.866	11.83	6.83	-1.1e-16	0.0	9.28e-13
y_5=10	y_1, y_2	0.5	-0.866	11.83	6.83	-1.1e-16	0.0	9.28e-13
SID		0.512	-0.859	11.777	7.016	-1.88e-4	0.0	3.77e-15
y_5=0	y_1, y_2	0.5	-0.866	11.83	6.83	-1.1e-16	0.0	4.73e-14
y_5=10	y_1, y_2				Failed			

Table 5: Results for consistent initial conditions for pendulum problem.

simultaneous method (SID) may fail with some initial values of y_5. This is because the SID method uses an approximate Jacobian (the block diagonal of the exact Jacobian) to solve for the state variables and sensitivities simultaneously. We also observed that the derivative input y' affected the output of the consistent initial conditions if no variables are fixed. If the consistent y' is input initially, the consistent initial conditions with no fixing are

$$(y_1, y_2, y_3, y_4) = (y_{10} + 0.0182, y_{20} + 4.13e^{-7}, y_{30} + 1.83013, y_{40} + 1.83013),$$

where $(y_{10}, y_{20}, y_{30}, y_{40})$ are the initial values. The constraints are

$$g_1 = 9.524e^{-7}, \quad g_2 = 0.0, \quad g_3 = 1.37e^{-8},$$

which are very close to the outputs when y_1 and y_2 are fixed. Thus if no variables are fixed, we strongly recommend using the consistent values of y'. We also calculated the initial conditions by solving optimization problem (18) with no fixing. The results are

$$(y_1, y_2, y_3, y_4) \simeq (0.6822, -0.6842, 10.0124, 9.9875). \tag{37}$$

Although (37) is much closer to the initial guess than the results of Table 5, it may not be better because the constraint $g_1 \simeq 0.0665$ is worse than $g_1 = 1.88e - 4$. The initialization might fail for a tight error tolerance. We have tested and found that it worked when RTOL=ATOL=10^{-6} but failed when RTOL=ATOL=10^{-7}. The initialization failed because of too many convergence failures.

References

[1] C. Bischof, A. Carle, G. Corliss, A. Griewank and P. Hovland, *ADIFOR–Generating derivative codes from Fortran programs*, Scientific Programming 1 (1992).

[2] K. E. Brenan, S. L. Campbell, and L. R. Petzold, Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations, Elsevier, New York, 1989 (second edition, SIAM 1996).

[3] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold, *Using Krylov methods in the solution of large-Scale differential- algebraic systems*, SIAM J. Sci. Comput., 15 (1994), pp. 1467-1488.

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

[5] M. Caracotsios and W. E. Stewart, *Sensitivity analysis of initial value problems with mixed ODEs and algebraic equations*, Computers and Chemical Engineering, 9:4 (1985), 359-365.

[6] N. Doss, W. Gropp, E. Luck, and A. Skjellum, *A model implementation of MPI*, Technical report, Argonne National Laboratory, 1993.

[7] W. F. Feehery, J. E. Tolsma and P. I. Barton, *Efficient sensitivity analysis of large-scale differential-algebraic systems*, Applied Numerical Mathematics 25 (1997), pp. 41-54

[8] P. E. Gill, W. Murray and M. A, Saunders, *SNOPT: An SQP algorithm for large-scale constrained optimization*, Numerical Analysis Report 96-2, Department of Mathematics, University of California, San Diego.

[9] W. Gropp, E. Luck, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT press, 1997.

[10] A. C. Hindmarsh, personal communication.

[11] S. Li, L. R. Petzold and W. Zhu, *Sensitivity analysis of differential-algebraic equations: A comparison of methods on a special problem*, submitted, 1999.

[12] T. Maly and L. R. Petzold, *Numerical methods and software for sensitivity analysis of differential-algebraic systems*, Applied Numerical Mathematics 20 (1996), pp. 57-79.

[13] L. R. Petzold, *A description of DASSL: A differential/algebraic system solver*, in Scientific Computing, R. S. Stepleman et al. (Eds.), North-Holland, Amsterdam, 1983, pp. 65-68.

[14] L. R. Petzold, J. B. Rosen, P. E. Gill, L. O. Jay and K. Park, *Numerical optimal control of parabolic PDEs using DASOPT*, In Large Scale Optimization with Applications, Part II: Optimal Design and Control, Eds. L. Biegler, T. Coleman, A. Conn and F. Santosa, IMA Volumes in Mathematics and its Applications, 93, 271-300.

[15] L. R. Petzold and W. Zhu, *Parallel sensitivity analysis for DAEs with many parameters,* submitted to Concurrency: Practice and Experience, 1998.

[16] Y. Saad and M. H. Schulz, *GMRES: A general minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comp. 7 (1986), 856-869.