Abstract

Over the last decade, the need to solve large stiff and nonstiff ordinary differential equation systems (initial value problems) has led to considerable software, mostly based on the methods of Adams (nonstiff case) and Gear (stiff case). The growing number of good general purpose initial value solvers has fueled discussions on the idea of a systematized collection of such solvers, called ODEPACK. In recent years, a tentative user interface standard was developed, and an initial collection of five solvers was written, primary at LLNL. These solvers handle stiff and nonstiff problems in standard (explicit) form, problems in linearly implicit form, full Jacobians, banded Jacobians, general sparse Jacobians, and problems with rootfinding requirements. They include solvers with automatic (stiff/nonstiff) method selection. These solvers are described briefly here, and their capabilities are illustrated with an example problem arising from a system of PDE's modeling atmospheric kinetics-transport.

1. Introduction

Initial value problems for systems of ordinary differential equations (ODE's) have long been a topic of great interest. Stiff systems are particularly prevalent and are, of course, much more challenging. Here stiffness can be roughly defined as the presence of one or more fast decay processes in time, having time constants that are short compared to the time span of interest.

Applications giving rise to large ODE systems vary widely. These include, to name only a few, discretized partial differential equation (PDE) systems, particle tracking problems, structural mechanics problems, and network analysis problems. Most of the ODE problems from these sources tend to be stiff as well as large.

Among the various numerical methods used for solving ODE initial value problems, a few are much more commonly used than others. The Adams multistep methods (explicit and implicit) are suitable for nonstiff systems, especially the implicit Adams methods. Explicit Runge-Kutta methods are also popular, but are also suitable only for nonstiff problems. Implicit Runge-Kutta methods of various types are being widely studied for use on stiff systems. But for large stiff problems, quite possibly the most popular methods used are based on the so-called backward differentiation formulas (BDF’s), which are multistep methods first implemented by C. W. Gear.

In 1968, Gear wrote a program called DIFSUB [1] that included the BDF method for stiff systems and implicit Adams methods for nonstiff systems. This program was reorganized, rewritten, and improved upon at LLNL, resulting in the GEAR package [2]. However, when solving a stiff system of size N, of the general form

\[ \frac{dy}{dt} = f(t, y) \]

this package makes use of the Jacobian matrix,

\[ J = \frac{\partial f}{\partial y} \]

in full N×N form. Because of this, variants of GEAR were developed later to handle large stiff problems having some sparse structure in the Jacobian. Among these were GEARB [3], for the case of a banded J; GEARS [4], for a general sparse matrix J; and GEARBI [5], for a regularly blocked J, with block-iterative (block-SOR) treatment of the associated linear systems. Another variant, GEARIB [6], was written for linearly implicit problems, i.e., problems of the form \( Ay' = g(t, y) \), in which the matrix A and \( \frac{\partial g}{\partial y} \) are banded.

As a frame of reference for later descriptions of algorithms and software, we give here a brief summary of the methods used in the GEAR package (and most of its variants and descendants). Consider the system \( y = f(t, y) \), and a discrete time
mesh \(t_0, t_1, \ldots, t_n, \ldots\). For the moment, we consider the step size \(h = t_n - t_{n-1}\) to be fixed. Discrete approximations \(y_n\) are to be constructed, with \(y_0\) given, and \(y_n\) will always denote \(f(t_n, y_n)\).

For nonstiff problems, we use the implicit Adams (or Adams-Moulton) formulas

\[
y_n = y_{n-1} + h \sum_{i=0}^{q-1} \beta_i y_{n-i}.
\]

Here \(q = 1 \leq q \leq 12\) is the order of accuracy, and the coefficients \(\beta_i\) depend only on \(q\). The formula is implicit in that \(\beta_i > 0\). Solution of this implicit equation is done by functional iteration,

\[
y_{n(m+1)} = y_{n-1} + h \beta_0 f(t_n, y_{n(m)}) + h \sum_{i=1}^{q-1} \beta_i y_{n-1},
\]

which is terminated by a convergence test. Both the step size \(h\) and order \(q\) are actually varied during the integration process, by use of estimates of the local errors committed, in relation to a user-supplied tolerance. Changes in \(h\) are achieved by interpolation of the multistep data. Note that no \(N\times N\) matrices are involved in this case.

