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 systems, BDF, Gear method, Rosenbrock method, non-stiff systems, Runge-Kutta method, Adams method, software

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

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

on a time interval $[t_0, t_f]$, 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. This influenced the choice of methods and how they were implemented.

As in many environments, the typical problem in MATLAB is solved interactively and the results displayed graphically. Generally functions defining the differential equations are not expensive to evaluate. The typical stiff problem is either of modest size or has a highly structured Jacobian. In MATLAB, linear algebra and the built-in array operations are relatively fast, and the language provides for sparse arrays. MATLAB handles storage dynamically and retains copies of arrays.

A new family of formulas for the solution of stiff problems called the numerical differentiation formulas, NDF's, are devised in §2. They are more efficient than the backward differentiation formulas, BDF's, though a couple of the higher order formulas are somewhat less stable. These formulas are conveniently implemented with backward differences. A way of changing step size in this representation is developed that is both compact and efficient in MATLAB. In §3 we devise a new linearly implicit one-step method for solving stiff systems, specifically a modified Rosenbrock method, and also a continuous extension of the method. §4 describes briefly how to modify these methods so as to solve conveniently and efficiently problems involving a mass matrix. In §5, we discuss briefly three methods for non-stiff problems. The two based on explicit Runge-Kutta methods are more efficient than those previously available in MATLAB and have free interpolants.

MATLAB has features, some of which are available in C and FORTRAN 90, that make possible an interesting and powerful user interface. §6 explains how the language was exploited to devise an interface that is unobtrusive, powerful, and extendable. Using the scheme of §7 for the numerical approximation of Jacobians, the design makes it possible for all the codes of the suite to be used in *exactly* the same manner. Options

* Mathematics Department, Southern Methodist University, Dallas, TX 75275. (shampine@na-net.ornl.gov)

† The MathWorks, Inc., 24 Prime Park Way, Natick, MA 01760. (mwr@mathworks.com)

allow users to supply more information that makes the solution of stiff problems more reliable and/or efficient. In particular, it is easy to exploit sparse Jacobians.

Examples in §8 show the relative performance of codes in the suite and the value of some of the options. The availability of the codes is described in §9.

2. Implicit Formulas for Stiff Systems. The BDF's are very popular for solving stiff problems. When the step size is a constant h and backward differences are used, the formula of order k, BDFk, for a step from (t_n, y_n) to (t_{n+1}, y_{n+1}) is

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

The algebraic equation for y_{n+1} is solved with a simplified Newton (chord) iteration. The iteration is started with the predicted value

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

The leading term of the BDFk truncation error can be conveniently approximated as

$$\frac{1}{k+1} h^{k+1} y^{(k+1)} \approx \frac{1}{k+1} \nabla^{k+1} y_{n+1}. \tag{3}$$

The typical implementation of a general-purpose BDF code is quasi-constant step size. This means that the formulas used are those for a constant step size and the step size is held constant during an integration unless there is good reason to change it. General-purpose BDF codes also vary the order during an integration.

2.1. The Numerical Differentiation Formulas. Noting that the predictor (2) has a longer memory than (1), Klopfenstein [25] and Reiher [31] considered how to exploit this to obtain better stability. Klopfenstein studied methods of the form

$$\sum_{m=1}^{k} \frac{1}{m} \nabla^m y_{n+1} - hF(t_{n+1}, y_{n+1}) - \kappa\gamma_k \left(y_{n+1} - y_{n+1}^{(0)} \right) = 0 \tag{4}$$

that he called numerical differentiation formulas, NDF's. Here κ is a scalar parameter and the coefficients γ_k are given by $\gamma_k = \sum_{j=1}^{k} \frac{1}{j}$. The role of the term added to BDFk is illuminated by the identity

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

and the approximation (3) 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 of its truncation error is

$$\left(\kappa\gamma_k + \frac{1}{k+1} \right) h^{k+1} y^{(k+1)} \tag{5}$$

For orders 3-6, Klopfenstein and Reiher found numerically the κ that maximizes the angle of $A(\alpha)$-stability. 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. The optimal choice is $\kappa = -1/9$, yielding a truncation error

TABLE 1
The Klopfenstein-Shampine NDF's and their efficiency and $A(\alpha)$-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%

coefficient half that of BDF2. This implies that for sufficiently small step sizes, NDF2 can achieve the same accuracy as BDF2 with a step size about 26% bigger.

