SAND82-8637
Unlimited Release
Printed September 1982

A DESCRIPTION OF DASSL: A DIFFERENTIAL/ALGEBRAIC SYSTEM SOLVER

Linda R. Petzold
Applied Mathematics Division
Sandia National Laboratories, Livermore

ABSTRACT

This paper describes a new code DASSL, for the numerical solution of implicit systems of differential/algebraic equations. These equations are written in the form $F(t,y,y') = 0$, and they can include systems which are substantially more complex than standard form ODE systems $y' = f(t,y)$. Differential/algebraic equations occur in several diverse applications in the physical world. We outline the algorithms and strategies used in DASSL, and explain some of the features of the code. In addition, we outline briefly what needs to be done to solve a problem using DASSL.

Introduction

This paper describes a new code DASSL, for the numerical solution of implicit systems of differential/algebraic equations. These equations are written in the form

$$\begin{aligned} F(t, y, y') &= 0 \\ y(t_0) &= y_0 \\ y'(t_0) &= y_0'. \end{aligned} \tag{1}$$

where F, y, and y' are N dimensional vectors. DASSL is useful for solving two general classes of problems which cannot be handled by standard ODE solvers. For the first class, it is not possible to solve for y' explicitly to rewrite (1) as a standard form ODE system $y' = f(t, y)$. For the second class, it is possible in theory to solve for y', but it is impractical to do so. For example, to convert $Ay' = By$ to standard form, we must multiply by A^{-1}. If A is a sparse matrix, A^{-1} may not be sparse, so it is advantageous to be able to solve the equations in their original form.

Systems of differential/algebraic equations (DAE) arise in several diverse applications in the physical world. Problems of this type occur frequently in the numerical method-of-lines treatment of partial differential equations[1]. In these applications, $\partial F/\partial y'$ may be singular (so that it is not possible to solve explicitly for y') because of boundary conditions, or because the PDE system includes both evolutionary and non-evolutionary equations (such as the incompressible Navier-Stokes equations, or the boundary layer equations[2]). Differential/algebraic equations arise in the simulation of electronic circuits, where they are sometimes called semistate equations[3]. These systems also occur in the dynamic analysis of mechanical systems[4]. These problems can all be solved using DASSL.

Nearly all of the most popular codes for solving ordinary differential equations have been directed at systems written in standard form

$$y' = f(t, y), \quad y(t_0) = y_0. \tag{2}$$

Several codes besides DASSL have been written for solving systems which cannot be written in standard form. In the early 1970's, C. W. Gear[5] first noticed that numerical methods for solving stiff differential systems could be adapted to solve some DAE systems, and Gear and Brown[6] wrote a code for this purpose. In 1980, G. Soderlind[7] published a code for solving systems of the form

$$\begin{aligned} y' &= f(t, y, z), \quad y(t_0) = y_0 \\ 0 &= g(t, y, z), \end{aligned} \tag{3}$$

in which y and z are treated by different methods. A. C. Hindmarsh and J. F. Painter recently released a code LSODI[1] for solving linearly implicit DAE systems $A(t, y)y' = f(t, y)$. LSODI is similar to DASSL in that it uses backward differentiation formulas (BDF) to advance the solution from one time step to the next. However, there are substantial differences between these two codes, both in how they are used and in the strategies which are used internally to compute the solution. We will explore some of these differences later; for other details, the reader is referred to Petzold[8].

DASSL was developed because of a need at Sandia National Laboratories to solve problems of the form (1). The code has been used for solving problems arising from several different applications by users with varying backgrounds, on several computers. We have taken care to make the code easy to use, while at the same time providing options which are needed for flexibility in solving practical problems. The Fortran source code for DASSL can be obtained by writing to the author.

How The Code Works

In this section, we outline the algorithms which DASSL uses for advancing the solution from one time step to the next. A complete description of the algorithms and strategies used in the code can be found in Petzold[8].