For stiff problems, we use the BDF

\[
y_n = \sum_{i=1}^{3} a_i y_{n-i} + h \beta_0 y_n = a_n + h \beta_0 f(t_n, y_n),
\]

where again \(q\) is the order (here \(1 \leq q \leq 5\)), and \(\beta_0 > 0\). Stiffness makes functional iteration fail to converge for the step sizes of interest, because of strong dependencies in \(f\) upon \(y\). Thus we use a modified Newton iteration,

\[
P[y_{n(m+1)} - y_n] = y_n - a_n + h \beta_0 f(t_n, y_{n(m)}),
\]

where \(P\) is an \(N\times N\) matrix approximating the Jacobian of the algebraic system to be solved:

\[
P \equiv I - h \beta_0 J, \quad J = \partial f/\partial y.
\]

This differs from a true Newton method in that \(J\) is only evaluated periodically, and the same value of \(P\) (or its LU decomposition, if used) is used over all iterations in any one step, and also over several time steps. Again, \(h\) and \(q\) are both varied to meet local error tolerance requirements.

In applying the BDF method to large stiff problems, it is important to note that a numerical solution of the linear system

\[
Px = r (x = \text{correction vector}, \quad r = \text{residual vector})
\]

can very often easily take advantage of a sparse structure in \(P\). This is accomplished either through suitable structured LU decompositions, or through iterative linear system methods that exploit a given matrix structure.

The linearly implicit problem \(Ay = g\) arises frequently, one of the most common sources being the discretization of time-dependent PDE's by applying collocation, Galerkin, finite element, or other weighted residual methods to the spatial variables. In these problems, \(A\) is a square matrix, usually nonsingular. We allow \(A = A(t,y)\), but often \(A\) is constant. A numerical method for such an implicit ODE system can be obtained from either of the multistep formulas given above, by multiplying both sides by \(A(t_n,y_n)\), replacing \(A(t_n,y_n)\) by \(g(t_n,y_n)\), and solving the resulting implicit relation for \(y_n\). Again, a modified Newton iteration is usually most appropriate for this.

The following section describes a new collection of ODE solvers that has evolved from the GEAR package and its variants. Following a period of inter-laboratory discussions on user interfaces, most of the software development was done at LLNL. The last section presents an example problem, arising from a system of PDE's in atmospheric modeling, and illustrates the use of various solvers on it.

2. The ODEPACK Solvers

The GEAR package and its variants were added to a list of available general purpose initial value ODE solvers that was growing quite sizable by 1975. The length and diversity of this list caused some concern to users and software developers alike. There was much duplication of capabilities offered, but at the same time there was very little in common among the solvers in terms of either their external appearance or their internal structure. This situation was in sharp contrast to that in other areas in which "systematized collections" of Fortran routines were being developed. The earliest examples were EISPACK [7], for computing matrix eigensystems, LINPACK [8], for solving linear systems, and FUNPACK, for certain special functions.

2.1 The ODEPACK Concept

The idea of a systematized collection of initial value ODE solvers, tentatively called ODEPACK, was discussed informally as early as 1974, in workshops attended by people from all over the world [9]. However, it was quickly realized that the task was more difficult in the ODE case than in other areas, partly because of the complexity of the subject, and partly because of widely divergent views of what ODEPACK should look like. Starting in 1976, attempts were made to address this difficulty by involving only people at U.S. Department of Energy laboratories, and LLNL received funding to study the feasibility of ODEPACK from the Applied Mathematical Sciences Research Program under the Office of Basic Energy Sciences in DOE.

The natural first step, and a necessary preliminary to any actual development of an ODEPACK, is the setting of standards for the interface between the user and the ODE solvers. The user interface to a solver consists mainly of the call sequence of the routine the user must call, together with definitions of the one or more user-supplied routines called by the solver. To the extent that solvers for various problem types and using various methods must all communicate certain specific things to and from the user, it is possible to formulate a loose set of standards for the user interface. An early proposal is given in [10]. A sequence of workshops and discussions on user interface standards for ODE solvers succeeded in producing a reasonable consensus in 1978 [11,12]. The resulting tentative interface standard was achieved only through considerable compromise by the various participants, which included ODE software authors and users at various DOE laboratories.