The formulas derived by Klopfenstein and Reiher at orders higher than 2 are less successful because the price of improved stability is reduced efficiency. Taking the opposite tack, we sought values of κ that would make the NDF's more accurate than the BDF's and not much less stable. Of course the leading term of the truncation error cannot be made too small, else it would not dominate and the formula would not behave as expected at realistic step sizes. Because Klopfenstein's second order formula optimally improves accuracy while retaining L-stability, it serves as the order 2 method of our NDF family. Correspondingly, we sought to obtain the same improvement in efficiency (26%) at orders 3-5. This comes at the price of reduced stability and we were not willing to reduce the stability angle by more than 10%. The search was carried out numerically. Our choices and the compromises made in balancing efficiency and stability are shown in Table 1. The stability of BDF5 is so poor that we were not willing to reduce it at all.

The first-order formula NDF1 has the form

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

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. The root-locus method obtains the boundary as a subset of $z = \rho(\theta)/\sigma(\theta)$ as $\theta = \exp(i\psi)$ ranges over all numbers of magnitude 1. For NDF1 it is found that

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

and that a sufficient condition for the formula to be A-stable is $1 - 2\kappa \geq 0$. As at other orders, we chose an improvement in efficiency of 26%, leading to $\kappa = -0.1850$.

2.2. Changing the Step Size. Backward differences are very suitable for implementing the NDF's in MATLAB because the basic algorithms can be coded compactly and efficiently. We develop here a way of changing step size that is also well-suited to the language.

When the integration reaches 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 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 two representations of $P(t)$ leads to the identity

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

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

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

which is in matrix terms $D^* U = DR$. The entries of the $k \times k$ matrix U are

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

Matrix U satisfies $U^2 = I$. This implies that $D^* = D\,(RU)$, the scheme we use for changing the step size. The entries of U are integers that do not depend on h nor on k. In terms of $\rho = h^*/h \neq 1$, the entries of the $k \times k$ matrix R are

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

R must be formed each time the step size is changed. This is done in a single line of MATLAB code using the cumprod function. Likewise, changing the representation by means of matrix multiplication is done in a single line. Accordingly, in MATLAB this way of changing the step size is both compact and efficient.

2.3. The `ode15s` Program. The code `ode15s` is a quasi-constant step size implementation of the NDF's in terms of backward differences. Options allow integration with the BDF's and integration with a maximum order less than the default of 5.

The identity

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

shows that equation (4) is equivalent to

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

In the simplified Newton iteration, 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}. Scaling the equation to remove the scalar multiplying J offers some advantages [36]. It is much more accurate to obtain the fundamental quantity $\nabla^{k+1} y_{n+1}$ as the limit of

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

computed from

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

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.

In the context of MATLAB it is natural to retain a copy of the Jacobian matrix. Of the codes cited, only VODE exploits this possibility. It is also natural to form and factor the iteration matrix every time the step size or order is changed. The rate of convergence is monitored [34] and the iteration 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. Otherwise the step size is reduced.

Our scheme for reusing Jacobians means that when the Jacobian is constant, `ode15s` will normally form a Jacobian just once in the whole integration. Also, the code will form very few Jacobians when applied to a problem that is not stiff. `ode15s` competes rather well with the codes for non-stiff problems because of this and the efficient linear algebra of MATLAB.

3. Linearly Implicit Formulas for Stiff Systems. In this section it is convenient at first to consider differential 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

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

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

A number of authors have explored formulas of this form with J that only approximate $\partial F(y_n)/\partial y$. A second order formula due to Wolfbrandt [40] is

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

Here the parameter $d = 1/\left(2+\sqrt{2}\right)$. It is characteristic of such W-formulas that the order of the formula does not depend on J. The stability does depend on J, and if $J = \partial F/\partial y$, the formula is L-stable.

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] achieved a one-step error estimate without additional evaluations of F by assuming that

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

We must assume that $J \approx \partial F/\partial y$ 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 describe formulas based on Scraton's assumption as modified Rosenbrock formulas. Scraton's error estimate is inefficient for very stiff problems because the companion formula of order 3 is not stable at infinity. Zedan [43], [42] derived a companion that is A-stable and also requires no additional evaluations of F.

