The MATLAB ODE Suite

Lawrence Shampine, Mark Reichelt

THE MATLAB ODE SUITE

LAWRENCE F. SHAMPINE* AND MARK W. REICHELT†

Abstract. This paper describes mathematical and software developments for a suite of programs for solving ordinary differential equations in MATLAB.

Key words. ordinary differential equations, stiff, BDF, Gear method, Rosenbrock method, non-stiff, Runge-Kutta method, Adams method

AMS subject classifications. 65L06, 65L05, 65Y99, 34A65

1. Introduction. This paper presents mathematical and software developments that are the basis for a suite of programs for the solution of initial value problems of the form

$$y' = F(t, y)$$

on a time interval $t_0 \le t \le t_f$, given initial values $y(t_0) = y_0$. The solvers for stiff problems allow the more general form

$$M(t)\, y' = f(t, y)$$

with a mass matrix $M(t)$ that is non-singular and (usually) sparse. The programs have been developed for MATLAB [29], a widely used environment for scientific computing. The environment imposed constraints, briefly described in Section 2, on the choice of methods and on how the methods were implemented. On the other hand, it has features, some of which are available in C and FORTRAN 90, that make possible an interesting and powerful user interface.

We begin in Section 3 by developing a new family of formulas for the solution of stiff problems called the numerical differentiation formulas, NDF's. They are substantially more efficient than the backward differentiation formulas, BDF's, and of about the same applicability. These formulas are conveniently implemented in a backward difference representation. Perhaps the least convenient aspect of using backward differences is changing step size. A new way of accomplishing this is developed that is both compact and efficient in MATLAB.

In Section 4 we devise a new linearly implicit one-step method for solving stiff systems. This modified Rosenbrock method overcomes a serious defect in a method of Wolfbrandt [40] and the efficient error estimating companion of Zedan [42]. An interpolant is derived that is "free" and of better quality than one previously used for Wolfbrandt's method.

Section 5 describes briefly how to modify the computational schemes developed for the NDF's and the modified Rosenbrock method so as to solve efficiently problems posed conveniently in terms of a mass matrix.

In Section 6, we discuss briefly three explicit methods for non-stiff problems and their implementations in the MATLAB environment. The two codes based on Runge-Kutta methods replace those of the previous generation of solvers. They are based on improved formulas that have an interpolation capability. The interface remedies deficiencies in the original design and exploits interpolation to achieve smooth plots.

* Math. Dept., Southern Methodist Univ., Dallas, TX 75275. (na.shampine@na-net.ornl.gov)
† The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760. (mwr@mathworks.com)

The third code is a variable order implementation of Adams-Bashforth-Moulton PECE formulas.

In the context of MATLAB the software interface is of primary importance. Section 7 explains how the language was exploited to devise an interface that is unobtrusive, powerful, and extendable. Indeed, in the course of developing the suite, the interface was extended in a remarkably convenient way to add to the codes for stiff problems the capability of working with sparse Jacobians. Later the interface was extended again to allow a more general form of the differential equations for stiff problems.

Section 8 describes an exceptionally strong scheme for the numerical approximation of the partial derivatives required by the programs for solving stiff problems. The new suite provides for general sparse Jacobians.

In Section 9, a selection of numerical examples illustrates the methods and their realizations. Section 10 gives conclusions and acknowledgements, and describes the availability of the codes.

2. Design Constraints. Certain aspects of the environment and the MATLAB language affected profoundly the selection of numerical methods and their implementation. Here we state briefly important constraints on the design of the suite, contrasting them implicitly with what might be typical of scientific computation in batch mode.

Although it is by no means always the case, the typical problem in the MATLAB environment is solved interactively and the results displayed graphically. The former argues for an interface that places minimal demands on the user. The fact that graphical accuracy of, say, 0.1% is common influences greatly the choice of numerical method. To obtain smooth plots efficiently and for other purposes, it is necessary that the integrators have an interpolation capability.

As a rule the functions defining the differential equations are not particularly expensive to evaluate. The language itself and interactive computation combine with this to make overhead a comparatively important part of the overall cost. This is to be addressed in two ways: careful exploitation of the capabilities of the language to minimize the effects of overhead, especially the fact that array operations are fast, and a careful weighing of algorithmic devices to decide whether potential benefits are worth the overhead.

Some issues are particularly important for the solution of stiff problems. The typical problem is either of modest size or has a Jacobian that is highly structured. MATLAB handles storage dynamically and retains copies of arrays. The language provides for sparse arrays and linear algebra is fast.

3. Implicit Formulas for Stiff Systems. Unquestionably the most popular methods for the solution of stiff problems are the backward differentiation formulas, BDF's. The BDF's have a particularly simple form when the step size is a constant h and backward differences are used. In a step from (t_n, y_n) to (t_{n+1}, y_{n+1}), where $t_{n+1} = t_n + h$, the backward differentiation formula of order k, BDFk, is

$$\sum_{m=1}^{k} \frac{1}{m} \nabla^m y_{n+1} - hF(t_{n+1}, y_{n+1}) = 0 \tag{1}$$

This implicit equation for y_{n+1} is typically solved with a simplified Newton (chord) method. The iteration is started with a value predicted by the simple formula

$$y_{n+1}^{(0)} = \sum_{m=0}^{k} \nabla^m y_n \tag{2}$$

The leading term in the truncation error of BDFk is

$$\frac{1}{k+1} h^{k+1} y^{(k+1)} \tag{3}$$

which is conveniently approximated by

$$\frac{1}{k+1} \nabla^{k+1} y_{n+1} \tag{4}$$

General-purpose BDF codes adapt the step size to the solution of the problem. For practical reasons the typical implementation of the BDF's is quasi-constant step size, meaning that the formulas used are always those for a constant step size h and the step size is held constant to the extent possible. When it is necessary or desirable to change the step size to a new step size h_{new}, previous solution values computed at a spacing of h are interpolated to obtain values at a spacing of h_{new}.

In addition to varying the step size, general-purpose BDF codes also vary the order of the formula. Unfortunately, the stability of the BDF's worsens as the order is increased. Although the first and second order BDF methods are A-stable (and L-stable), the formulas are not even zero-stable for orders higher than 6. In fact, the stability of the formula of order 6, BDF6, is so poor that it is not used in the popular codes.

In this section we obtain a family of formulas called the NDF's that are closely related to the BDF's and offer some advantages. A backward difference representation is particularly convenient for these formulas, so we develop a way of changing the step size when using this representation that is well-suited to MATLAB. Finally, some of the important details of implementing the NDF's in the program `ode15s` are discussed.

3.1. The Numerical Differentiation Formulas. To address the issue of the poor stability of the higher order BDF's, Klopfenstein [25] and Reiher [31] considered a related family of formulas. Noting that the standard predictor equation (2) involves one more value from the past than is used by the BDFk method (1), they considered how this extra value might be used to improve the stability of BDFk. Klopfenstein studied methods of the form

$$\sum_{m=1}^{k} \frac{1}{m} \nabla^m y_{n+1} - hF(t_{n+1}, y_{n+1}) - \kappa\gamma_k \left(y_{n+1} - y_{n+1}^{(0)} \right) = 0 \tag{5}$$

that he called numerical differentiation formulas, NDF's. Here κ is a scalar parameter and the coefficients γ_k are given by

$$\gamma_k = \sum_{j=1}^{k} \frac{1}{j}$$

When $\kappa = 0$, the NDF's reduce to the BDF's. Reiher [31] investigated the same class of formulas, but wrote them in Lagrangian form.

The role of the term that Klopfenstein added to BDFk is illuminated by the identity

$$y_{n+1} - y_{n+1}^{(0)} = \nabla^{k+1} y_{n+1}$$

and the approximation (4) to the truncation error of BDFk. It follows easily that for any value of the parameter κ, the method is of order (at least) k and the leading term in the truncation error is

$$\left(\kappa\gamma_k + \frac{1}{k+1}\right) h^{k+1} y^{(k+1)} \tag{6}$$

For NDF's of orders 3-6, Klopfenstein and Reiher computed parameters κ that maximize the angle of A(α)-stability, yielding formulas more stable than the corresponding BDFk. The situation at order 2 was special. Because BDF2 is already A-stable, Klopfenstein considered how to choose κ so as to reduce the truncation error as much as possible whilst still retaining A-stability. He proved a theorem to the effect that the optimal choice is $\kappa = -1/9$, yielding a truncation error coefficient half that of BDF2. This implies that while maintaining L-stability, NDF2 can achieve the same accuracy as BDF2 with a step size about 26% bigger, a substantial improvement in efficiency.

A few details about efficiency will be useful in what follows. When selecting a step size, a code tries to choose the largest h for which the leading term in the truncation error is no bigger than a given tolerance. For BDFk, the term is given by (3) and for NDFk, by (6). Both have the form $Ch^{k+1}y^{(k+1)}$. A smaller truncation error coefficient C allows a given accuracy to be achieved with a larger step size h and the interval of integration to be traversed in fewer steps. Because steps taken with BDFk and NDFk cost the same, the integration is more efficient with the formula having the smaller C. More specifically, for a given error tolerance, if BDFk uses a step size of h, then NDFk can use a step of size rh, where

$$r = \left[\frac{\left(\frac{1}{k+1}\right)}{\left(\kappa\gamma_k + \frac{1}{k+1}\right)}\right]^{\frac{1}{k+1}} \tag{7}$$

For NDF2 with $\kappa = -1/9$, this gives $r = 1.26$ and we say that NDF2 is 26% more efficient than BDF2.

Unfortunately, the formulas derived by Klopfenstein and Reiher at orders higher than 2 are less successful. The price of improved stability is reduced efficiency. In our estimation, the improvements are not worth the cost. If they had found a formula of order 6 that is sufficiently stable to be used, then it would not matter that it is less efficient than BDF6. But the angle of A(α)-stability for their NDF6 is improved from 18° to only 29°, an angle that is still too small for us to consider using the formula.

Taking a tack opposite to Klopfenstein and Reiher, we computed values of κ that make the numerical differentiation formulas more accurate (hence more efficient) than the BDF's and not much less stable. Because Klopfenstein's NDF2 with $\kappa = -1/9$ optimally improves accuracy while retaining L-stability, we adopted this as the order 2 method of our NDF family. For orders 3-5, our goal was to obtain an improvement in efficiency similar to that of NDF2 (26%) with the proviso that we were not willing

TABLE 1
The Klopfenstein-Shampine NDF's and their efficiency and A(α)-stability relative to the BDF's.

order	NDF coeff	step ratio	stability angle		percent
k	κ	percent	BDF	NDF	change
1	-0.1850	26%	90°	90°	0%
2	-1/9	26%	90°	90°	0%
3	-0.0823	26%	86°	80°	-7%
4	-0.0415	12%	73°	66°	-10%
5	0	0%	51°	51°	0%

to accept a stability angle less than 90% of that of the corresponding BDFk. Using MATLAB, an M-file of a few lines provides for the computation of the stability angle and plotting of the stability region by the root-locus method.

Using equation 7 and the root-locus method, we selected the κ values shown in Table 1. At order 3 we could obtain the desired efficiency and stability without difficulty. However, at order 4 we could achieve no more than a 12% increase in efficiency and still get a stability angle of 66°, which is 90% of the angle of 73° of BDF4. Stability is especially important at order 5 because it is the highest we plan to use and because the stability of BDF5 is significantly worse than that of lower order formulas. After finding that we could not achieve worthwhile improvements in efficiency without unacceptable reductions in stability, we chose $\kappa = 0$, hence took NDF5 = BDF5.