At that time, it was agreed that several of the more popular ODE solvers, including GEAR, GEARB, DE/STEP [13] and RKF45 [14], would be rewritten to conform with the tentative standard interface, resulting in a small collection that was at least systematized in its external appearance. The first result of that agreement was a package based on the GEAR and GEARB packages, called LSODE (Livermore Solver for ODE's) [15].
subsequently (all in accordance with the tentative standard interface [11], with minor modifications) are briefly described in the following subsections. In the meantime, unfortunately, the other software authors involved withdrew from the agreement, and so this collection does not yet have analogous rewritten versions of their codes. However, Shampine and Watts (Sandia-Albuquerque) have developed an alternative interface design [16] with a somewhat narrower scope, and have rewritten DE/STEP and RKF45 accordingly. At present, therefore, there remains a lack of agreement among software authors on a standard user interface for ODE solvers.

2.2 LSODE: The Basic Solver

LSODE combines the capabilities of GEAR and GEARB. Thus it solves explicitly given stiff and nonstiff systems \( \dot{y} = f(t,y) \), and in the stiff case it treats the Jacobian as either full or banded, and as either user-supplied or internally approximated by difference quotients. By comparison with GEAR and GEARB, LSODE offers a number of new features that make it more convenient, more flexible, more portable and easier to install in software libraries. Some of these are the following:

(a) Through the redesigned user interface, many new options and capabilities are available, and others are much more convenient than before. Some examples are: more flexible error tolerance parameters, independent flags for starting and stopping options, internally computed initial step size, two work arrays in the call sequence for all internal dynamic work space, user names for \( f \) and \( J \) in the call sequence, easy changing of input parameters in mid-problem, another routine to add the matrix \( A \) to a different diagonal matrix (if used) is treated. The reason for requiring \( r \) to be computed in a subroutine called XERRWV. Two other small subroutines are user-callable for optional changing of the output unit number and optional suppression of messages. This trio of routines is compatible with a much larger error package (the SLATEC Error Handling Package) written elsewhere [17].

(g) LSODE easily allows a user to interrupt a problem and restart it later (e.g. in switching between two or more ODE problems). Also, using LSODE in overlay mode is very easy, with no loss of needed local variables.

(h) The user documentation, which is contained in the initial comment cards of the source, is given in a two-level form. A short and simple set of instructions, with a short example program, is given first, for the casual user. Then detailed instructions are given for users with special problem features or a desire for nonstandard options. The latter is also organized so as to allow selective reading by a user who wants only a fraction of the nonstandard capabilities.

(c) When stiff options are selected, linear systems are solved with routines from LINPACK [8], which is becoming a widely accepted standard collection of linear system solvers.

(d) Some retuning of various heuristics was done so that performance should be more reliable than for GEAR/GEARB.

(e) The core routine which takes a single step, called STODE, is independent of the way in which the Jacobian matrix (if used) is treated. Thus as variant versions of LSODE are written for other matrix structures (such as LSODES), these will share the same subroutine STODE.

(f) The writing of all error messages is done in a small isolated general-purpose message handler called XERRWV. Two other small subroutines are user-callable for optional changing of the output unit number and optional suppression of messages. This trio of routines is compatible with a much larger error package (the SLATEC Error Handling Package) written elsewhere [17].

(g) LSODE easily allows a user to interrupt a problem and restart it later (e.g. in switching between two or more ODE problems). Also, using LSODE in overlay mode is very easy, with no loss of needed local variables.

(h) The various lists of constants needed for the integration, formerly appearing in a subroutine called COSET, are now computed (once per problem). This adds to the portability of LSODE.

2.3 LSODI: Implicit Systems

The LSODI solver [15], written jointly with J. E. Painter (LLNL.), treats systems in the linearly implicit form \( A(t,y)y = g(t,y) \), where \( A \) is a square matrix. Many problems, including PDE’s treated by finite elements and the like, result in such systems, and it is almost always more economical to treat the system in the given form than to convert it to an explicit form \( \dot{y} = f \). LSODI allows A to be singular, but the user must then input consistent initial values of both \( y \) and \( \dot{y} \).

In the singular case, the system is a differential-algebraic system, and then the user must be much more cautious about formulating a well-posed problem, as well as in using LSODI, which was not designed to be robust in this case. LSODI is based on (and supersedes) the GEARB package, but corrects a number of deficiencies, as follows:

(a) The matrices involved can be treated as either full or banded, by use of the method flag.