3.1. A Modified Rosenbrock Triple. In a step from t_n to t_{n+1} with Wolfbrandt's formula and any of the error estimates cited, F is evaluated only at t_n and $t_n + 2h/3$. It is possible that if a solution component changes sharply somewhere in $(t_n + 2h/3, t_n + h)$, a poor approximation to this component at t_{n+1} will be produced and accepted. Very sharp changes in a solution component are common when solving stiff problems and robust software ought to be able to deal with them. For this reason, many authors attempt to derive pairs that evaluate at both ends of a step.

Wolfbrandt's formula is a member of a family of second order, L-stable W methods [40] that involve two evaluations of F. By making the *first* evaluation of F for the next step the *same as* the *last* of the current step (FSAL), we have at our disposal a function evaluation that is usually "free" because most steps are successful. Exploiting this we derived a pair of formulas that we present in a form that avoids unnecessary matrix multiplications [36]. When advancing a step from (t_n, y_n) to $t_{n+1} = t_n + h$, the modified Rosenbrock pair is

$$\begin{aligned} F_0 &= F\left(t_n, y_n\right) \\ k_1 &= W^{-1}\left(F_0 + hdT\right) \\ F_1 &= F\left(t_n + 0.5h, y_n + 0.5hk_1\right) \\ k_2 &= W^{-1}\left(F_1 - k_1\right) + k_1 \\ y_{n+1} &= y_n + hk_2 \\ F_2 &= F\left(t_{n+1}, y_{n+1}\right) \end{aligned}$$

$$k_3 = W^{-1}\left[F_2 - e_{32}\left(k_2 - F_1\right) - 2\left(k_1 - F_0\right) + hdT\right]$$
$$error \approx \frac{h}{6}\left(k_1 - 2k_2 + k_3\right)$$

Here $W = I - hdJ$ with $d = 1/\left(2+\sqrt{2}\right)$ and $e_{32} = 6 + \sqrt{2}$,

$$J \approx \frac{\partial F}{\partial y}\left(t_n, y_n\right) \quad \text{and} \quad T \approx \frac{\partial F}{\partial t}\left(t_n, y_n\right).$$

If the step is a success, the F_2 of the current step is the F_0 of the next. If $J = \partial F/\partial y$, the second order formula is L-stable. Because the pair samples at both ends of the step, it has a chance of recognizing very sharp changes that occur within the span of a step.

On p. 129 of his dissertation, Zedan describes briefly a scheme for interpolating his modified Rosenbrock method. On $[t_n, t_n + h]$ he approximates the solution with the quadratic polynomial interpolating y_n, y_{n+1}, and $F_0 = F\left(t_n, y_n\right)$. Interpolating F_0 can be unsatisfactory when the problem is very stiff. A standard way to obtain a continous extension to a one-step method is to derive a family of formulas, each of which provides an approximation at $t_n + h^*$ for a given $h^* = sh$. For any s, it is easy to derive a formula that has the same form as that of the basic step to t_{n+1}; the trick is to reuse the computations of the basic step as much as possible. In the new formula 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 basic step itself. Specifically, it is found that a second order approximation to $y\left(t_n + sh\right)$ is provided by

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

Though derived as a family of formulas depending on a parameter s, the continuous extension turns out to be a quadratic polynomial in s. It interpolates both y_n and y_{n+1}, hence connects continuously with approximations on adjacent steps. This interpolant behaves better for stiff problems than the one depending on F_0, in essence because of the W^{-1} implicit in the k_i.

3.2. The `ode23s` Program. The code `ode23s` based on the scheme derived here provides an alternative to `ode15s` for the solution of stiff problems. It 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, which is relatively fast in MATLAB. The integration is advanced with the lower order formula, so `ode23s` does not do local extrapolation. To achieve the same L-stability in `ode15s`, the maximum order would have to be restricted to 2.

We have considered algorithms along the lines of [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 the current version of `ode23s` forms a new Jacobian at every step for several reasons. 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. In MATLAB the Jacobian is typically of modest size or sparse and its evaluation is not very expensive compared with evaluating F. Lastly, evaluating the Jacobian at every step enhances the reliability and robustness of the code.