Klopfenstein did not attempt to improve the accuracy of BDF1 in the same way that he did with BDF2. Perhaps he did not want to lose the one step nature of BDF1. Considering that both the predictor and the error estimator already make use of a previously computed solution value, this would not really be a loss. We derived a more accurate member of the family of NDF's at order 1 just as Klopfenstein did at order 2. The numerical differentiation formula of order 1 is

$$y_{n+1} - y_n - \kappa\,(y_{n+1} - 2y_n + y_{n-1}) = hF(t_{n+1}, y_{n+1})$$

The boundary of the stability region of a linear multistep method consists of those points z for which the characteristic equation $\rho(\theta) - z\sigma(\theta) = 0$ has a root θ of magnitude 1. For NDF1

$$\begin{aligned}\rho(\theta) &= (1-\kappa)\theta^2 - (1-2\kappa)\theta - \kappa\\ \sigma(\theta) &= \theta^2\end{aligned}$$

The root-locus method obtains the boundary as a subset of the points $z = \rho(\theta)/\sigma(\theta)$ as $\theta = \exp(i\psi)$ ranges over all numbers of magnitude 1. For this formula

$$Re(z) = 1 - (1-2\kappa)\cos(\psi) - 2\kappa\cos^2(\psi)$$

If $1 - 2\kappa \geq 0$, then

$$Re(z) \geq 1 - (1-2\kappa) - 2\kappa = 0$$

A sufficient condition for these formulas to be A-stable is that $1/2 \geq \kappa$ because then there are no points of the boundary of the stability region in the left half complex plane. At order 1 the requirement that the formula be A-stable does not prevent the

truncation error coefficient from being reduced to 0. However, the coefficient cannot be made too small, else the formula would not "look" like a first order formula (the leading term in the Taylor series expansion of the error would not dominate). We chose the parameter so that the improvement in efficiency at order 1 would be the same as at order 2, namely 26%. This is achieved with $\kappa = -0.1850$.

The roles of stability and accuracy are different in practical computation. The stability of the family of formulas implemented in a variable-order code governs its applicability, the class of problems that can be solved efficiently by the code. It is well-known [37] that the popular variable-order BDF codes might choose to integrate with a formula that suffers from a stability restriction rather than a lower order formula that does not. This is the reason that the codes limit the maximum order; none uses order 6 and some do not use order 5. In addition, users may be given the option of reducing the maximum order still further. In light of this one might argue that the applicability of a variable-order BDF code is determined more by the least stable member of the family than by the stability of individual members. Further, if two variable-order codes are both applicable to a given problem, meaning that all the formulas are stable for the problem, it does not matter if the formulas in one code have larger A(α)-stability regions than those in the other. It *does* matter if the formulas in one code are more efficient than those in the other. As a family of formulas of orders 1-5, the NDF's we have selected are significantly more efficient than the BDF's and have about the same applicability to stiff problems.

It is plausible that a fully variable step size implementation would be more stable than the quasi-constant step size implementation we have discussed and there are some BDF codes, e.g., [5] and [8], that use it. However, Curtis [10] argues that the difficulties with a quasi-constant step size implementation reported in [8] are the result of an insufficiently flexible step size selection rather than the approach itself. Also, there are theoretical results [18] to the effect that with acceptable constraints on how often the order is changed, integration with a quasi-constant step size implementation is stable. Even in the case of the BDF's there have been few investigations of stability when the step size is not constant. Reiher [31] investigated the family of second order NDF's when the step size is not constant. Such investigations are rather complicated and it is difficult to be optimistic about the prospects for choosing appropriate values of κ for a fully variable step size implementation. Because quasi-constant step size implementations have been quite satisfactory in practice, we have not pursued numerical differentiation formulas for fully variable step size.

3.2. Changing the Step Size. Each of the popular representations of the BDF formulas has its advantages and disadvantages. Backward differences are very convenient for implementing both the BDF's and NDF's. The basic algorithms are simple and can be coded compactly in a way that takes advantage of the efficient handling of arrays in MATLAB. But a disadvantage of using backward differences is the difficulty of changing the step size. A new scheme is developed here that is based on an interesting factorization of the dependence of a difference table at one step size to the table at a different step size.

There are efficient ways of halving and doubling the step size due to Krogh [26], but we consider the general case because we prefer more flexibility in the selection of step size. Suppose that a constant step size h has been used and the integration has reached t_n. There are available solution values $y(t_{n-j})$ at $t_{n-j} = t_n - jh$ for

$j = 0, 1, ..., k$. The interpolating polynomial $P(t)$ is

$$P(t) = y(t_n) + \sum_{j=1}^{k} \nabla^j y(t_n) \frac{1}{j! h^j} \prod_{m=0}^{j-1} (t - t_{n-m})$$

By definition $\nabla^j P(t_n) = \nabla^j y(t_n)$. In this representation the solution is held in the form of the current value $y(t_n)$ and a table of backward differences

$$D = \left[\nabla P(t_n), \nabla^2 P(t_n), ..., \nabla^k P(t_n)\right]$$

Changing to a new step size $h^* \neq h$ amounts to evaluating $P(t)$ at $t^* = t_n - jh^*$ for $j = 0, 1, ..., k$ and then forming

$$D^* = \left[\nabla^* P(t_n), \nabla^{*2} P(t_n), ..., \nabla^{*k} P(t_n)\right]$$

Here the asterisk on the backward difference ∇^* indicates that it is for step size h^*.

Equating the representations of $P(t)$ with step sizes h and h^* leads to the identity

$$\sum_{j=1}^{k} \nabla^{*j} P(t_n) \frac{1}{j! h^{*j}} \prod_{m=0}^{j-1} \left(t - t^*_{n-m}\right) = \sum_{j=1}^{k} \nabla^j P(t_n) \frac{1}{j! h^j} \prod_{m=0}^{j-1} (t - t_{n-m})$$

Evaluating this identity at $t = t^*_{n-r}$ for $r = 1, ..., k$ leads to the system of equations

$$\sum_{j=1}^{k} \nabla^{*j} P(t_n) U_{jr} = \sum_{j=1}^{k} \nabla^j P(t_n) R_{jr}$$

In matrix terms this is

$$D^* U = DR$$

and we seek a convenient expression for D^* in terms of D.

The entries of the $k \times k$ matrix U are

$$U_{jr} = \frac{1}{j! h^{*j}} \prod_{m=0}^{j-1} \left(t^* - t^*_{n-m}\right) = \frac{1}{j!} \prod_{m=0}^{j-1} (m - r)$$

Evidently $U_{jr} = 0$ for $r < j$ so that U is upper triangular. For $r > j$

$$U_{jr} = (-1)^j \binom{r}{j}$$

The diagonal entries of U have magnitude 1 so the matrix is non-singular, but U is quite special because $U^2 = I$. To see this, we first note that U^2 is upper triangular. Then for $r \geq i$,

$$\begin{aligned}
\left(U^2\right)_{ir} &= \sum_{j=i}^{r} (-1)^i \binom{i}{j} (-1)^j \binom{r}{j} \\
&= \binom{r}{i} \sum_{s=0}^{r-s} (-1)^s \binom{r-i}{s} \\
&= \binom{r}{i} (1-1)^{r-i}
\end{aligned}$$

This shows that $\left(U^2\right)_{ii} = 1$ and $\left(U^2\right)_{ir} = 0$ for $r \neq i$, hence $U^2 = I$.

The $k \times k$ matrix R is full. In terms of $\rho = h^*/h \neq 1$, its entries are

$$R_{jr} = \frac{1}{j!} \prod_{m=0}^{j-1} (m - r\rho)$$

Because this matrix depends on the new step size, it must be constructed each time the step size is changed. This can be done efficiently by columns according to

$$R_{1,r} = -r\rho$$
$$R_{j+1,r} = R_{j,r} \left(\frac{j - r\rho}{j+1} \right)$$

Multiplying the basic relation on the right by U, we arrive at an interesting factorization that shows how the difference tables depend on the step sizes:

$$D^* = D\,(RU)$$

The entries of U are integers that do not depend on h nor on k, so they can be defined in a code for the maximum k permitted once and for all. Because the matrix R depends on the step sizes h and h^*, it must be computed each time the step size is changed. The computation is simple and inexpensive. In MATLAB this way of changing the step size is compact and efficient.

3.3. The `ode15s` Program. The `ode15s` code is a quasi-constant step size implementation in terms of backward differences of the Klopfenstein-Shampine family of NDF's. The user interface is common to all the codes in the suite, but some options apply only to the codes for stiff problems or even only to this particular code. For example, as we observed in connection with the derivation of the NDF's, it is easy to accomodate the BDF's in this framework. The user is, then, given the option of integrating with the classic BDF's rather than the default choice of the NDF's. Also, the user can reduce the maximum order from the default value of 5 should this appear desirable for reasons of stability.

To implement the NDF formula, we rewrite the Klopfenstein form to make its evaluation efficient. The identity

$$\sum_{m=1}^{k} \frac{1}{m} \nabla^m y_{n+1} = \gamma_k \left(y_{n+1} - y_{n+1}^{(0)} \right) + \sum_{m=1}^{k} \gamma_m \nabla^m y_n$$

shows that equation (5) is equivalent to

$$(1 - \kappa)\,\gamma_k \left(y_{n+1} - y_{n+1}^{(0)} \right) + \sum_{m=1}^{k} \gamma_m \nabla^m y_n - hF(t_{n+1}, y_{n+1}) = 0$$

(This generalizes a form of BDFk that we learned from F.T. Krogh.) When the implicit formula is evaluated by a simplified Newton (chord) method, the correction to the current iterate

$$y_{n+1}^{(i+1)} = y_{n+1}^{(i)} + \Delta^{(i)}$$

is obtained by solving

$$\left(I - \frac{h}{(1-\kappa)\,\gamma_k} J\right) \Delta^{(i)} = \frac{h}{(1-\kappa)\,\gamma_k} F(t_{n+1}, y_{n+1}^{(i)}) - \Psi - \left(y_{n+1}^{(i)} - y_{n+1}^{(0)}\right)$$

Here J is an approximation to the Jacobian of $F(t,y)$ and

$$\Psi = \frac{1}{(1-\kappa)\,\gamma_k} \sum_{m=1}^{k} \gamma_m \nabla^m y_n$$

is a quantity that is fixed during the computation of y_{n+1}. This is done by defining

$$d^{(i)} = y_{n+1}^{(i)} - y_{n+1}^{(0)}$$

and computing

$$\begin{aligned} d^{(i+1)} &= d^{(i)} + \Delta^{(i)} \\ y_{n+1}^{(i+1)} &= y_{n+1}^{(0)} + d^{(i+1)} \end{aligned}$$

Proceeding in this manner the fundamental quantity $\nabla^{k+1} y_{n+1}$ is obtained directly and more accurately as the limit of the $d^{(i)}$.

Where the "previously computed" solution value comes from at the start of the integration is worth comment. The standard device is to predict then with the first order Taylor series (forward Euler) method. The Nordsieck (Taylor) representation is in terms of y_0 and hy_0'. It is so natural to set $hy_0' = hF(t_0, y_0)$ in this representation that it is rarely appreciated that this amounts to forming a fictitious y_{-1}. The equivalent for the backward difference form is to set $\nabla y_0 = hF(t_0, y_0)$. After initializing in this way, the start is no different from any other step.

Many of the tactics adopted in the code resemble those found in the well-known codes DIFSUB [17], DDRIV2 [24], LSODE [22], and VODE [7]. In particular, local extrapolation is not done. The selection of the initial step size follows Curtis [10] who observes that by forming partial derivatives of $F(t,y)$ at t_0, it is possible to estimate the *optimal* initial step size when starting at order 1.

In the context of MATLAB it is natural to retain a copy of the Jacobian matrix. Of the well-known codes mentioned, only VODE exploits this possibility. VODE employs a number of devices with the aim of using a matrix factorization as long as possible. This does not seem appropriate to us when working in MATLAB with its fast linear algebra and Jacobians of modest size or sparse Jacobians. We form and factor the iteration matrix for the simplified Newton method every time the step size or order is changed. Our basic approach is to form a new Jacobian only when the simplified Newton iteration is converging too slowly. A small point is that we approximate the Jacobian at the current solution (t_n, y_n) rather than at a predicted solution $(t_{n+1}, y_{n+1}^{(0)})$ as is done in the other codes cited. This means that it is never necessary to approximate a Jacobian more than once at a given point in the integration. Also, it is consistent with what must be done with another method that we take up in Section 4. As in [34], we monitor the rate of contraction of the iteration and predict how far the current iterate is from y_{n+1}. The process is terminated if it is predicted that convergence will not be achieved in four iterations. Should this happen and the Jacobian not be current, a new Jacobian is formed. Unlike VODE which forms a Jacobian every now and then, this is the only time that Jacobians are formed in

`ode15s`. If the Jacobian is current, the step size is reduced by a factor of 0.3 and the step tried again.

Our scheme for reusing Jacobians means that when the Jacobian is constant, the code will normally form a Jacobian just once in the whole integration. There is an option of informing the codes that the Jacobian is constant, but it is of little importance to `ode15s` for this reason. It is pointed out in [35] that the well-known set of stiff test problems [15] contains a good many problems with constant Jacobians and that this would cause difficulties of interpretation when the codes were "smart" enough to take advantage of the fact. VODE and `ode15s` *do* recognize these problems and form *very* few Jacobians when solving them. Our scheme also results in `ode15s` forming very few Jacobians when it is applied to a problem that is not stiff. The codes in the suite that are intended for non-stiff problems are more efficient when applied to non-stiff problems than those intended for stiff problems. This said, it must be added that `ode15s` does surprisingly well when applied to a non-stiff problem because it forms few Jacobians and the extra linear algebra is performed efficiently in MATLAB.