(b) The dependence of \( A \) on \( y \) is automatically and inexpensively accounted for, whether partial derivatives are supplied by the user or computed internally by difference quotients.

(c) When \( A \) is singular, the user needs to supply only the initial value of \( dy/dt \), and this array (along with the initial \( y \)) is passed through the call sequence, rather than computed in a user-replaceable package routine. (Admittedly, correct initial data can be difficult to obtain for some types of problems.) When the initial \( dy/dt \) is not being supplied, an input flag instructs LSODI to compute it on the assumption that \( A \) is initially nonsingular. Thereafter, no such assumption is made, but ill-conditioning can be a problem when \( A \) is singular.

(d) The user-supplied residual routine includes a flag which allows the user to signal either an error condition or an interrupt condition.

(e) To the maximum extent possible, LSODI shares the same user interface as LSODE, and so reflects all the advantages over GEARB that LSODE has over GEAR and GEARB.

The differences between the LSODI and LSODE user interfaces occur primarily in the user-supplied subroutines. With LSODI, one must supply a routine to compute the residual function \( r = g(t,y) - A(t,y)s \) for a given \( t, y, \) and \( s \), and another routine to add the matrix \( A \) to a given array. Optionally, the user can supply a routine to compute the Jacobian matrix \( A(t,y) \). The reason for requiring \( r \) from the user, as opposed to \( g \) and \( A \) separately, is that this allows the user to evaluate \( r \) in a manner that may be more economical (in both computation time and storage) than what would otherwise be done. In PDE applications, the resulting economies have been very great.
By virtue of the modular and systematized organization of LSODE and LSODI, the two packages share most of their routines with each other.

Some examples of the use of LSODE and LSODI on systems arising from PDE problems can be found in [18] and [19]. In the latter, experiments by Painter on incompressible Navier-Stokes problems shed some light on the difficulties involved with differential-algebraic systems.

2.4 LSODES: General Sparse Jacobian

The LSODES package solves explicit systems \( y = f \) but treats the Jacobian matrix \( J \) as a general sparse matrix in the stiff case. LSODES was written jointly with A. H. Sherman (Exxon Production Research Co.), and supersedes a sparse Matrix Package [4, 20]. In LSODES, the linear systems are solved using parts of the Yale Sparse Matrix Package (YSMP) [21, 22]. This involves several phases:

(a) Determination of sparsity structure. This is either inferred from calls to the \( f \) routine, inferred from calls to a \( J \) routine (if one is supplied), or supplied directly by the user. A user input flag determines which is done.

(b) Determination of pivot order. Diagonal pivot locations are chosen, and the choice is based on maximizing sparsity. This is done by YSMP.

(c) Symbolic LU factorization of the matrix \( J \). This is based only on sparsity and the pivot order, and uses the module in YSMP designed for non-symmetric matrices with compressed pointer storage.

(d) Construction of \( J \). This can be done internally by difference quotients, or with a user-supplied routine. In the difference quotient case, the number of \( f \) evaluations needed is kept to a minimum by a column grouping technique due to Curtis, Powell, and Reid [23]. In the other case, the user-supplied routine provides one column of \( J \) at a time, in the form of a vector of length \( N \) (although only non-zero elements need be computed and stored), so that users need never deal with the internal data structure for \( J \) and \( P \). In any case, \( J \) is stored internally in an appropriate packed form. Evaluations of \( J \) are done only occasionally, as explained below.

(e) Construction of \( P = I - h_\beta_0 J \). In contrast to GEARS, LSODES does not force a re-evaluation of \( J \) whenever the existing \( P \) is deemed unsuitable for the corrector iterations. Instead, when the value of \( J \) contained in the stored value of \( P \) is likely to be usable (and \( P \) is not, only because \( h_\beta_0 \) has changed significantly), then a new matrix \( P \) is constructed from the old one, with careful attention to roundoff error. This cuts down greatly on the number of \( J \) evaluations necessary.

(f) Numerical LU factorization of \( P \). This is done by YSMP in sparse form, and the array containing \( P \) is saved in the process (this allows for updating \( P \) as described above). Because of the absence of partial pivoting for numerical stability, this operation can conceivably fail. However, this has only rarely been observed in practice, and if it does occur (with a current value of \( J \)), the step size \( h \) gets reduced and the problem disappears.