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.

4. Stiff Systems of More General Form. A stiff problem $M(t)\,y' = f(t,y)$ with a non-singular mass matrix $M(t)$ can always be solved by transforming the equation to the equivalent form $y' = F(t,y) = M^{-1}(t)\,f(t,y)$. Small modifications to the methods avoid the inconvenience and expense of this transformation.

Because the modified Rosenbrock method was derived for autonomous systems, it is awkward to accomodate matrices M that depend on t. Accordingly, we allow only constant mass matrices in `ode23s`. The modification is derived by applying the method to the transformed equation and then rewriting the computations in terms of M and $f(t,y)$. The first stage k_1 is obtained from

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

Here $F_0 = M^{-1}f(t_0, y_0) = M^{-1}f_0$,

$$J \approx \frac{\partial F}{\partial y} = M^{-1}\frac{\partial f}{\partial y} \quad \text{and} \quad T \approx \frac{\partial F}{\partial t} = M^{-1}\frac{\partial f}{\partial t}.$$

Scaling the equation for k_1 by M leads to

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

The remaining computations are treated similarly. The usual form of the method is recovered when $M = I$. This modification of the method allows the user to pose the problem in terms of M and $f(t,y)$. It avoids the solution of linear equations that arise when the equation is transformed.

The `ode15s` code allows the mass matrix to depend on t. This causes only one difference in the modification of the methods of this code. The simplified Newton method involves solution of 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 $m \le n$. Because M depends on t, it is not possible to remove all the inverses simply by scaling with $M(t_{n+1})$. We approximate the scaled iteration matrix by

$$M(t_m) - \frac{h}{(1-\kappa)\gamma_k}J \quad \text{where} \quad J \approx \frac{\partial f}{\partial y}(t_m, y_m).$$

With this approximation, the computations can be written in terms of $M(t)$ and $f(t,y)$. The modified method reduces to the usual one when $M(t) = I$.

5. Explicit Formulas for Non-Stiff Systems. The two explicit Runge-Kutta codes, `ode23` and `ode45`, in previous versions of MATLAB have been replaced by codes with the same names that remedy some deficiencies in design and take advantage of developments in the theory and practice of Runge-Kutta methods. A new code, `ode113`, is a PECE implementation of Adams-Bashforth-Moulton methods.

The new `ode23` is based on the Bogacki-Shampine $(2, 3)$ pair [3] (see also [37]) and the new `ode45` is based on the Dormand-Prince $(4, 5)$ pair [12]. Workers in the field employ a number of quantitative measures for evaluating the quality of a pair of formulas. In the standard measures these pairs are of high quality and significantly more efficient than those used in the earlier codes. Both pairs are FSAL and constructed for local extrapolation.

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 of the solution for plotting. This phenomenon is exacerbated when plotting in the phase plane. A good way to deal with this is to form additional values by means of a continuous extension (interpolant). A continuous extension also makes possible event location, a valuable capability in ODE codes. We selected pairs for which continuous extensions were available. In the case of the $(2, 3)$ pair, accurate solution values can be obtained for "free" (no additional evaluations of F) by cubic Hermite interpolation to the values and slopes computed at the ends of the step. Dormand and Prince obtained a number of inexpensive interpolants for their pair in [13]; they communicated to us another interpolant of order 4 that is of high quality and "free". The default in `ode45` is to use this interpolant to compute solution values at four points spaced evenly within the span of each natural step. Figure 1 shows the importance of this development. We return to this issue in §6.

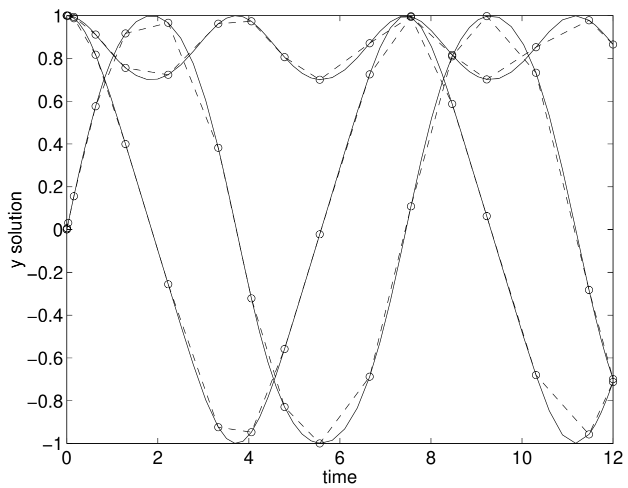

FIG. 1. *The* `rigid` *solution as computed by* `ode45`*. The continuous curves result from the default output of* `ode45`*. The discrete values are the results at the end of each step. The dashed curves show what happens when the new interpolation capability is disabled.*