4. Linearly Implicit Formulas for Stiff Systems. In this section it is convenient at first to consider diferential equations in autonomous form

$$y' = F(y)$$

Rosenbrock methods have the form

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i$$

where the k_i are obtained for $i = 1, 2, ..., s$ by solving

$$Wk_i = F\left(y_n + h\sum_{j=1}^{i-1} a_{ij}k_j\right) + hJ\sum_{j=1}^{i-1} d_{ij}k_j$$

Here $W = I - hdJ$ and $J = \partial F(y_n)/\partial y$. Rosenbrock methods are said to be linearly implicit because the computation of y_{n+1} requires the solution of systems of linear equations.

A Rosenbrock formula makes explicit use of the Jacobian evaluated at the beginning of the step. It is some trouble for users to supply a function for the Jacobian and it can be quite inconvenient to work out analytical expressions for the partial derivatives. For this reason codes like ROS4 [21] permit the use of numerical Jacobians and others require it. This may be satisfactory, but the order of a Rosenbrock formula *depends* on J being equal to $\partial F(y_n)/\partial y$ and it is difficult to compute reliably accurate Jacobians by numerical means.

A number of authors have explored formulas of the form stated without assuming that J is equal to $\partial F(y_n)/\partial y$. Wolfbrandt studied W-formulas, formulas that make *no* assumption about the relationship of J to the Jacobian. A second order formula presented in [40] sparked considerable interest. It is delightfully simple:

$$\begin{aligned} Wk_1 &= F\left(y_n\right) \\ Wk_2 &= F\left(y_n + \frac{2}{3}hk_1\right) - \frac{4}{3}hdJk_1 \\ y_{n+1} &= y_n + \frac{h}{4}\left(k_1 + 3k_2\right) \end{aligned}$$

Here the parameter $d = 1/\left(2+\sqrt{2}\right)$. The order of this formula does not depend on J, but its stability does. If J is equal to the Jacobian, the formula is L-stable. As noted in [40], W-methods reduce to explicit Runge-Kutta formulas when $J = 0$. This particular formula reduces to the improved Euler method.

One of the attractions of Rosenbrock and W-methods is that they are one-step. However, it is not easy to retain this property when estimating the error, and the error estimate of [40] for Wolfbrandt's formula gives it up. Scraton [33] managed to achieve a one-step error estimate for this formula by making stronger assumptions about J. Specifically, he assumed that

$$J = \frac{\partial F}{\partial y}(t_n, y_n) + hB + O\left(h^2\right)$$

He and others making this assumption have motivated it by the possibility of using a Jacobian evaluated previously to avoid the cost of forming a current Jacobian. It also seems a reasonable assumption for describing approximations to current Jacobians that are obtained numerically. The fact is, we must assume that J is approximately equal to $\partial F/\partial y$ if we are to apply the usual linear stability theory to conclude the good stability that leads us to consider such formulas in the first place. We shall describe formulas based on Scraton's assumption as modified Rosenbrock formulas.

The error estimate that Scraton derived for Wolfbrandt's formula is not qualitatively correct at infinity. In his dissertation, Zedan modified Scraton's estimator to correct this and also derived an estimator in the conventional form of a pair of formulas of orders 2 and 3. Both approaches to error estimation lead to one-step formulas. In the case of the Wolfbrandt pair, if J is the Jacobian, the companion formula of order 3 is A-stable. Zedan made a clever choice of parameters to obtain a companion that does not require an additional evaluation of F; at each step three systems of linear equations are solved with the same matrix W and there are two evaluations of F. The investigation is described in [43] and a code based on this pair and a higher order pair is to be published [42].

The paper [38] shows how to improve further Scraton's estimate to obtain the correct behavior at infinity. We began developing a code based on Wolfbrandt's formula with this estimate and learned the hard way that the formula and its various error estimates all have a fundamental defect. In the remainder of this section, we explain the difficulty and develop a new formula and error estimate that avoids it. We also develop an interpolant. We close with some details of the implementation of the new triple in the program `ode23s`.

4.1. A Modified Rosenbrock Triple. Wolfbrandt's formula and the error estimates cited all evaluate F only at t_n and $t_n + 2h/3$. As pointed out in [36], schemes of this kind can go badly wrong when solving problems with solutions that exhibit very sharp changes, "quasi-discontinuities". If a solution component should change sharply between $t_n + 2h/3$ and $t_n + h$, the formula and its error estimate may not recognize that the change took place. As far as the formula can discern, the solution changes smoothly throughout the step. Because a one-step formula does not make use of previously computed solution values, the sharp change is not recognized after the step to $t_n + h$. Quasi-discontinuities are very real possibilities when solving stiff problems and robust software must be able to deal with them.

A family of second order, L-stable W-methods depending on one parameter is presented in [40] and there seems to be no strong reason for favoring the one that reduces to the improved Euler method when $J = 0$. Zedan's derivation of a companion

third order formula was done solely for Wolfbrandt's formula, but using the equations of condition from his thesis or [43], it is easy enough to derive A-stable companions for other choices of parameter. Unfortunately, it is not possible to obtain a pair that makes only two evaluations of F with one being at t_{n+1}. There is, however, another way to achieve this, and that is to find a pair that is FSAL, First Same As Last. By making the first evaluation of F for the next step the same as the last of the current step, we have at our disposal another function evaluation that is usually "free" because most steps are successful.

The W-method that is the basis of our procedure reduces to another familiar explicit Runge-Kutta method when $J = 0$, namely the midpoint Euler method. In practice Rosenbrock methods are written in a form that avoids unnecessary matrix multiplications, see e.g. [36]. The formula that follows is written in such a form and is given for non-autonomous equations. When advancing a step from (t_n, y_n) to $t_{n+1} = t_n + h$, the modified Rosenbrock formula is

$$\begin{aligned}
F_0 &= F(t_n, y_n)\\
k_1 &= W^{-1}(F_0 + hdT)\\
F_1 &= F(t_n + 0.5h, y_n + 0.5hk_1)\\
k_2 &= W^{-1}(F_1 - k_1) + k_1\\
y_{n+1} &= y_n + hk_2\\
F_2 &= F(t_{n+1}, y_{n+1})\\
k_3 &= W^{-1}[F_2 - e_{32}(k_2 - F_1) - 2(k_1 - F_0) + hdT]\\
error &\approx \frac{h}{6}(k_1 - 2k_2 + k_3)
\end{aligned}$$

Here

$$\begin{aligned}
W &= I - hdJ\\
J &\approx \frac{\partial f}{\partial y}(t_n, y_n)\\
d &= 1/\left(2 + \sqrt{2}\right)\\
T &\approx \frac{\partial f}{\partial t}(t_n, y_n)\\
e_{32} &= 6 + \sqrt{2}
\end{aligned}$$

If the step is a success, the F_2 of the current step is the F_0 of the next; the formula is FSAL. Because the pair samples at both ends of the step, it can recognize a quasi-discontinuity.

The derivation of a FSAL formula requires a decision about whether the integration is to be advanced with the higher order result, local extrapolation, or the lower. We have experimented a little with both possibilities and have come to favor advancing with the lower order formula. In large measure this is because the lower order formula is a W-formula, so is less sensitive to the quality of the approximate Jacobian. With this choice and a good approximation to the Jacobian, the lower order formula is L-stable and the higher is A-stable. This implies that the error estimate is unduly conservative at infinity. It could be corrected in the manner of [38], but we have found more reliable results are obtained with the more conservative estimate.

As mentioned in Section 2, we considered it essential that all codes in the new suite have an interpolation capability, a matter scarcely considered previously for

modified Rosenbrock methods. On p. 129 of his dissertation, Zedan describes briefly a scheme he uses in a research code. In our notation, after a successful step from t_n to $t_n + h$, he represents the solution throughout the step with the quadratic polynomial that interpolates y_n, y_{n+1}, and $F_0 = F(t_n, y_n)$. Using F_0 is dangerous; it is easily seen by considering the test equation $y' = \lambda y$ that the behavior of the interpolant is unsatisfactory when $h\lambda$ is large. We have derived an interpolant in a manner customary for explicit Runge-Kutta formulas. The idea is to derive a formula of the same type for computing a solution at $t_n + h^*$ when $h^* = sh$. The matrix $W^* = I - h^* d^* J$, so if we take the parameter $d^* = d/s$, then $W^* = W$. It is easily found that with similar definitions, it is possible to obtain a second order W-method for the intermediate value that requires no major computations not already available from the step itself. Specifically, it is found that a second order approximation to $y(t_n + sh)$ is provided by

$$y_{n+s} = y_n + h\left[\frac{s(1-s)}{1-2d}k_1 + \frac{s(s-2d)}{1-2d}k_2\right]$$

It is seen by inspection that this quadratic interpolant is continuous. It has better behavior than the one depending on F_0, in essence because of the W^{-1} implicit in the k_i. The formula, its error estimate, and the interpolant displayed above form a new modified Rosenbrock triple.

4.2. The `ode23s` Program. The code `ode23s` based on the triple presented above provides an alternative to `ode15s`. `ode23s` is especially effective at crude tolerances, when a one-step method has advantages over methods with memory, and when Jacobians have eigenvalues near the imaginary axis. It is a fixed order method of such simple structure that the overhead is low except for the linear algebra, and linear algebra is fast in MATLAB. The integration is advanced with the lower order formula, so `ode23s` does not do local extrapolation. To achieve the same stability in `ode15s`, the maximum order would have to be restricted to 2, the same as the order in `ode23s`.

Although our implementation of `ode23s` resembles that of an ordinary Rosenbrock method with numerical Jacobians, in principle it is quite different because a modified Rosenbrock method does not depend on having accurate Jacobians, just reasonable approximations. To enhance the robustness of `ode23s`, we employ an exceptionally strong algorithm for the computation of numerical Jacobians that is described in Section 8.

We have considered algorithms along the lines of one mentioned in [11] for recognizing when a new Jacobian is needed and we have also considered tactics like those of [40] and [42] for this purpose. This is promising and we may revisit the matter, but for this version of `ode23s` we decided to form a new Jacobian at every step for several reasons. First of all, a formula of order 2 is most appropriate at crude tolerances. At such tolerances solution components often change significantly in the course of a single step, so it is often appropriate to form a new Jacobian. Secondly, the Jacobian is typically of modest size or sparse, so that its evaluation is not very expensive compared to evaluating F. Finally, we placed a premium on the reliability and robustness of `ode23s`, which is enhanced by evaluating the Jacobian at every step. A special case is sufficiently common to warrant special attention. If the code is told that the Jacobian is constant, it will evaluate the Jacobian just once rather than at every step.

For non-autonomous problems `ode23s` requires an approximation to $\partial f/\partial t$ in addition to the approximation to $\partial f/\partial y$. For the convenience of the user and to make

the use of all the codes the same, we have chosen always to approximate this partial derivative numerically. Because the step size h provides a measure of the time scale, approximating $\partial f/\partial t$ is a less delicate task than approximating $\partial f/\partial y$.

5. Stiff Systems of More General Form. In this section we consider how to modify the computational schemes developed for the NDF's and the modified Rosenbrock method so as to solve efficiently a system of the more general form

$$M(t)\,y' = f(t,y)$$

with a mass matrix $M(t)$ that is non-singular and (usually) sparse. Of course, the schemes could be applied in their original form to the equivalent system

$$y' = F(t,y) = M^{-1}(t)\,f(t,y)$$

but small modifications make it possible to avoid the inconvenience and expense of this transformation.

The modified Rosenbrock method was derived for autonomous systems and its application to these more general problems is awkward when M depends on t. For this reason `ode23s` has been extended only to problems with constant M. We see how to modify the evaluation of the formulas by applying them to the equivalent problem $y' = F(t,y)$ and rewriting the computations in terms of M and $f(t,y)$. The first stage k_1 is obtained by solving

$$Wk_1 = (I - hdJ)k_1 = F_0 + hdT$$

Here

$$\begin{aligned} F_0 &= M^{-1}f(t_0,y_0) = M^{-1}f_0 \\ J &\approx \frac{\partial F}{\partial y} = M^{-1}\frac{\partial f}{\partial y} \\ T &\approx \frac{\partial F}{\partial t} = M^{-1}\frac{\partial f}{\partial t} \end{aligned}$$

Scaling the equation for k_1 by M leads to

$$\left(M - hd\frac{\partial f}{\partial y}\right)k_1 = f_0 + hd\frac{\partial f}{\partial t}$$

This form allows the user to pose the problem in terms of M and $f(t,y)$. Moreover, proceeding in this way avoids repeated solution of the linear equations that arise when the differential equation is transformed in order to apply the usual form of the method. A similar modification is possible for all the computations. The usual form of the method is recovered when $M = I$.