The underlying idea of Gear[5] for solving DAE systems is to replace the derivative in (1) by a difference approximation, and then to solve the resulting equation for the solution at the current time t_n using Newton's method. For example, replacing the derivative by the backward difference in (1), we obtain the first order formula

$$F\left(t_n, y_n, \frac{y_n - y_{n-1}}{\Delta t_n}\right) = 0. \tag{4}$$

This equation is then solved using Newton's method,

$$y_n^{m+1} = y_n^m - \left(\frac{\partial F}{\partial y'} + \frac{1}{\Delta t_n}\frac{\partial F}{\partial y}\right)^{-1} F\left(t_n, y_n^m, \frac{y_n^m - y_{n-1}}{\Delta t_n}\right), \tag{5}$$

where m is the iteration index. The algorithms used in DASSL are an extension of this approach. Instead of using the first order formula (4), DASSL approximates the derivative using the k^{th} order backward differentiation formula (BDF), where k ranges from one to five. On every step it

chooses the order k and stepsize Δt_n, based on the behavior of the solution.

Newton's method(5) converges most rapidly when the initial guess y_n^0 is accurate. DASSL obtains an initial guess for y_n by evaluating the polynomial which interpolates the computed solution at the last $k+1$ times $t_{n-1}, t_{n-2}, \ldots, t_{n-(k+1)}$, at the current time t_n. An initial guess for y_n' is obtained by evaluating the derivative of this polynomial at t_n. Once y_n^0 is found, Newton's method is used to solve for y_n as in (5), except that in general the derivative is approximated by the k^{th} order BDF formula, instead of by the backward difference of y_n. When the stepsize is not constant, there is a choice as to which form of the BDF formula to use. DASSL uses the fixed leading coefficient form of the BDF formula (see Jackson and Sacks-Davis[9]). These formulas tend to be more stable than the fixed coefficient formulas used in LSODI[1], and are more efficient in some respects than the variable coefficient formulas used in EPISODE[10]. In DASSL, these polynomials are represented in terms of scaled divided differences; the details are discussed in Petzold[8].

It is important to solve the nonlinear equation (4) efficiently. To simplify notation, we can rewrite this equation as

$$F(t, y, \hat{\alpha} y + \beta) = 0, \tag{6}$$

where $\hat{\alpha}$ is a constant which changes whenever the stepsize or order changes, β is a vector which depends on the solution at past times and $t, y, \hat{\alpha}, \beta$ are evaluated at t_n. This equation is solved in DASSL by a modified version of Newton's method,

$$y^{m+1} = y^m - c\left(\frac{\partial F}{\partial y'} + \alpha \frac{\partial F}{\partial y}\right)^{-1} F(t, y^m, \hat{\alpha} y^m + \beta). \tag{7}$$

The iteration matrix $G = \partial F/\partial y' + \alpha \partial F/\partial y$ is computed and factored, and is then used for as many time steps as possible. In general, the value of α when G was last computed is different from the current value of $\hat{\alpha}$. If α is too different from $\hat{\alpha}$, then (7) may not converge. The constant c in (7) is chosen to speed up the convergence when $\alpha \neq \hat{\alpha}$, and is given by

$$c = \frac{2}{(1 + \hat{\alpha}/\alpha)}. \tag{8}$$

The rate of convergence ρ of (7) is estimated whenever two or more iterations have been taken by

$$\rho = \left(\frac{\|y^{m+1} - y^m\|}{\|y^1 - y^0\|}\right)^{1/m}. \tag{9}$$

(The norms are scaled norms which depend on the error tolerances specified by the user.) The iteration has converged when

$$\frac{\rho}{1 - \rho}\|y^{m+1} - y^m\| < 0.3. \tag{10}$$

If $\rho > 0.9$, or $m > 4$, and the iteration has not yet converged, then the stepsize is reduced, and/or an iteration matrix based on current approximations to y, y', and α is formed, and the step is attempted again.

The linear systems are solved using routines from the LINPACK[11] subroutine package. The matrix can either be dense or have a banded structure. For most problems, the iteration matrix is computed by finite differences. The j^{th} column of G is approximated by incrementing the j^{th} component of y in (6), and then forming the finite difference quotient. The choice of the increment is a delicate but important issue; for details, see Petzold[8]. When G is banded, it is computed using the algorithm of Curtis et. al.[12] so as to minimize the number of function evaluations required. There is an option available for the user to write a routine to compute G, given t, y, y' and α. For some problems, this can be more efficient than using finite differences to compute the matrix.