(g) Solution of \( Px = r \). This is done by YSMP using the existing sparse factorization of \( P \). Because a modified Newton iteration is used, many values of \( r \) (i.e., many linear systems) can arise for the same \( P \), and the separation of the various phases takes advantage of the fact.

The first three phases, and part of the fourth (column grouping for difference quotients), are normally done only at the start of the problem. However, the user can specify that the sparsity structure is to be redetermined in the middle of the problem, and then these operations are repeated.

Actually, the matrix operated on by YSMP is \( A = P^T \), not \( P \), because \( P \) is generated in column order while YSMP requires the matrix to be described and stored in row order. This causes no difficulty, however, because YSMP includes a routine for solving the transpose problem \( x^TA = r^T \) (which is equivalent to \( Px = r \)) as well as for the direct problem \( Ax = b \).

A package called LSODIS, similar to LSODI (for the \( A\dot{y} = g \) problem) but using YSMP for general sparse treatment of matrices as in LSODES, is in the development process, and will soon be available.

2.5 LSODA: Automatic Method Selection

LSODA is a variant of LSODE of yet another kind. It was written jointly with L. R. Petzold (Sandia-Livermore), and switches automatically between nonstiff (Adams) and stiff (BDF) methods, by an algorithm developed by Petzold [24]. (The suffix A is for Automatic.) Thus it is more convenient than LSODE for users who would rather not be bothered with the issue of stiffness. Also, it is potentially more efficient than LSODE (when used with a fixed method option), when the nature of the problem changes between stiff and nonstiff in the course of the solution. In place of the method flag parameter of LSODE, the user of LSODA supplies only a Jacobian type flag. The storage space supplied to the solver can be either static (and thus allow for either the stiff or nonstiff method), or dynamic (and altered each time there is a method switch, to an amount specified by the solver).

2.6 LSODAR: Rootfinding

LSODAR combines the capabilities of LSODA with a rootfinder. It allows one to find the roots of a set of functions \( g_i(t,y) \) of the independent and dependent variables in the ODE system. Thus, for example, it could be used in a particle tracking problem to determine when a particle path reaches any of the walls of a container. LSODAR was also written jointly with L. R. Petzold, based on work by K. Hiebert and L. F. Shampine [25] (Sandia-Albuquerque). The user must supply, in addition to the LSODA inputs, a subroutine that computes a vector-valued function \( g(t,y) = (g_i) (i=1,2,\ldots,NG) \) such that a root of any of the \( NG \) functions \( g_i \) is desired. Of course there may be several roots in a given interval, and LSODAR returns them one at a time, in the order in which they occur along the solution, with an integer array to tell the user which \( g_i \) (if any) were found to have a root on a given return.

3. An Example Problem

In order to illustrate the various solvers described above, and to demonstrate their relative merits on a realistic problem, we consider here an example problem. The problem is a simple atmospheric model with two chemical species undergoing diurnal kinetics and transport in two space dimensions. The independent variables in the PDE system are horizontal position \( x \), altitude \( z \) (both in kilometers), and time \( t \) (in see), with \( 0 \leq x \leq 20, 30 \leq z \leq 50, 0 \leq t \leq 86400 \) (1 day). The dependent variables are \( c^1(x,z,t) = \) the concentration of the oxygen singlet \([O] \), and \( c^2(x,z,t) = \) that of ozone \([O_3] \) (both in
mole/cm³). The concentration of molecular oxygen [O₂] is assumed constant. The equations of the model are:

\[ c_t = (K_v(z)c_x^i) + K_h c_x^i + R^i(c^1, c^2, t) \]
\[ + R^j(c^1, c^2, t) \quad (i = 1, 2), \]

where \( R^1 \) and \( R^2 \) represent the chemistry and are given by

\[ R^1(c^1, c^2, t) = -(k_1 + k_2 c^2)c^1 + k_3(t)c^2 + k_4(t) \cdot 7.4 \cdot 10^{16} \]
\[ R^2(c^1, c^2, t) = (k_1 - k_2 c^2)c^1 - k_3(t)c^2. \]