The `ode15s` code allows the mass matrix to depend on t. The requisite modifications can be deduced just as with a constant mass matrix except at one point. The simplified Newton method used to evaluate the implicit formula solves linear systems with the iteration matrix

$$I - \frac{h}{(1-\kappa)\gamma_k}J$$

and right hand sides involving

$$F(t_{n+1},y_{n+1}^{(i)}) = M^{-1}(t_{n+1})f(t_{n+1},y_{n+1}^{(i)})$$

Here

$$J \approx \frac{\partial F}{\partial y}(t_m, y_m) = M^{-1}(t_m)\frac{\partial f}{\partial y}(t_m, y_m)$$

for some $m \leq n$. Because the mass matrix depends on t, it is not possible to remove all the inverses by a simple scaling of the equations. The standard procedure exploits the fact that the iteration matrix is itself an approximation. The equations are scaled by $M(t_{n+1})$ to remove all the inverses from the right hand sides and then the scaled iteration matrix is approximated further as

$$M(t_m) - \frac{h}{(1-\kappa)\gamma_k}J$$

where

$$J \approx \frac{\partial f}{\partial y}(t_m, y_m)$$

With this approximation, it is again found that the computational scheme can be written in a form that reduces to the usual one when $M(t) = I$.

6. Explicit Formulas for Non-Stiff Systems. Previous versions of MATLAB had two codes, `ode45` and `ode23`, based on explicit Runge-Kutta formulas. These programs have been replaced in the new suite by codes with the same names that remedy some deficiencies in the original design and take advantage of significant developments in the theory and practice of Runge-Kutta methods. The new suite also adds an explicit Adams code, `ode113`, based on a PECE implementation of Adams-Bashforth-Moulton methods.

6.1. The `ode45` Program. Of the two explicit Runge-Kutta codes in the previous generation of MATLAB solvers, it appears that the code based on the lower order pair was favored. This seems to be mainly because plots obtained using the code based on the higher order pair are sometimes not smooth. Also, the higher order pair is not as reliable as the lower at the very lax default tolerance of the old `ode23`. These objections are overcome in the new suite by the use of more stringent default tolerances and new formulas.

Workers in the field employ a number of quantitative measures for evaluating the quality of a pair of formulas. A pair of formulas due to Dormand and Prince [12] has become generally recognized as being of high quality and significantly more efficient than the Fehlberg pair used in the previous version of `ode45`. The authors of the pair refer to it as RK5(4)7FM, but it is also known as DOPRI5, DP(4,5), and DP54. It is FSAL and was constructed for local extrapolation.

Exploiting the freedom provided by a seven stage FSAL formula, Dormand and Prince [13] obtained a number of inexpensive interpolants. They have communicated to us another interpolant of order 4 that is of high quality and "free". By "free" is meant that after a step is accepted, it is possible to construct an interpolant valid throughout the span of the step with no additional evaluations of F. Because solution components can change substantially in the course of a single step, the values computed at the end of each natural step may not provide adequate resolution for graphical display of the solution. This is remedied by computing intermediate values by interpolation, specifically by computing four values spaced evenly within the span of each natural step. Plotted in Figure 1 are the solution components of the non-stiff initial value problem `rigid`, the Euler equations of a rigid body without external

forces as proposed by Krogh. The solid curves result from the default output of `ode45`, the discrete values are the results at the end of each step, and the dashed curves result from connecting these results. Clearly the values obtained by interpolation between the natural output points are necessary for adequate representation of the solution.

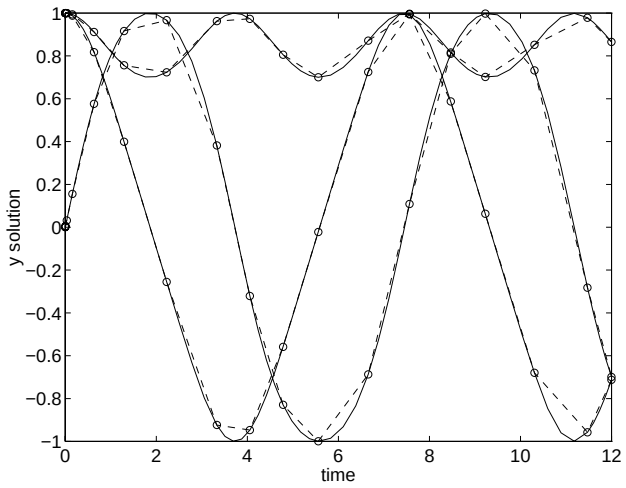

FIG. 1. *The* `rigid` *solution as computed by* `ode45`*. The continuous curves result from the default output of* `ode45`*. The discrete values are the results at the end of each step. The dashed curves show what happens when the new interpolation capability is disabled.*

The implementation of `ode45` was influenced both by its predecessor and by RKSUITE [4]. Though the other codes for non-stiff problems in the suite, `ode23` and `ode113`, are to be preferred in certain circumstances, `ode45` is the method of choice.

6.2. The `ode23` Program. The new version of `ode23` is based on the Bogacki-Shampine $(2,3)$ pair [3] (see also [37]). This FSAL pair was constructed for local extrapolation. In the standard measures, the pair is of high quality and significantly more efficient than the pair used in the previous version of `ode23`. Accurate solution values can be obtained throughout a step for free by cubic Hermite interpolation to the values and slopes computed at the ends of the step. The pair is used in a number of recent software projects, including RKSUITE, the Texas Instruments TI-85 engineering graphics calculator, and the teaching package Differential Systems of Gollwitzer [20].

At the default tolerances `ode23` is generally more expensive than `ode45`, but not by a great deal, and it is to be preferred at cruder tolerances. The advantage it enjoys at crude tolerances is largely because a step in `ode23` is about half as expensive as a step in `ode45`, hence the step size is adjusted more often. When a step fails in `ode23`, fewer evaluations of F are wasted. This is particularly important in the presence of mild stiffness because of the many failed steps then. The stability regions of the $(2,3)$ pair are rather bigger than those of the $(4,5)$ pair when scaled for equal cost, so `ode23` is advantageous in the presence of mild stiffness.

6.3. The `ode113` Program. The code `ode113` is a descendant of the variable-order Adams-Bashforth-Moulton suite ODE/STEP, INTRP [39]. Though the code differs considerably in detail, its basic algorithms follow closely those of STEP and need no amplification here. Like the other codes for non-stiff problems, it does local

extrapolation. Adams methods are based on polynomial interpolants that are used to obtain output at specific points. The authors of ODE/STEP, INTRP, tried to obtain as cheaply as possible solutions of moderate to high accuracy to problems involving F that are expensive to evaluate and to do so as reliably as possible. This was accomplished by monitoring the integration very closely and by providing formulas of quite high orders. In the present context the overhead of this monitoring is comparatively expensive. Although more than graphical accuracy is necessary for adequate resolution of solutions of moderately unstable problems, the high order formulas available in `ode113` are not nearly as helpful in the present context as they are in general scientific computation. These considerations lead us to expect that `ode113` will usually be the most efficient code in the suite in terms of the number of evaluations of F, but not the fastest.

Close monitoring of the solution is always valuable for reliability, and when F is expensive and/or high accuracy is desired, `ode113` is the fastest code. It is also relatively efficient in the presence of mild stiffness. This is partly because it can resort to formulas with relatively large scaled stability regions and partly because only one evaluation of F is wasted on a failed step.

7. User Interface. Every author of an ODE code wants to make it as easy as possible to use. At the same time the code must be able to solve reliably typical problems. It is not easy to reconcile these goals. In developing an approach to the task, we had the benefit of experience with many codes. Naturally, experience with the previous generation of codes in MATLAB, the Runge-Kutta codes `ode45` and `ode23`, was very important. Other codes that were especially influential are the suite of explicit Runge-Kutta codes RKSUITE [4], the BDF and Adams-Moulton code DDRIV2 [24], and the Adams-Bashforth-Moulton suite ODE/STEP, INTRP [39].

In the MATLAB environment we considered it essential that it be possible to use all the codes in *exactly* the same way. However, we also considered it essential to provide codes for the solution of stiff problems. Methods for stiff problems all make use of approximations to the Jacobian, $\partial F/\partial y$, of $F(t, y)$. If a code for solving stiff problems is to look like a code for non-stiff problems, it is necessary to conceal the approximation of partial derivatives by doing it numerically. Unfortunately, it is difficult to approximate Jacobians reliably. Moreover, when it is not inconvenient to supply some information about the structure of the Jacobian, it can be quite advantageous. Indeed, this information is crucial to the solution of "large" systems. Clearly a user interface to a code for stiff problems must allow for the provision of additional information and this without complication when users do not have the additional information or do not think it worth the trouble of supplying. The same is true of other optional information, so a key issue is how to make options unobtrusive. Further, the design must be extendable. We have already had experience with this. After writing the codes for the solution of stiff problems, we added the capability of working with sparse Jacobians. Providing this capability typically complicates the interface so much that hitherto authors have preferred to write a separate code for the task. Our approach deals with it easily. Later we again extended the codes for stiff problems so that they could solve problems of more general form and again we found that the approach dealt with it easily.

It is possible to use all five codes in the suite, `ode15s`, `ode23s`, `ode45`, `ode23`, `ode113`, in precisely the same manner, so in the examples that follow the code `ode15s` is generic. An initial value problem can be solved by

```
[tout,yout] = ode15s('yprime',tspan,y0);
```

The results can then be displayed with the usual plotting tools of MATLAB, e.g., by

```
plot(tout,yout)
```

In the syntax of MATLAB, the arguments of `ode15s` are pure input arguments and the arguments on the left of the assignment statement are pure output arguments. Specification of the general problem cannot be made easier because the input here is the minimum needed just to define the mathematical problem. The argument `yprime` is a string providing the name of a function that defines the differential equation. For example, a non-stiff version of the van der Pol equation written as a first order system is coded as function `vdpns` by creating a file `vdpns.m` shown in Figure 2.

```
function dy = vdpns(t,y)
dy = zeros(2,1);          % preallocate column vector dy
dy(1) = y(2);
dy(2) = (1-y(1)^2)*y(2)-y(1);
```

FIG. 2. *The* MATLAB *code for the initial value problem* `vdpns`.

(MATLAB displays the result of a line unless it ends with a semicolon. It is not necessary to declare the sizes of arrays nor to declare data types.) The interval of integration is provided in the vector `tspan=[t0,tfinal]` and the initial conditions in the vector `y0`. `ode15s` needs the number of equations `neq`, but it is not necessary for the user to supply this because the code can obtain it from `y0` by using an intrinsic for the length of a vector: `neq = length(y0)`.

A feature of the language that we have exploited to simplify the user interface is the ability to handle a variable number of input and output arguments. As a simple example, `ode15s` monitors the cost of the integration with measures such as the number of steps and the number of Jacobian evaluations. To see these statistics, an extra output argument is used. Namely, the left hand side of the assignment has the form `[tout,yout,stats]` instead of `[tout,yout]`. The user who is not interested in these statistics ignores this optional output.

The language provides for empty arrays. We have exploited this for a number of purposes, one being the definition of the entire initial value problem in a single file. We illustrate this in Figure 3 with the file `chm6ex.m` containing a stiff initial value problem for modelling catalytic fluidized bed dynamics. This is a standard test problem (CHM6 of [14]) proposed by Luss and Amundson [28]. Some numerical results for this problem will be presented later.

The built-in function `length` seen in Figure 3 is used to test whether `t` is an empty array. With the function coded in this manner, `ode15s` can be invoked with empty or missing arguments for `tspan` and `y0`, e.g. `ode15s('chm6ex',[],[])` or `ode15s('chm6ex')`. Then when initializing, `ode15s` will call `chm6ex` with an empty argument for `t` to obtain the information not supplied via the call list.