`ode113` is a descendant of ODE/STEP, INTRP [39]. Though the code differs considerably in detail, its basic algorithms follow closely those of STEP. In particular, it does local extrapolation. The authors of ODE/STEP, INTRP tried to obtain as cheaply and reliably as possible solutions of moderate to high accuracy to problems involving functions F that are expensive to evaluate. 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.

6. 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 typical problems. It is not easy to reconcile these goals. 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 make use of Jacobians. If a code for solving stiff problems is to "look" like a code for non-stiff problems, it is necessary to approximate these Jacobians inside the code. Unfortunately, it is difficult to do this 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. Indeed, we have already extended the functionality of our original design three times.

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

```
[t,y] = ode15s('ydot',tspan,y0);
```

The results can then be displayed with the usual plotting tools of MATLAB, e.g., by `plot(t,y)`. Here `ydot` is the name of a function that defines the differential equation. Figure 2 shows an example of the van der Pol equation coded as function `vdpnsode`. The interval of integration is `tspan=[t0,tfinal]` and the initial conditions are `y0`. The code obtains the number of equations by measuring the length of the vector `y0`.

```
function dy = vdpnsode(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* `vdpnsode`.

The suite exploits the possibility of a variable number of arguments. For instance, the codes monitor several measures of cost, such as the number of steps and the number of Jacobian evaluations, and return them in an optional output argument, viz. `[t,y,stats]`. It also exploits the possibility of empty arrays. For example, it is possible to define the initial value problem in one file. Figure 3 illustrates this for the CHM6 [14] stiff test problem of [28]. With the function coded as shown, if `ode15s` is invoked with empty or missing arguments for `tspan` and `y0`, it will call `chm6ode` with an empty argument for `t` to obtain the information not supplied in the call list.

Returning the independent variable and approximate solution at each step in arrays `[t,y]` is natural in MATLAB because it facilitates plotting and the sizes of

```
function [out1,out2,out3] = chm6ode(t,y)
if isempty(t)             % 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* `chm6ode`.

output arrays do not have to be specified in advance. The steps chosen by a code tend to cluster where solution components change rapidly, so this design generally results in satisfactory plots. However, this is not always the case, a point made by Polking [30] for the old `ode45` and illustrated by Figure 1 for the new. All the methods implemented in the suite have free interpolants that can be evaluated at additional points to obtain a smooth plot. In our design there is an option `refine` for specifying the number of answers to be computed at points equally spaced in the span of each step. By increasing `refine`, it is always possible to get a smooth plot. The additional output is obtained inexpensively via a continuous extension of the formula.

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. If `tspan` contains more than two entries, the code is to produce output at the values of the independent variable specified in `tspan` and only these values. Because output is obtained by evaluating a polynomial, the number and placement of specified output points has little effect on the cost of the integration.

It is difficult to accomodate all the possibilities for optional input 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. The options vector is optional. When it is employed, the syntax of a call to the solver is

```
[t,y] = ode15s('ydot',tspan,y0,options);
```

The vector `options` is built by means of the function `odeset` that accepts name-value pairs. We make use of keywords and exploit the fact that in MATLAB we can specify values for the options that are of different data types. Indeed, an option can have more than one data type as a value. `odeset` allows options to be set in any order and default values are used for any quantity not explicitly set by the user. A number of things are done to make the interface more convenient, e.g., the name alone instructs `odeset` to assign the value "true" to a Boolean variable.