After the corrector iteration has converged, an error test is made to determine whether the solution satisfies a local error tolerance specified by the user. The test is satisfied whenever $C\|y_n - y_n^0\| \leq 1$, where C is a constant which depends on the order and recent stepsize history of the method. The constant C is chosen to control both the variable stepsize local truncation error, and the error in interpolated values of y between mesh points. If the error test is satisfied, the code takes another step. Otherwise, the stepsize and/or order are reduced and the step is attempted again.

The stepsize and order for the next step are determined using basically the same strategies as in Shampine and Gordon[13]. The code estimates what the error would have been if the last few steps had been taken at constant stepsize, at the current order k, and at $k-2, k-1$, and $k+1$. If these estimates increase as k increases, the order is lowered; if they decrease, it is raised. The new stepsize Δt_{n+1} is chosen so that the error estimate based on taking constant stepsizes Δt_{n+1} at order k_{n+1} satisfies the error test.

One of the main complications which arise in solving DAE's, which has no counterpart in ODE's, is that it may not be a trivial matter to obtain initial values for all of the components of y or y'. Depending upon the application, users may know y_0 but not y_0', y_0' but not y_0, or various combinations of these possibilities. In our experience, it is fairly common for users to know all of y_0, and some but not all of the components of y_0'. To facilitate solving problems of this type, there is an option in DASSL to compute the initial values for y', given the initial values of y and an initial guess for y_0'. The algorithm uses the backward Euler method (4), in conjunction with a damped Newton iteration. A stepsize Δt_0 is chosen based on considerations explained in Petzold[8], and the iteration matrix is computed at $y = y_0 + \Delta t_0 y_0', y' = y_0'$. This algorithm works best when the iteration matrix does not depend on y', or depends on this value only weakly. In contrast to the approach used in LSODI[1], it is applicable even if $\partial F/\partial y'$ is singular.

The code is arranged so that a driver routine, called DASSL, allocates storage, checks for illegal input and other error conditions, sets up the initial stepsize and optionally calls a subroutine to compute the initial derivative. DASSL calls the one-step solver DASTEP to advance the solution over each time step, and manages the output and error messages. Communication between DASSL and the other routines is via parameter lists and one labeled common block whose elements can only be altered by the driver. The common block contains pointers into work arrays, and these pointers can be changed only by the driver. Two routines NJAC and SOLVE manage the solution of the system of linear equations in DASTEP. NJAC computes the iteration matrix and factorizes it, and SOLVE calls the appropriate linear system solver to solve the decomposed system. Because the linear algebra is localized to these two routines, it is a simple matter to add new linear equation solvers for different types of matrices.

Using DASSL

DASSL is designed to be as easy to use as possible, while providing enough flexibility and control for solving a wide variety of problems. It is extensively documented in the source code. In this section we outline what needs to be done to solve a problem using this code.

The most important information the code needs is how to define the function F in (1), which describes the equation to be solved. To define F, the user writes a subroutine RES which takes as input the time T and the vectors Y and YPRIME, and produces as output the vector DELTA, where DELTA= F(T,Y,YPRIME) is the amount by which the function F fails to be zero for the input values of T,Y, and YPRIME.

To get started, DASSL needs a consistent set of initial values T, Y, and YPRIME. This means that we must have F(T,Y,YPRIME)=0 at the initial time. As we pointed out earlier, finding a consistent set of initial values for a given problem may not be trivial, and there is an option in DASSL to compute the initial value for YPRIME. We know of no procedures which apply in all situations for obtaining consistent initial conditions. Often, it is not difficult to find these values for specific classes of physical problems.

Additional information which must be supplied to the solver is virtually identical to that needed by ODE solvers[14], so we will not discuss it further here.

When DASSL is finished (either successfully or unsuccessfully), it returns to the user's calling program with a flag IDID which indicates what happened. If the flag is positive, the problem was solved successfully. Otherwise, something went wrong. DAE systems are in general quite a bit more complex than ODE systems (after all, (1) includes (2) as a special case), and the number of complications which can occur in solving them is correspondingly greater. The way that DASSL handles failures is explained in Petzold[8], and the code documentation gives information about the most likely cause of the problem in the event that a negative IDID is encountered.