Having defined the mathematical problem, there remains the definition of the computational problem. One part of this is specification of output. Codes produce answers at "natural" steps, i.e. steps they choose so as to solve the problem both efficiently and reliably. These steps tend to cluster where solution components change rapidly, and for many methods answers at these points generally furnish satisfactory

```
function [out1,out2,out3] = chm6ex(t,y)
if length(t) == 0          % return default tspan, y0, options
  out1 = [0; 1000];
  out2 = [761; 0; 600; 0.1];
  out3 = odeset('atol',1e-13);
  return;
end
dy = zeros(4,1);           % preallocate column vector dy
K = exp(20.7 - 1500/y(1));
dy(1) = 1.3*(y(3) - y(1)) + 10400*K*y(2);
dy(2) = 1880 * (y(4) - y(2) * (1+K));
dy(3) = 1752 - 269*y(3) + 267*y(1);
dy(4) = 0.1 + 320*y(2) - 321*y(4);
out1 = dy;
```

FIG. 3. *The* MATLAB *code for the initial value problem* `chm6ex`.

plots. This is true, for example, of `ode23` in previous versions of MATLAB. However, as pointed out by Polking [30], the previous version of the code `ode45` may take steps so large that plots are not smooth. If the method allows it, the intermediate values needed for smooth plots can be obtained inexpensively by interpolation. Interpolation can also be used to obtain at little cost as many answers at specific points as a user desires. Event location is quite a useful capability. By this is meant integrating until an event occurs, e.g. until a particular solution component vanishes. Although we have not provided this functionality at this time, we intend to do so and interpolation is key to event location. For all these reasons, we considered for the new suite only methods for which interpolation is possible.

The question now is how to specify where answers are desired. The convenient interface of DDRIV2 produces answers wherever the user specifies. This possibility is mandatory, but the design is not convenient for graphical output because the user generally does not know where to ask for answers. It may seem natural to specify that answers be obtained at equally spaced points in `[t0,tfinal]`, but with the wide range of values typical of stiff problems, this often fails to provide an acceptable plot of the solution. In our design the default is for the codes to report answers at the natural steps. This design is made convenient by the language because it is not necessary to specify in advance the sizes of output arrays. We also provide an option of refinement, meaning that the code is told to produce a specified number of answers at equally spaced points in the span of each natural step. The default value of `refine` is 1 for all the codes in the suite, i.e. answers at natural steps only, except for the new `ode45`, for which it is 4, reflecting the fact that solution components can change significantly in the course of a typical step of the method implemented in this code.

To deal with output at specific points, we overload the definition of `tspan` and use the length of this vector to dictate how its values are to be interpreted. An input `tspan` with two entries means that output at the natural steps is desired, the default. If `tspan` contains more than two entries, the code is to produce output at these points, and only at these points, so that `tout=tspan`. Because output is obtained by interpolation, the number and placement of specified output points has little effect on the cost of the integration. Occasionally users want an answer only at `tfinal`. This is somewhat awkward in our design, but we have provided for it by using a value 0

for `refine` to specify no intermediate output.

Another part of the computational problem is specification of the error control. A discussion of this difficult matter is to be found in [37]. There it is explained how the simplifying assumptions made in the earlier versions of `ode23` and `ode45` can lead to unsatisfactory results. Polking [30] makes the same point. There seems to be no way to provide automatically a default control of the error that is completely satisfactory, another reason why we must provide for optional input.

We have now seen several examples of important optional input. Control of the error applies to all the codes and information about Jacobians applies only to some. There are instances in the suite of important, but optional, input that applies to a single code. It is difficult to accomodate all the possibilities without complicating the interface to the point that users despair. A traditional approach is to use an options vector. We do this too, but with some innovations that make the design unobtrusive and powerful. The first point is that the options vector is optional. When it is employed, the syntax of a call to the solver is

```
[tout,yout] = ode15s('yprime',tspan,y0,options);
```

The vector `options` is built by means of the function `odeset`. Like others, we recognize the virtues of specifying options by means of keywords. However, in MATLAB we can specify values for the options that are of different data types. This makes possible a very powerful and convenient interface. Some examples will illustrate the possibilities.

The most common options are those associated with the error control. `ode15s` uses a mixed relative-absolute error test and a maximum norm. Accordingly the local error e_i in y_i is estimated at each step and it is required that

$$|e_i| \leq r\,|y_i| + a_i$$

for each i. The scalar relative error tolerance r has a default value of 10^{-3}. The vector of absolute error tolerances a has by default all its values equal to 10^{-6}. Whether these values are appropriate depends on the scaling of the problem. Numerical examples are taken up later that cannot be solved properly without providing more suitable absolute error tolerances. Quite often one wants to provide a scalar value for the absolute tolerance a that is to be assigned to all elements of the array used internally, and a number of popular codes provide for this. This is done elegantly in a language which makes it possible for a code to recognize whether the value input is a scalar or an array. Naturally an unobtrusive design requires that options be set in any order and default values be used for any quantity not explicitly set by the user. A number of things are done to make the interface even more convenient. The `odeset` function recognizes some synonyms, allowing, e.g., the string `'abstol'` to substitute for `'atol'` when specifying the absolute error tolerance. It is not necessary to provide values for Boolean variables; the presence of the name alone instructs `odeset` to assign the value "true" to the variable. It is not necessary to provide all the name to `odeset`, just sufficiently many of its leading characters to identify the option. If what is provided is not sufficient, the function reports this along with possible completions of the name.

As a concrete example of setting options, suppose that we wish to solve a stiff problem with a constant Jacobian, the scaling is such that absolute error tolerances of 10^{-20} are appropriate for all components, and we wish to impose a maximum step size of 3500 to assure that the code will recognize phenomena occurring on this time scale. This can be done with

```
options = odeset('constantJ','atol',1e-20,'hmax',3500);
```

Here `constantJ` is a Boolean option that is set "true" by this call and the other entries are obvious. When performing a number of integrations, the function `odeset` can be used to reset and/or set additional options in a previously specified options vector. For example, changing the maximum step size to 5000 and specifying a relative error tolerance of 10^{-2} is accomplished by

```
options = odeset(oldoptions,'hmax',5000,'rtol',1e-2);
```

An illuminating example is provided by the `chm6ex` problem shown earlier. The equations for this problem are

$$
\begin{aligned}
K &= \exp\left(20.7 - 1500/y_1\right) \\
y_1' &= 1.3\left(y_3 - y_1\right) + 10400 K y_2, \quad && y_1(0) = 761 \\
y_2' &= 1880\left(y_4 - y_2\left(1 + K\right)\right), && y_2(0) = 0 \\
y_3' &= 1752 - 269 y_3 + 267 y_1, && y_3(0) = 600 \\
y_4' &= 0.1 + 320 y_2 - 321 y_4, && y_4(0) = 0.1
\end{aligned}
\tag{8}
$$

The problem is posed on the interval $[0, 1000]$. Its solution is discussed in [27] and a log-log plot of y_2 is shown in Section 9 as Figure 5.

A fundamental difficulty with this problem is that with an initial value of 0, there is no natural measure of scale for $y_2(t)$. As it happens, this component never gets bigger than about 7×10^{-10}. Clearly an absolute error tolerance like the default of 10^{-6} in the suite assures no accuracy at all in this component. Some accuracy may be achieved as the code tries to achieve a specified accuracy in components that are influenced by this one, but it is found that numerical results can be unreliable unless some accuracy is required directly of the component. With the default relative tolerance of 10^{-3} and an optional absolute error tolerance of 10^{-13} on all components, this problem is solved routinely by `ode15s` in only 139 steps. However, the step sizes range from 5×10^{-14} to 10^{2}! This is mainly due to $y_2(t)$ changing rapidly on a very short time scale. It rises from the initial value of 0 to an approximately constant maximum value by $t = 10^{-11}$ and returns to a value of about 0 by $t = 10^{-2}$. This behavior is not likely to be visible if output is taken at equally spaced points in an interval of length 10^3. A design that returns output at points chosen by the code does reveal the behavior of this component. On plotting the output it is seen that a logarithmic scale in t would be more appropriate. Because all the solution values are provided to users in our design and they are retained by the language, they can be displayed in a more satisfactory way without having to recompute them.

In order that all codes in the suite have the same appearance to the user, the codes intended for stiff problems by default compute internally the necessary partial derivatives by differences. How this somewhat delicate task is accomplished efficiently and reliably is detailed in another section. Users are given the option of providing a function for the analytical evaluation of the Jacobian. (The value of the `analyticJ` option is a string for the name of the function.) They are also given the option of specifying that the Jacobian is constant, a special case that leads to significant savings in `ode23s`. (The value of the `constantJ` option is Boolean.) The default is to treat the Jacobian as being a full matrix. If the Jacobian is sufficiently sparse that it is worth taking sparsity into account, the codes must be informed of the sparsity

pattern. The distinction between banded Jacobians and the much more complicated case of general sparse Jacobians that is important in other codes is absent in the new suite. All that a user must do is provide a matrix S of zeros and ones that represents the sparsity pattern of the Jacobian. It is not necessary to know anything about how the language deals with sparse matrices [19]. Some details will make the point. The matrix S might be defined at the command line or in an M-file. There are a number of ways this might be done, but the most general and most straightforward is as follows. If there are neq equations, an $neq \times neq$ sparse matrix of zeros is first created by

```
S = sparse(neq,neq);
```

Then for each equation i in $F(t,y)$, if y_j appears in the equation, the (i,j) entry of S is to be set to 1 by

```
S(i,j) = 1;
```

These quantities can be set in any order. If the Jacobian has a regular structure, it may be possible to define S more compactly by means of one of the powerful commands for sparse matrices. For example, when the Jacobian is banded with bandwidth $2m+1$, S can be defined in a single line by

```
S = spdiags(ones(neq,2m+1),-m:m,neq,neq);
```

It is remarkably easy to take advantage of sparse Jacobians. In a call to `odeset` the option `sparseJ` is assigned the value S (a sparse matrix).

The two codes for stiff problems permit a more general form of the differential equation, namely

$$M(t)\,y' = f(t,y)$$

with a mass matrix $M(t)$ that is non-singular and (usually) sparse. `ode23s` is restricted to a constant mass matrix. The more general form raises some issues that are awkward in other computing environments. One is the specification of the mass matrix and its structure. Another is the relationship between the structure of $M(t)$ and that of the Jacobian. In our interface, the codes are informed of the presence of a mass matrix by means of the `mass` option. The value of this option is the name of a function that returns $M(t)$. No special attention need be devoted to the structure of the mass matrix because the language recognizes whether a matrix is sparse and does linear algebra in an appropriate and efficient manner. A convenience made possible by the language is that if the mass matrix is constant, M itself can be passed to the codes as the value of the option. However, if it should prove convenient to evaluate a constant mass matrix by means of a function when using `ode15s`, the integration is performed more efficiently if the Boolean option `constantM` is set true. (This option is set true automatically in `ode23s`.)

Many libraries have been formed by collecting codes in wide use that were written by different people in different circumstances and modifying them so that they present a similar appearance to the user. All the codes of the new suite were written at the same time by the same people. Because of this and a decision to make the codes as much alike as possible, there is a remarkable homogeneity. In only one instance did we start with an existing code and the experience is illuminating. The

ODE/STEP, INTRP suite [39] formed the basis for `ode113`. Because the codes are organized in rather different ways, `ode113` is not a simple translation. Even the translation of key portions of STEP was not straightforward because it is written in FORTRAN 66, a language that lacks features considered necessary nowadays for readability. For example, the arithmetic IF is used in STEP to handle compactly three way decisions common in this code. (The cover of [39] shows two instances.) For a person intimately familiar with FORTRAN 66, the coding is clear enough, but the arithmetic IF has fallen into disuse as FORTRAN has evolved and the coding would no longer be regarded as clear. There is an important point here. The powerful commands in MATLAB for dealing with arrays permit very compact and efficient code that can be obscure to the person not intimately familiar with the language. As a rule users do not care about the coding, but they do care about how quickly their problems are solved. For this reason we have tried to program all the solvers in a clear way, but have given more weight to efficiency than to clarity. For instance, some computations that are naturally coded in terms of `for` loops are coded instead with fancy indexing and vectorized, built-in functions because this makes them significantly faster.

8. Numerical Partial Derivatives. Methods for the solution of stiff problems involve partial derivatives of the function defining the differential equation. Many codes (including those of the MATLAB ODE suite) allow users to provide a routine for evaluating these derivatives. This routine might be constructed by automatic differentiation of the code for evaluating the function. Another possibility is to use a computer algebra system to derive and code expressions for the derivatives. These tools are becoming more powerful and more widely available, but it is still typical that some portion of a routine for partial derivatives be derived and/or coded by hand. This is so much trouble for users and so prone to error that the default in all the popular codes is to approximate partial derivatives internally by numerical differentiation.