The most commonly used options are `rtol` and `atol`, tolerances associated with the error control. Specification of the error control is a difficult matter discussed in

[37]. There it is explained how the simplifying assumptions made in the old ode23 and ode45 can lead to unsatisfactory results. Polking [30] makes the same point. Unfortunately, there seems to be no way to provide automatically a default control of the error that is completely satisfactory.

In the suite, the local error e_i in y_i is estimated in each step and made to satisfy

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

where r = rtol and a_i = atol(i). The scalar relative error tolerance rtol has a default value of 10^{-3}. The vector of absolute error tolerances atol has by default all its values equal to 10^{-6}. If a scalar absolute error tolerance is input, the code understands that the value is to be assigned to all entries of the atol vector.

As another example of setting options, suppose that we wish to solve a stiff problem with a constant Jacobian, an absolute error tolerance of 10^{-20} is 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 is done by

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

An illuminating example is provided by the chm6ode function shown in Figure 3. Its solution is discussed in [27]. Figure 5 is a log-log plot of $y_2(t)$. A fundamental difficulty is that with an initial value of 0, there is no natural measure of scale for $y_2(t)$. It turns out that the component never gets bigger than about 7×10^{-10}, so the default absolute error tolerance of 10^{-6} is inappropriate. After an integration that revealed the general size of this solution component, we solved the problem again with the default relative tolerance of 10^{-3} and an optional absolute error tolerance of 10^{-13}. This is accomplished by ode15s in only 139 steps. The step sizes ranged from 5×10^{-14} to 10^2! This is mainly due to $y_2(t)$ changing rapidly on a very short time scale. 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 in MATLAB, 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. Users are given the option of providing a function for the analytical evaluation of the Jacobian. They are also given the option of specifying that the Jacobian is constant, a special case that leads to significant savings in ode23s. The default is to treat the Jacobian as a full matrix. To take advantage of a sparse Jacobian, the code 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 (sparse) matrix S of zeros and ones that represents the sparsity pattern of the Jacobian. There are a number of ways to create matrix S. If there are neq equations, an neq $\times$ neq sparse matrix of zeros can be 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 set to 1, i.e. 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. For example, if the Jacobian is banded with bandwidth $2m + 1$, it can be defined in a single line

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

After defining the sparsity pattern, the value S is assigned to the option `sparseJ` with `odeset`. No further action is required of the user.

As discussed in §4, 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 $M(t)$. In other computing environments a mass matrix raises awkward questions about specification of M and its structure and how its structure relates to that of the Jacobian of $f(t,y)$. 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 $M(t)$. Or, if the mass matrix is constant, the matrix itself can be provided as the value of the option. The language deals automatically with the structures of the matrices that arise in the specification and solution of the problem [19].

7. Numerical Partial Derivatives. Methods for the solution of stiff problems involve partial derivatives of the function defining the differential equation. The popular codes allow users to provide a routine for evaluating these derivatives analytically. However, this is so much trouble for users and so prone to error that the default is to approximate them internally by numerical differentiation. The new suite follows this approach, using a function `numjac` to compute the numerical approximation.

The scaling difficulties that are possible when approximating partial derivatives by differences are well-known [37]. The `numjac` code implements a scheme of D. E. Salane [32] that takes advantage of experience gained at one step to select good increments for difference quotients at the next step. If it appears that a column might consist mainly of roundoff, the increment is adjusted and the column recomputed.

The solvers invoke `numjac` by

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

where the argument `'F'` is a string naming the function that defines the differential equation and `Fty` is the result of `'F'` evaluated at the current point in the integration `(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.