Subscripts \( t, z, \) and \( x \) denote partial derivatives. The various coefficients are as follows:

\[ K_v(z) = 10^{-8} \cdot \exp(x/5), \quad K_h = 4 \cdot 10^{-6}, \quad k_1 = 6.03, \quad k_2 = 4.66 \cdot 10^{-16} \]

\[ k_3(t) = \begin{cases} 
\exp[-7.601/\sin(\pi t/43200)] & \text{for } t < 43200 \\
0 & \text{for } t \geq 43200
\end{cases} \]

\[ k_4(t) = \begin{cases} 
\exp[-22.62/\sin(\pi t/43200)] & \text{for } t < 43200 \\
0 & \text{for } t \geq 43200
\end{cases} \]

The initial conditions are

\[ c^i(x, z, 0) = 10^{6i} \left[ 1 - \left(\frac{x-10}{10}\right)^2 + \frac{(x-10)^4}{2} \right] \]
\[ \left[ 1 - \left(\frac{z-40}{10}\right)^2 + \frac{(z-40)^4}{2} \right] \quad (i = 1, 2), \]

and both \( c^1 \) and \( c^2 \) are required to satisfy homogeneous Neumann boundary conditions along all the \( x \) and \( z \) boundaries.

To solve the above system numerically, we apply the method of lines using a regular rectangular mesh with constant mesh spacings

\[ \Delta x = 20/(M_x - 1), \]
\[ \Delta z = 20/(M_z - 1). \]

The spatial derivatives are approximated by standard 5-point central differences, and the boundary conditions are similarly replaced by difference relations. The resulting ODE system \( y' = f(t, y) \) has size \( N = 2M_x M_z \). It is quite stiff because of the presence of short kinetics time constants.

The initial value vector \( y_0 \) is taken from the initial condition functions given above. The system Jacobian \( J \) is sparse, with roughly 12M,M \( = 6N \) nonzero elements. As a band matrix, with component ordering first by species, then by \( x \), and lastly by \( z \), it has a half-bandwidth of 2M., and thus a full bandwidth of 4M.,+1.

As a nominal case, consider the choice \( M_x = M_z = 10 \). As to accuracy, a crude model of this type calls for no more than a few significant figures. To be conservative in recognizing that tolerance parameters are applied to local errors, which can accumulate into global error, we might impose a local relative tolerance of \( 10^{-4} \). We must also specify a positive absolute tolerance on the values of \( c^1 \) because it decays to negligible values at night. A reasonable absolute tolerance is \( 10^{-2} \). With the ODEPACK solvers, specifying such a mixed relative/absolute error control is trivial, but with the GEAR family, a slight modification to the driver is necessary.

Of the various solvers mentioned, four are suitable for this particular problem—LSODE, LSODA, LSODES, and GEARBI. Recall that LSODES uses a general sparse treatment of the Jacobian matrix. GEARBI uses block-SOR, while LSODE and LSODA (in this case) treat the Jacobian as banded. The problem was set up for each of these four solvers and run on a CDC-7600 computer. For all but GEARBI, both the user-supplied Jacobian option and the internal difference quotient Jacobian option were tested. (For GEARBI, there is no difference quotient option.) The results of the various runs are given in Table 1. The tabulated quantities are:

- R.T. = CPU run time in sec
- NST = number of steps
- NFE = number of f evaluations
- NJE = number of J evaluations
- NLU = number of LU decompositions
- W.S. = total size of work space arrays

In the table, the notation USJ denotes the user-supplied Jacobian option, and DQJ denotes the internal difference quotient Jacobian option.

For the sake of illustration, the GEARBI test was repeated on a 38×38 grid, and the results given in the last line of Table 1. This is the largest square grid that could be accommodated with that solver on the CDC-7600 within its Small Core Memory (about 57,000 words). For the actual atmospheric models on which this problem is based, the Large Core Memory (about 400,000 words) was used. The Cray-1 computer will accept even larger problem sizes.

Several points of interest can be noted in the table, for the 10×10 problem. First, the number of steps does not vary greatly from solver to solver, because that is determined almost entirely by the accuracy requirement, and the accuracy is much the same for all these runs. The relative merits of the solvers must be judged from other statistics.

The performance characteristics of LSODE and LSODA are similar, as expected, since both use a banded Jacobian. The superiority of LSODA over LSODE here is primarily due to the use (by
of the nonstiff method on the initial transient of the problem. The use of a difference quotient Jacobian is invariably more expensive here, owing to its cost of 41 evaluations of f for each evaluation of J.