In the MATLAB ODE suite, the program `numjac` is used to compute numerically the Jacobian $\partial F/\partial y$ of the function $F(t,y)$. The algorithms used in `numjac` were determined by the needs of the stiff solvers `ode15s` and `ode23s`. It is important to appreciate that the roles of the partial derivatives differ in the two codes. G.D. Byrne has often pointed out to one of us that a modern BDF code like VODE [7] can cope with remarkably poor approximate Jacobians and the same is true of `ode15s`. However, as discussed in Section 4, a modified Rosenbrock code like `ode23s` requires the best Jacobian that can be obtained at "reasonable" cost. This affected considerably the algorithms used in `numjac`.

The scaling difficulties that are possible when approximating partial derivatives by differences are well-known [37]. `numjac` is an implementation of an exceptionally robust scheme due to D. E. Salane [32] for the approximation of partial derivatives in the context of integrating ordinary differential equations. A key idea is to use experience gained at one step to select good increments for difference quotients at the next step. This responds to scaling difficulties by recognizing that when a problem is stiff, the solution changes slowly so that scale information gathered when forming partial derivatives at one step is relevant when the next set of partial derivatives is formed.

At present very few solvers exploit this idea. One is the DDRIV2 code [24] of D. Kahaner and C. D. Sutherland which implements much of Salane's algorithm. Another is the METAN1 code [2] of G. Bader and P. Deulfhard, a code based on the extrapolated semi-implicit midpoint rule. Because this linearly implicit method

is related to modified Rosenbrock methods, we share the interest these authors had in obtaining high-quality partial derivatives. Like Salane, they are willing to recompute a column of the Jacobian matrix if they are in doubt as to whether the values might consist only of roundoff. This should not often be necessary, but it seems to us to be worth the expense in the present context, so we have also implemented this aspect of Salane's algorithm. Brenan, Campbell, and Petzold [6] apparently share our views. Starting on p. 132 they write

> *It is a difficult problem to devise an algorithm for calculating the increments for the differencing which is both cheap and robust. ... It is likely that in future versions of DASSL a more robust finite difference Jacobian algorithm such as the one proposed by Salane will be considered.*

The `numjac` code can generate full or sparse Jacobian matrices. A full Jacobian matrix `dFdy` is obtained by the invocation

```
[dFdy,fac] = numjac('F',t,y,Fty,thresh,fac,vectorized);
```

The argument `'F'` is a string naming the function M-file that defines the differential equation, the current point in the integration is `(t,y)`, and `Fty` is the vector given by `'F'` evaluated at `(t,y)`. The vector `thresh` provides a threshold of significance for `y`, i.e. the exact value of a component `y(i)` with magnitude less than `thresh(i)` is not important. The vector `fac` is working storage for scale information from previous calls to `numjac`. Because of the syntax of MATLAB the array must appear on both sides of the assignment. The first time that the solver calls the program, it does so with `fac=[]`. `numjac` interprets this as an instruction to initialize the `fac` array.

The Boolean option `vectorized` allows `numjac` to exploit the array nature of MATLAB. In the MATLAB language it is generally an easy matter to code the function `'F'` so that it can return an array of function values. For example, the vectorized code for the van der Pol equation in relaxation oscillation is shown in Figure 4. Here `.^` and `.*` are MATLAB notations for element-by-element operations.

```
function dy = vdpex(t,y)
dy = zeros(size(y));      % preallocate column vector dy
dy(1,:) = y(2,:);
dy(2,:) = 1000*(1 - y(1,:).^2).*y(2,:) - y(1,:);
```

FIG. 4. *The vectorized* MATLAB *code for the* `vdpex` *problem.*

If function `'F'` has been coded so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]`, then `vectorized` should be set true. When the function evaluation has been vectorized, `numjac` approximates all columns of the Jacobian with a single call to `'F'`. This avoids the cost of repeatedly calling the function and it may reduce the cost of the evaluations themselves.

When the Jacobian is structured, the user specifies its sparsity pattern to the solver by means of the options vector and a non-empty sparse matrix `S` of zeros and ones. The solver then makes a call of the form

```
[dFdy,fac,g] = numjac('F',t,y,Fty,thresh,fac,vectorized,S,g);
```

When `numjac` is invoked with a sparsity pattern `S`, it returns a sparse matrix `dFdy`. The first time that the solver calls `numjac`, it does so with `g=[]`. `numjac` interprets this as an instruction to call `colgroup(S)` to find groups of columns of `dFdy` that can be approximated with a single call to `'F'`. This is done only once and the grouping is saved in `g`. `colgroup` tries two schemes (first-fit and first-fit after reverse column minimum-degree ordering [19]) and returns the better grouping. The result may not be optimal because finding the smallest packing is an NP-complete problem equivalent to K-coloring a graph [9].

The partial derivative `dFdt` plays a different role. The modified Rosenbrock code `ode23s` requires this derivative every time it requires `dFdy`. In contrast, the NDF code `ode15s` requires it only at the initial point where it is used for selecting automatically the initial step size. On reaching t, the step size h provides a measure of scale for the approximation of `dFdt` by a forward difference. The computation is so simple when coded in MATLAB that it is done in line. The situation at the initial point is a little tricky because the value of `dFdt` is to be used to select the step size h. The codes use the rule of thumb discussed in [37], p. 377 ff., to obtain a first approximation to h. This value is then used to compute `dFdt` and a better approximation to h.

9. Examples. In this section, we present some experiments with the MATLAB ODE suite applied to stiff and non-stiff test problems taken from classic collections [15, 14, 35, 39]. All stiff problems were coded to take advantage of the `vectorized` option when computing numerical Jacobians. When solving the four stiff problems with constant Jacobians (`a2ex`, `a3ex`, `b5ex`, `hb3ex`), the `constantJ` option was used.

9.1. Stiff Examples. We begin by examining the advantages of using NDF's instead of BDF's in the `ode15s` code. As mentioned in Section 3, the `ode15s` program allows users to integrate with the classic BDF's rather than the default choice of the NDF's. Table 2 shows the number of steps and the real time required for the two choices when applied to a set of 13 stiff problems. For all but one problem (`hb2ex`), `ode15s` using the default NDF's takes fewer steps than when using the BDF's (an average of 10.9% fewer), and for *all* problems, the code is faster when using the NDF's (an average of 8.2% faster).

Next we compare `ode15s` using the NDF's to the popular code DDRIV2 [24] on some relatively difficult problems. For reasons that we take up shortly, it is not possible to compare the codes in detail. However, our goal is merely to verify that the performance of `ode15s` is comparable to that of a modern BDF code, and the codes are similar enough for this purpose. DDRIV2 is an easy-to-use driver for a more complex code with an appearance not too different from `ode15s`. It is a quasi-constant step size implementation of the BDF's of orders 1-5 that computes answers at specified points by interpolation. It approximates Jacobians internally by differences with an algorithm related to that of `ode15s`.

An obvious and very important obstacle to comparing DDRIV2 and `ode15s` is that they are implemented in different languages for different computing environments. A fundamental difference is that they cannot be used to solve exactly the same computational problem. For one thing, the error controls are different. DDRIV2 measures the error in a solution component relative to the larger of the magnitude of the component and a threshold, and the error is measured in a root-mean-square norm. It is possible to make the controls roughly equivalent by taking the threshold to be equal to the desired absolute error and dividing the tolerances given to DDRIV2 by the square root of the number of equations.

TABLE 2
Comparison of the NDF's and BDF's in `ode15s`*. Times are measured as seconds on a Sparc2.*

	BDF steps	NDF steps	percent fewer	BDF time	NDF time	percent faster
`a2ex`	118	101	14.4	3.60	3.14	12.8
`a3ex`	134	130	3.0	3.96	3.87	2.4
`b5ex`	1165	936	19.7	32.58	25.95	20.4
`buiex`	57	52	8.8	2.05	1.92	6.4
`chm6ex`	152	139	8.6	4.05	3.63	10.3
`chm7ex`	57	39	31.6	1.82	1.48	18.4
`chm9ex`	910	825	9.3	30.53	29.38	3.8
`d1ex`	67	62	7.5	2.35	2.29	2.5
`gearex`	20	19	5.0	1.12	1.08	3.5
`hb1ex`	197	179	9.1	5.57	5.09	8.5
`hb2ex`	555	577	-4.0	13.49	13.45	0.3
`hb3ex`	766	690	9.9	19.79	17.77	10.2
`vdpex`	708	573	19.1	20.75	19.33	6.9

Another difficulty is that the two codes handle output differently. DDRIV2 provides answers wherever requested, but only at those points. We asked the codes to produce 150 answers equally spaced within the interval of integration. This is inadequate for some of the examples, but asking for more answers could increase the cost significantly in DDRIV2 because it has an internal maximum step size that is twice the distance between output points. Accepting a possible reduction in efficiency in `ode15s`, we used an option to impose the same maximum step size on this code.

Table 3 compares the performance of DDRIV2 to that of `ode15s` using the NDF's. We interpret these comparisons as showing that `ode15s` is an effective and efficient code for the solution of stiff problems. DDRIV2 does not save Jacobians and the numerical results indicate that to some degree `ode15s` is trading linear algebra for a smaller number of approximate Jacobians. Because these examples involve just a few equations, the benefits of reusing Jacobians are masked. The individual experiments are discussed below.

TABLE 3
Comparison of DDRIV2 to `ode15s` *using the NDF's. The table shows the number of successful steps, the number of failed steps, the number of function calls, the number of partial derivative evaluations, the number of LU decompositions, and the number of linear system solutions.*

example	code	time steps	failed steps	f evals	$\partial f/\partial y$ evals	LU's	linear solves
`chm6ex`	DDRIV2	218	6	404	33	33	271
	`ode15s`	177	4	224	2	29	213
`chm9ex`	DDRIV2	1073	217	2470	220	220	1802
	`ode15s`	750	227	2366	83	322	2033
`hb2ex`	DDRIV2	1370	162	2675	176	176	2316
	`ode15s`	939	70	1321	4	165	1308
`vdpex`	DDRIV2	871	185	1836	167	167	1497
	`ode15s`	724	197	1965	33	261	1865

The chemical reaction problem `chm6ex` was described in Section 7. For this prob-

lem, it is observed that `ode15s` is able to solve the problem effectively using a remarkably small number of Jacobians. Figure 5, a log-log plot of the second component of the problem on a time interval of $[0, 1000]$, indicates that this problem exercises the step size control of `ode15s`. The step size ranged over 15 orders of magnitude. With an absolute error tolerance of 10^{-13}, the `ode15s` code traversed the interval in only 139 steps.

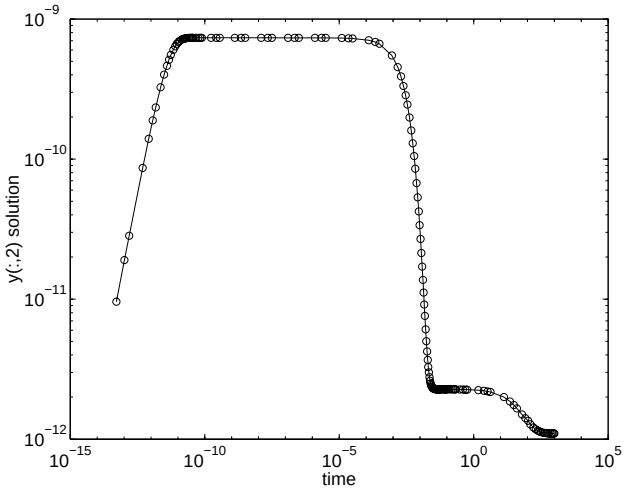

FIG. 5. *A log-log plot of the second component of the solution of* `chm6ex`.

The standard test problem `chm9ex`, CHM9 of [14], is a scaled version of the Belousov oscillating chemical reaction. A discussion of this problem and plots of the solution components on various intervals are found in [1] starting on p. 49. The limit solution is periodic and exhibits regions of very sharp change. We chose the time interval $[0, 650]$ so as to include two complete periods and part of another. The default error tolerances (`rtol`=10^{-3} and `atol`=10^{-6}) were used. As compared to DDRIV2, it is seen that `ode15s` is trading some linear algebra for a reduction in the number of Jacobians. Since the problem involves only three solution components, reducing the number of Jacobians does not have a great effect on the number of function evaluations.