If the DAE system does not have a well-defined solution, then the code is likely to have trouble. It is possible to write down problems of the form (1) which do not have any solutions, or which have solutions that are not unique. These problems cannot (fortunately!) be solved by the code, and will result in a negative IDID. Other problems may have solutions which are unique except at a single point. Still another type of system has a unique solution, but the problem is not well posed in the sense that a small (but discontinuous) perturbation to the equations causes an enormous change in the solution. All of these systems can cause problems for a code. Often, the system can be rewritten in a mathematically equivalent form so that it is solvable. This sometimes involves differentiating an algebraic constraint and/or eliminating a variable from the system by solving for it in terms of other variables and their derivatives. A complete description of the sources of these difficulties is beyond the scope of this paper. The interested reader is referred to Petzold[15] for a more detailed discussion of these types of systems. These complications will never occur for many practical problems. Anyone using this type of code should, however, think carefully about both his problem, and the formulation of his problem, before trying to solve it numerically.

References

1. A. C. Hindmarsh, "ODE Solvers for Use with the Method of Lines," *Advances in Computer Methods for Partial Differential Equations - IV*, R. Vichnevetsky and R. S. Stepleman, eds., IMACS, New Brunswick, NJ, 1981, pp. 312-316.
2. R. J. Kee and J. A. Miller, "A Computational Model for Chemically Reacting Flow in Boundary Layers, Shear Layers, and Ducts," Sandia National Laboratories report, SAND81-8241, August 1981.
3. R. W. Newcomb, "The Semistate Description of Nonlinear Time-Variable Circuits," *IEEE Trans. on Circuits and Systems*, CAS-28, No. 1, pp.62-71, (1981).
4. N. Orlandea, M. A. Chace, and D. A. Calahan, "A Sparsity-Oriented Approach to the Dynamic Analysis and Design of Mechanical Systems," *Journal of Engineering for Industry*, pp. 773-784, August 1981.
5. C. W. Gear, "Simultaneous Numerical Solution of Differential/Algebraic Equations," *IEEE Trans. on Circuit Theory*, CT-18, No. 1, pp. 89-95, (1971).
6. C. W. Gear and R. L. Brown, "Documentation for DFASUB–A Program for the Solution of Simultaneous Implicit Differential and Nonlinear Equations," University of Illinois, Dept. of Computer Science, UIUCDCS-R-73-575, July 1973.
7. G. Soderlind, "DASP3 - A Program for the Numerical Integration of Partitioned Stiff ODE's and Differential/Algebraic Systems," The Royal Institute of Technology, Stockholm, Sweden, Dept. of Numerical Analysis and Computing Science, TRITA-NA-8008.
8. L. R. Petzold, "DASSL: A Differential/Algebraic Systems Solver," Sandia National Laboratories, Livermore CA, in preparation.
9. K. R. Jackson and R. Sacks-Davis, "An Alternative Implementation of Variable Step-Size Multistep Formulas for Stiff ODEs," *ACM Trans. on Math. Software*, 6, No. 3, pp. 295-318, (1980).
10. A. C. Hindmarsh and G. D. Byrne, "A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations," *ACM Trans. on Math. Software*, 1, pp. 71-96, March 1975.
11. J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, "LINPACK Users' Guide," *SIAM*, Philadelphia (1979).
12. A. R. Curtis, M. J. D. Powell, and J. K. Reid, "On the Estimation of Sparse Jacobian Matrices," *J. Inst. Math. Applic.*, 15 (1974).
13. L. F. Shampine and M. K. Gordon, "Computer Solution of Ordinary Differential Equations," W. H. Freeman and Co., (1975).
14. L. F. Shampine and H. A. Watts, "DEPAC–Design of a User Oriented Package of ODE Solvers," Sandia National Laboratories report, SAND79-2374, (1980).
15. L. R. Petzold, "Differential/Algebraic Equations Are Not ODEs," to appear, *SIAM Journal on Scientific and Statistical Computing.*