The LSODEs results show that a general sparse matrix treatment gives a significant speedup over the band treatment. This results primarily from the algorithm of effectively saving old values of J for greater reuse. Note that each computed value of J is used for 36 to 46 steps, as opposed to only 5 to 6 steps with the solvers using a banded Jacobian. Also, the cost penalty for a difference quotient Jacobian is much smaller with LSODEs, because each J evaluation here costs only 8 f evaluations. The storage requirement is only slightly smaller, reflecting the need for sparsity information arrays and the fact the Newton matrix P is not overwritten with its LU decomposition, as it is in the band case.

The best performance on this problem, however, is that of GEARBI. This should not be a surprise, since the Jacobian has a very regular block structure of which the block-SOR method in GEARBI is taking full advantage, both in storage and computation. The LU decompositions here are only those of the block-diagonal part of the Newton matrix (with 2x2 blocks). The total number of block-SOR iterations for the 10x10 grid was 607, or an average of less than 2 per step. For the 38x38 grid this cost rose to 2122 iterations, or an average of 5.4 per step. The latter run also shows that spatial discretization errors in the 10x10 grid answers are as large as 2%. For an earlier comparison test on this problem, see [20]. A variant of LSODE which embodies the methods in GEARBI will be added to the ODEPACK collection at some future time.

In closing, we mention some truly large problems to which the GEARBI package has been applied. In the early 1970's, a number of atmospheric models were developed at LLNL, involving chemical kinetics and transport in up to 2 space dimensions. Typically, the number of chemical species was 5 to 20, and typical 2-D mesh sizes were about 40 by 40. Thus when finite differences, these problems generated ODE systems of sizes exceeding 10,000. The smallest kinetics time constants were typically in the range of milliseconds to microseconds, while the largest diffusion time constants were measured in years, making these systems extremely stiff. The GEARBI package, and an extension of it using Large Core Memory (CDC-7600), were successfully used to solve these problems [26,27,28].

References
Abstract

MATRIX is designed as a software system to perform all the steps in the control design cycle starting from system modeling to data analysis, identification, control synthesis and simulation. The package features a powerful matrix interpreter, a user-friendly environment with device independent graphics, state-of-the-art numerical algorithms for reliable computations and user-transparent file management. The program is implemented in ANSI-77 FORTRAN, and is designed to run on any allowing interactive execution of FORTRAN programs.

Introduction

Computer-aided-design tools have a significant role to play in the future of control design practice. Each step of the control design cycle starting from system modeling to data analysis, system identification, model reduction, control design, simulation and implementation can be made more efficient with software tools. MATRIX offers classical as well as modern approaches to control design. MATRIX provides a comprehensive set of capabilities in a single integrated package with uniform data and file formats. Most "bookkeeping" chores are performed by the software, leaving the control designer free to tackle control problems. Graphics allow the designer to rapidly visualize information. The system-build capability lets the designer see the schematic block diagram as it is built.

MATRIX is built on reliable numerical algorithms drawn from LINPACK, EISPACK and recent research in numerical analysis. Numerical stability and robustness are always important, but particularly for system dimensions exceeding approximately 20. The designer can handle systems with large dimensions in MATRIX, relying on superior numerical software with comprehensive reporting and control of numerical errors.

Perhaps the most remarkable aspect of MATRIX is its simplicity. Commands are natural, simple, yet powerful. Online documentation with HELP commands, extensive, direct, jargon-free diagnostics and reasonable defaults make MATRIX easy to use. The interpreter can execute higher level constructs called MACROS and command files. They allow personalization of MATRIX commands, eliminating superfluous interactions with the designer. The command files are typically less than ten lines long and consist of interpreter statements. Creation and modification of specific design and analysis procedures are easy because of the rich set of primitives and the hierarchical structure.

MATRIX inherited many of its capabilities from its predecessor MATLAB [6], developed by Cleve Moler, which was developed as a pedagogical aid for numerical analysis. In contrast, MATRIX was designed to serve as an engineering design tool. The code-length of MATRIX (although code-length is not necessarily a measure of capability) is five times that of MATLAB. MATRIX is significantly more efficient in memory usage and computations. Capabilities to perform control synthesis, system identification, signal processing and simulation required a number of new numerical algorithms. MATRIX also incorporates a state-of-the-art device independent graphics package which conforms to the 1979 core system defined by Graphics Standard Planning Committee of ACM/SIGGRAPH.