Hindmarsh and Byrne present in Example 2 of [23] the non-autonomous problem `hb2ex` that arises in a diurnal kinetics model and they discuss its solution with EPISODE. They make the point that the scaling of one component is such that an absolute error tolerance of 10^{-20} is needed. The problem is also discussed at length in [24] where it is solved with DDRIV2. We compared `ode15s` to DDRIV2 in the solution of this problem and used plots like that of p. 312 of [24] to judge the adequacy of the solution. As the measures of cost given in Table 3 show, `ode15s` performs quite well in comparison to a good BDF code. For this problem, the reuse of Jacobians proved to be quite advantageous. With only two equations, numerical evaluation of a Jacobian normally costs merely two additional function evaluations. Clearly the difference in cost is not entirely due to saving Jacobians.

Finally we solved `vdpex`, the van der Pol equation in relaxation oscillation in the form specified in [35]. The tradeoffs in the tuning of `ode15s` as compared to DDRIV2 are clear. Because this problem involves only two solution components, more frequent

Jacobians would probably have been a bargain in `ode15s`.

Next, we compare the performance of the modified Rosenbrock code `ode23s` to `ode15s`. For two stiff problems, Table 4 shows the time required for solution, the number of successful steps, the number of failed steps, the number of function calls, the number of partial derivative evaluations, the number of LU decompositions, and the number of linear system solutions. Note that Table 4 contains results for `ode15s` using both the NDF's and the BDF's. The fact that the NDF's of orders 1 and 2 are more efficient than the BDF's of these orders is particularly evident for the `b5ex` experiments in which the maximum order was limited.

TABLE 4

Comparison of `ode23s` *to* `ode15s`*. For* `vdpex`*, the relative accuracy was changed from the default of* 0.1% *to* 1%*. Times are measured as seconds on a Sparc2.*

example	code	time	time steps	failed steps	f evals	$\partial f/\partial y$ evals	LU's	linear solves
`vdpex`	`ode15s` (BDF)	17.06	525	203	1594	45	266	1458
	`ode15s` (NDF)	15.83	490	179	1514	46	249	1375
	`ode23s`	14.21	302	96	1706	303	398	1194
`b5ex`	`ode15s` (BDF)	32.37	1165	124	2586	1	319	2578
	`ode15s` (NDF)	26.23	936	97	2074	1	263	2066
	`ode23s`	15.38	549	17	1689	1	566	1698

As a first example we solved `vdpex` but asked only for a relative accuracy of 1.0% (i.e. $rtol = 10^{-2}$). At this accuracy `ode23s` requires about 14 seconds for the integration and `ode15s` using the NDF's requires about 16. This is true despite the fact that `ode23s` forms 303 Jacobians and `ode15s` only 46. In Figure 6, we see that the plot of $y(t)$ resulting from the use of `ode23s` is notably better than that of `ode15s` at this rough accuracy. When the relative accuracy tolerance is tightened to the default of 0.1%, `ode23s` requires about 29 seconds and `ode15s` requires about 19. At this more stringent tolerance, the plot of the solution resulting from the use of `ode15s` is completely satisfactory, as seen in Figure 6.

Problem B5 of the test set [15] was devised specifically to show the unsatisfactory behavior of variable-order BDF codes when applied to a problem for which the Jacobian has eigenvalues near the imaginary axis. Gaffney [16] investigates the matter further with other problems and other kinds of codes. We solved `b5ex` with the default error tolerances. `ode15s` recognizes automatically that this problem has a constant Jacobian. The problem is an artificial one and the fact that the Jacobian is constant was not considered important. However, for a fair comparison, we used the option of telling both codes that the Jacobian is constant. In addition we restricted the maximum order of the NDF's used by `ode15s` to 2, the highest order that is L-stable, to have equivalent stability in both codes. It is evident that `ode23s` is to be preferred for the solution of this problem, despite a relative accuracy of 10^{-3}.

When it is not inconvenient to supply a function for evaluating the Jacobian, it can be quite advantageous. Although the algorithms used by `ode15s` and `ode23s` for numerical approximation of the Jacobian are strong, the reliability of an integration can be enhanced by an analytical Jacobian. It is generally assumed that it is less expensive to evaluate a Jacobian analytically than numerically. This may be true, or it may not. As it happens, for the simple examples we present here, it is less expensive to evaluate Jacobians analytically. Whether it is worth providing a function for the

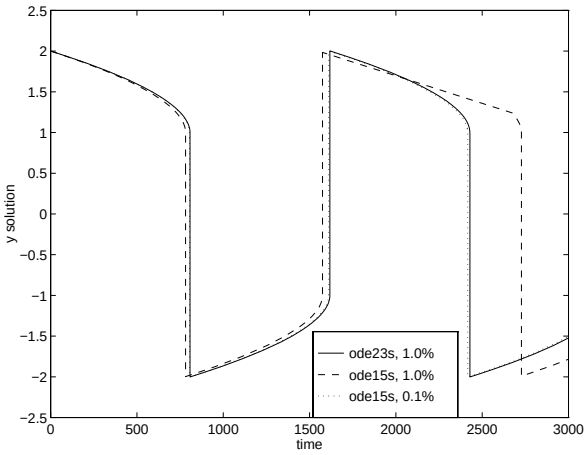

FIG. 6. *The* `vdpex` *solution as computed by* `ode23s` *and* `ode15s` *at 1.0% accuracy. The solution of* `ode15s` *at 0.1% accuracy is also plotted to show that the 1.0% solution of* `ode15s` *is less accurate than that of* `ode23s`.

Jacobian depends on how difficult it is to write out the Jacobian, how expensive integrations are, and which integrator is used. Certainly it is easy enough to write out the Jacobians for our examples, and MATLAB has a Symbolic Toolbox that provides assistance working out analytical Jacobians when the equations are more complicated. For four of the most expensive of our examples with non-constant Jacobians, Table 5 shows the effects of using the `analyticJ` option to pass in the name of a function that evaluates the Jacobian. Note that `brussjac`, the analytic Jacobian function of the `brussex` problem, returns a 100×100 sparse matrix. Because `ode23s` evaluates the Jacobian at every step, reducing the cost of this evaluation has an important effect on the overall cost. Reducing this cost is much less significant when using `ode15s` because it makes comparatively few evaluations of the Jacobian.

TABLE 5

The solutions of four problems by `ode23s` *and* `ode15s` *showing the effect of using the* `analyticJ` *option to supply a function for evaluation of the Jacobian (time2). The* `brussex` *problem is* 100×100 *and its Jacobian function returns a sparse matrix. Times are measured as seconds on a Sparc2.*

	`ode23s`		`ode15s`	
problem	time	time2	time	time2
`brussex`	80.00	13.82	23.48	9.70
`chm9ex`	47.12	27.32	29.60	25.86
`hb2ex`	140.78	87.29	13.65	13.50
`vdpex`	28.61	16.36	19.51	19.35

Next, we examine the solution of large systems of equations when information about the sparse structure of the Jacobian is passed to the code with the `sparseJ` option. As a large stiff test example, we use `brussex`, the classic "Brusselator" system of $2N$ equations modelling diffusion in a chemical reaction [21],

$$u_i' = 1 + u_i^2 v_i - 4u_i + \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1})$$
$$v_i' = 3u_i - u_i^2 v_i + \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1})$$

where $\alpha = 1/50$ and

$$\left.\begin{array}{rcl} u_i(0) & = & 1 + \sin(2\pi x_i) \\ v_i(0) & = & 3 \end{array}\right\} \text{with } x_i = i/(N+1) \text{ for } i = 1, \ldots, N$$

on the time interval $[0, 10]$. There are $2N$ equations in this system. When the equations are ordered as $u_1, v_1, u_2, v_2, \ldots$, the Jacobian of this system is banded with width 5.

For progressively larger values of N, Table 6 shows the number of steps taken and compares the number of seconds required to solve the `brussex` problem by `ode45`, `ode23s` and `ode15s`. The `ode45` results indicate that the system becomes quite stiff for the larger N. The first columns of `ode23s` and `ode15s` results were produced using the default numerical approximation of Jacobians. As the second columns show, the `sparseJ` option makes a tremendous difference. For the `brussex` problem, `numjac` requires only 4 additional function evaluations to approximate the sparse Jacobian matrix for any N. Until N becomes large, `ode15s` is efficient even without the `sparseJ` option because it generates relatively few Jacobians.

TABLE 6

The solutions of various size `brussex` *problems by* `ode45`, `ode23s`, `ode23s` *using the* `sparseJ` *option (time2),* `ode15s`, *and* `ode15s` *using the* `sparseJ` *option (time2). Times are measured as seconds on a Sparc2.*

	`ode45`		`ode23s`			`ode15s`		
size	steps	time	steps	time	time2	steps	time	time2
100	629	143.95	59	80.10	15.04	82	23.37	10.36
200	2458	4052.99	59	499.44	24.50	82	104.49	17.78
400	NA	NA	59	3398.47	43.00	85	574.42	32.19
600	NA	NA	59	NA	62.84	85	1703.68	49.21
800	NA	NA	59	NA	83.91	85	NA	63.51
1000	NA	NA	59	NA	105.93	85	NA	80.74

To illustrate the solution of problems involving a mass matrix M, we use Example 4 of Chapter 6 in [41]. The system of ordinary differential equations comes from a method of lines solution of the partial differential equation

$$e^{-t}\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

with initial condition $u(0, x) = \sin(x)$ and boundary conditions $u(t, 0) = u(t, \pi) = 0$. The analytical solution of this problem is

$$u(t, x) = \exp\left(1 - e^t\right)\sin(x)$$

An integer N is chosen, h is defined as $1/(N+1)$, and the solution of the partial differential equation is approximated at $x_k = k\pi h$ for $k = 0, 1, \ldots, N+1$ by

$$u(t, x_k) \approx \sum_{k=1}^{N} c_k(t)\phi_k(x)$$

Here $\phi_k(x)$ is a piecewise linear function that is 1 at x_k and 0 at all the other x_j. The

Galerkin discretization leads to the system of ordinary differential equations

$$A(t)\,c' = R\,c \quad \text{where} \quad c(t) = \begin{bmatrix} c_1(t) \\ \vdots \\ c_N(t) \end{bmatrix}$$

and the tridiagonal matrices $A(t)$ and R are given by

$$A_{ij} = \begin{cases} \exp(-t)2h/3 & \text{if } i = j \\ \exp(-t)h/6 & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad R_{ij} = \begin{cases} -2/h & \text{if } i = j \\ 1/h & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

The initial values $c(0)$ are taken from the initial condition for the partial differential equation.

Because the mass matrix A depends on t, this equation can be solved with `ode15s` but not `ode23s`. However, scaling the equation by $\exp(t)$ results in an equivalent system with a constant mass matrix that can be solved with both codes. As is typical of the method of lines, the mass matrix of this example is sparse and generally it is very advantageous to account for this in the solvers. But in this instance, we have followed [41] by taking $N = 9$, which is too small for sparsity to be important. As the analytical solution of the partial differential equation makes clear, the solution decays very rapidly, so the problem is set on the time interval $[0, \pi]$. The solution of `fem2ex` is shown in Figure 7 and statistics are presented in Table 7.

TABLE 7
Comparison of `ode23s` *to* `ode15s` *for a problem with a constant mass matrix,* `fem2ex` *with* $N = 9$. *Times are measured as seconds on a Sparc2.*

example	code	time	time steps	failed steps	f evals	$\partial f/\partial y$ evals	LU's	linear solves
`fem2ex`	`ode15s`	4.71	46	14	175	5	21	124
	`ode23s`	5.35	40	1	493	41	41	123

9.2. Non-stiff Examples. In this section we consider four non-stiff examples drawn from classic collections of test problems [15, 39]. `vdpns` is the van der Pol equation with $\mu = 1$. `rigid` is the Euler equations of a rigid body without external forces as proposed by Krogh. The solution is displayed in Figure 1. `twobody`, D5 of [15], is the two body problem with an elliptical orbit of eccentricity 0.9. And `r3body` describes a periodic orbit for a restricted three body problem [39]. The two and restricted three body problems are somewhat unstable in portions of the interval of integration, so must be solved with tolerances more stringent than the default values to represent adequately their solutions. The details are not needed for our purpose here.

Because the problems are non-stiff, we can solve them with all the MATLAB ODE solvers. Table 8 contains the time required for solution, the number of successful steps, the number of failed steps, the number of function calls, the number of partial derivative evaluations, the number of LU decompositions, and the number of linear system solutions.

10. Conclusion. This paper reports the development of methods and a suite of programs for the solution of initial value problems for systems of first order ordinary differential equations in the context of MATLAB, a widely used environment for

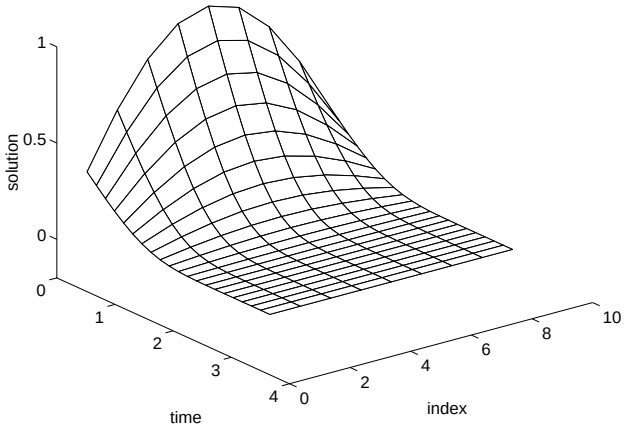

FIG. 7. *The* `fem2ex` *solution with* $N = 9$ *as computed by* `ode15s`.

scientific computing. The environment imposes constraints on the methods and the way they are used, but it also offers interesting possibilities for both algorithms and software, some of which could be realized in languages like C and FORTRAN 90.

New methods and variations on recent ones were developed for the solution of stiff problems. One of the programs in the suite, `ode15s`, is based on a family of implicit linear multistep formulas, the numerical differentiation formulas, NDF's, developed here. The NDF's are substantially more efficient than the backward differentiation formulas, BDF's, and of about the same applicability. A new way of changing the step size when using backward differences was devised that is both compact and efficient in MATLAB. Another program for stiff problems, `ode23s`, is based on a linearly implicit one-step method developed here. This modified Rosenbrock formula and its error estimate remedy a serious defect in a formula of Wolfbrandt [40] and its various error estimates found in the literature. An interpolant is derived that is free and of better quality than one previously proposed for Wolfbrandt's formula. The design of the interface makes it possible for users to solve stiff problems without giving any thought to Jacobians. An exceptionally strong routine for the numerical approximation of partial derivatives makes this possible. `ode15s` minimizes the number of Jacobians formed, ordinarily only one when the Jacobian is constant and a very few when the problem is not stiff. The design makes it convenient to solve large systems of equations with general sparse Jacobians.

Three programs were developed for the solution of non-stiff problems. Two are completely new versions of the explicit Runge-Kutta programs `ode23` and `ode45` found in earlier versions of MATLAB. They are based on new formulas that are more reliable and more efficient. The formulas also have high-quality interpolants that are free. A powerful variable-order Adams-Bashforth-Moulton program, `ode113`, was developed for the problem with exceptionally expensive F and/or stringent accuracy requirements.

A primary concern of our investigation was the software interface. Exploiting the MATLAB language it is possible to devise a user interface that is unobtrusive, powerful, and extendable. Indeed, in the course of developing the suite, the interface was first

TABLE 8
Comparison of all five of the ODE solvers on a set of four non-stiff problems. Times are measured as seconds on a Sparc2.

example	code	time	time steps	failed steps	f evals	$\partial f/\partial y$ evals	LU's	linear solves
`rigid`	`ode23s`	2.76	58	10	373	59	68	204
	`ode15s`	2.08	82	17	184	1	30	179
	`ode113`	2.12	65	4	135	0	0	0
	`ode23`	0.92	55	13	205	0	0	0
	`ode45`	0.57	19	2	127	0	0	0
`r3body`	`ode23s`	22.79	372	1	2612	373	373	1119
	`ode15s`	8.74	321	48	575	1	87	569
	`ode113`	10.72	237	20	495	0	0	0
	`ode23`	6.19	301	4	916	0	0	0
	`ode45`	3.84	73	27	601	0	0	0
`twobody`	`ode23s`	44.45	871	1	6105	872	872	2616
	`ode15s`	13.66	584	64	963	2	135	952
	`ode113`	18.46	396	29	822	0	0	0
	`ode23`	11.51	727	0	2182	0	0	0
	`ode45`	4.98	133	35	1009	0	0	0
`vdpns`	`ode23s`	6.65	158	21	836	159	179	537
	`ode15s`	4.48	192	35	426	1	60	422
	`ode113`	5.33	162	12	337	0	0	0
	`ode23`	2.10	146	19	496	0	0	0
	`ode45`	1.43	51	11	373	0	0	0

extended to deal with sparse Jacobians and then to deal with a more general form of the differential equations.

Availability of the Codes. This paper presents mathematical and software developments that are the basis for the MATLAB suite of ODE solvers. Source code for the solvers and examples may be obtained gratis by ftp on ftp.mathworks.com in the pub/mathworks/toolbox/matlab/funfun directory. The solvers require MATLAB version 4.2 or later.

Acknowledgments. We have had the pleasure of correspondence and discussions with many experts whose publications and advice have been crucial to a number of aspects of this project. C. B. Moler helped us with just about every aspect. We have had the benefit of advice from a number of leading experts in explicit Runge-Kutta methods, viz. J. Dormand and P. Prince; M. Calvo, L. Randez, and J. Montijano; and P. Sharp and J. Verner. D. Salane provided us with FORTRAN versions of his algorithm for computing numerical partial derivatives along with helpful comments. T. Coleman provided advice about column grouping strategies. I. Gladwell provided advice about mathematical software for ordinary differential equations. H. Zedan provided copies of his publications that were essential to the development of our modified Rosenbrock method. J. Polking's experience teaching the solution of ordinary differential equations using the previous generation of codes in MATLAB influenced the new generation in a number of ways. We are grateful for all this help.

REFERENCES

[1] R. C. Aiken, ed., *Stiff Computation*, Oxford Univ. Press, Oxford, 1985.

[2] G. Bader and P. Deulfhard, *A semi-implicit mid-point rule for stiff systems of ordinary differential equations*, Tech. Report 114, Institut für Angewandte Mathematik, Universität Heidelberg, Germany, 1981.

[3] P. Bogacki and L. F. Shampine, *A 3(2) pair of Runge-Kutta formulas*, Appl. Math. Letters, 2 (1989), pp. 1–9.

[4] R. W. Brankin, I. Gladwell, and L. F. Shampine, *RKSUITE: A suite of Runge-Kutta codes for the initial value problem for ODEs*, Tech. Report 92-S1, Math. Dept., Southern Methodist Univ., Dallas, 1992.

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

[6] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, Elsevier Science Publishing Co., New York, 1989.

[7] P. N. Brown, G. D. Byrne, and A. C. Hindmarsh, *VODE: a variable-coefficient ODE solver*, SIAM J. Sci. Comput., 10 (1989), pp. 1038–1051.

[8] G. D. Byrne and A. C. Hindmarsh, *A polyalgorithm for the numerical solution of ordinary differential equations*, ACM Transactions on Mathematical Software, 1 (1975), pp. 71–96.

[9] T. F. Coleman, B. S. Garbow, and J. J. More, *Software for estimating sparse Jacobian matrices*, ACM Transactions on Mathematical Software, 11 (1984), pp. 329–345.

[10] A. R. Curtis, *The FACSIMILE numerical integrator for stiff initial value problems*, in Computational Techniques for Ordinary Differential Equations, I. Gladwell and D. K. Sayers, eds., Academic, London, 1980, pp. 47–82.

[11] P. Deuflhard, *Recent progress in extrapolation methods for ordinary differential equations*, SIAM Review, 27 (1985), pp. 505–535.

[12] J. R. Dormand and P. J. Prince, *A family of embedded Runge-Kutta formulae*, J. Comp. Appl. Math., 6 (1980), pp. 19–26.

[13] ———, *Runge-Kutta triples*, Comp. & Maths. with Appls., 12A (1986), pp. 1007–1017.

[14] W. H. Enright and T. E. Hull, *Comparing numerical methods for the solution of stiff systems of ODE's arising in chemistry*, in Numerical Methods for Differential Systems, L. Lapidus and W. Schiesser, eds., Academic, New York, 1976, pp. 45–66.

[15] W. H. Enright, T. E. Hull, and B. Lindberg, *Comparing numerical methods for stiff systems of ODE's*, BIT, 15 (1975), pp. 10–48.

[16] P. W. Gaffney, *A performance evaluation of some FORTRAN subroutines for the solution of stiff oscillatory ordinary differential equations*, ACM Trans. Math. Software, 10 (1984), pp. 58–72.

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

[18] C. W. Gear and D. S. Watanabe, *Stability and convergence of variable order multistep methods*, SIAM J. Numer. Anal., 11 (1974), pp. 1044–1058.

[19] J. R. Gilbert, C. Moler, and R. Schreiber, *Sparse matrices in MATLAB: design and implementation*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 333–356.

[20] H. Gollwitzer, *Differential Systems User Manual*, Dept. Math. & Comp. Sci., Drexel Univ., Philadelphia, 1991.

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

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

[23] A. C. Hindmarsh and G. D. Byrne, *Applications of EPISODE: an experimental package for the integration of systems of ordinary differential equations*, in Numerical Methods for Differential Systems, L. Lapidus and W. Schiesser, eds., Academic, New York, 1976, pp. 147–166.

[24] D. Kahaner, C. Moler, and S. Nash, *Numerical Methods and Software*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[25] R. W. Klopfenstein, *Numerical differentiation formulas for stiff systems of ordinary differential equations*, RCA Review, 32 (1971), pp. 447–462.

[26] F. T. Krogh, *Algorithms for changing the step size*, SIAM J. Numer. Anal., 10 (1973), pp. 949–965.

[27] L. Lapidus, R. C. Aiken, and Y. A. Liu, *The occurrence and numerical solution of physical and chemical systems having widely varying time constants*, in Stiff Differential Systems,

R. Willoughby, ed., Plenum Press, New York, 1974, pp. 187–200.
[28] D. Luss and N. R. Amundson, *Stability of batch catalytic fluidized beds*, AIChE J., 14 (1968), pp. 211–221.
[29] The MathWorks, Inc., *MATLAB 4.2*, 24 Prime Park Way, Natick MA, 1994.
[30] J. C. Polking, *MATLAB Manual for Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
[31] T. Reiher, *Statilitätsuntersuchungen bei rückwärtigen Differentiationsformeln in Abhängigkeit von einem Parameter*, Tech. Report #11, Sektion Mathematik, Humboldt–Universität zu Berlin, 1978.
[32] D. E. Salane, *Adaptive routines for forming Jacobians numerically*, Tech. Report SAND86–1319, Sandia National Laboratories, Albuquerque, NM, 1986.
[33] R. E. Scraton, *Some L-stable methods for stiff differential equations*, Intern. J. Computer Maths., 9 (1981), pp. 81–87.
[34] L. F. Shampine, *Implementation of implicit formulas for the solution of ODE's*, SIAM J. Sci. Statist. Comput., 1 (1980), pp. 103–118.
[35] ———, *Evaluation of a test set for stiff ODE solvers*, ACM Trans. Math. Software, 7 (1981), pp. 409–420.
[36] ———, *Implementation of Rosenbrock methods*, ACM Trans. Math. Software, 8 (1982), pp. 93–113.
[37] ———, *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, New York, 1994.
[38] L. F. Shampine and L. S. Baca, *Error estimators for stiff differential equations*, J. Comp. Appl. Math., 11 (1984), pp. 197–207.
[39] L. F. Shampine and M. K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W. H. Freeman, San Francisco, 1975.
[40] T. Steihaug and A. Wolfbrandt, *An attempt to avoid exact Jacobian and non–linear equations in the numerical solution of stiff differential equations*, Math. Comp., 33 (1979), pp. 521–534.
[41] Visual Numerics, Inc., *IMSL MATH/LIBRARY. FORTRAN subroutines for mathematical applications*, Suite 440, 9990 Richmond, Houston, TX, 1994.
[42] H. Zedan, *A variable order/variable-stepsize Rosenbrock-type algorithm for solving stiff systems of ODE's*, Tech. Report YCS114, Dept. Comp. Sci., Univ. of York, York, England, 1989. (to appear in ACM Trans. Math. Software).
[43] ———, *Avoiding the exactness of the Jacobian matrix in Rosenbrock formulae*, Computers Math. Applic., 19 (1990), pp. 83–89.