One aspect of the formation of partial derivatives special to the suite is the Boolean option `vectorized`. It is generally easy to code `'F'` so that it can return an array of function values. A vectorized version of the van der Pol example is shown in Figure 4. If `'F'` is coded so that `F(t,[y1 y2 ...])` returns `[F(t,y1) F(t,y2) ...]` and `vectorized` is set true, `numjac` will approximate all columns of the Jacobian with a single call to `'F'`. This avoids the overhead of repeatedly calling the function and it may reduce the cost of the evaluations themselves.

```
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.*

Another special aspect of `numjac` is that it computes sparse Jacobians as well as full ones. The structure of the Jacobian is provided by means of a sparse matrix `S` of zeros and ones. The first time that a solver calls `numjac`, the function finds 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`. Two schemes are tried (first-fit and first-fit after reverse column minimum-degree ordering [19]) and the more efficient grouping is adopted. This may not result in an optimal grouping because finding the smallest packing is an NP-complete problem equivalent to K-coloring a graph [9].

The modified Rosenbrock code requires the partial derivative `dFdt` every time it requires `dFdy`. 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 that it is done in the solver itself.

8. Examples. The first question a user must ask is, which of the five codes in the suite should I use and which, if any, of the options? By design it is very easy simply to try the most promising codes, but it is useful to have some insight. The method implemented suggests circumstances in which a code might be particularly efficient or not. This section presents some experiments with test problems from classic collections that illustrate the performance of the solvers and the effects of using certain options. We report how much it costs to solve a problem for a given tolerance, usually the default tolerance. This is ordinarily what users of MATLAB want to know. A discussion of some of the issues that arise in comparing solvers is found in [37] where this approach is called the *first measure of efficiency*. Implicit in the use of this measure is the assumption that for routine problems, the codes compared will produce solutions with accuracy at least comparable to the tolerance. No credit is given for producing solutions with much more accuracy than requested. Because the solvers control local errors, the true, or global, error can be sensitive to the method and the details of its implementation. We have tuned the step size selection algorithms so that for a wide range of test problems, consistent global accuracies comparable to the tolerance are delivered by the various solvers applied to the same problem.

From our experience with writing solvers in other computing environments, we believe that our implementations of the methods are comparable in quality to popular codes based on the same or related methods. Naturally we tested this by comparing the new Runge-Kutta codes to the old MATLAB ones. Also, we report here tests showing that the NDF code is comparable to a popular BDF code written in FORTRAN.

The experiments reported here and others we have made suggest that except in special circumstances, `ode45` should be the code tried first. If there is reason to believe the problem to be stiff, or if the problem turns out to be unexpectedly difficult for `ode45`, the `ode15s` code should be tried. When solving stiff problems, it is important to keep in mind the options that improve the efficiency of forming Jacobians.

8.1. Stiff Examples. In MATLAB it is advantageous to vectorize computations whenever possible. Accordingly, all stiff problems were coded to use the `vectorized` option when computing numerical Jacobians. Also, the advantages of the `constantJ` option are so obvious that we used it when solving the stiff problems with constant Jacobians, namely `a2ex`, `a3ex`, `b5ex`, and `hb3ex`.

The `ode15s` code was developed for the NDF's. It is such an easy matter to provide for the BDF's in this code that they were allowed as an option. Some experiments show the consequences of exercising this option. Table 2 gives 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 the default NDF's result in fewer steps than 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).

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

To verify that the performance of `ode15s` is comparable to that of a modern BDF code, we have compared `ode15s` using the NDF's to DDRIV2 [24] on some relatively difficult problems. 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`.

It is not possible to compare DDRIV2 and `ode15s` in detail because they cannot be used to solve exactly the same computational problem. For one thing, the error controls are different. DDRIV2 uses a root-mean-square norm to measure the error in a solution component relative to the larger of the magnitude of the component and a threshold. We made 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. In addition 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 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.

Table 3 compares DDRIV2 to `ode15s` using the NDF's. We interpret these comparisons as showing that `ode15s` is an effective code for the solution of stiff problems. DDRIV2 does not save Jacobians and the numerical results indicate that to a degree `ode15s` is trading linear algebra for a smaller number of approximate Jacobians. This is appropriate in MATLAB, but because these examples involve just a few equations, the benefits of reusing Jacobians are masked.

The chemical reaction problem `chm6ex` is given in §6. A plot of one component is displayed in Figure 5. `ode15s` is able to solve the problem effectively using a remarkably small number of Jacobians. Problem `chm9ex` is a scaled version of the Belousov oscillating chemical reaction [14]. A discussion of this problem and plots are

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. Note that the integration parameters of* `ode15s` *were changed from their default values.*

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

found in [1], p. 49 ff. The limit solution is periodic and exhibits regions of very sharp change. We chose the interval $[0, 650]$ so as to include two complete periods and part of another. `ode15s` is trading some linear algebra for a reduction in the number of Jacobians. Because there are only three solution components, reducing the number of Jacobians does not have much effect on the number of function evaluations.

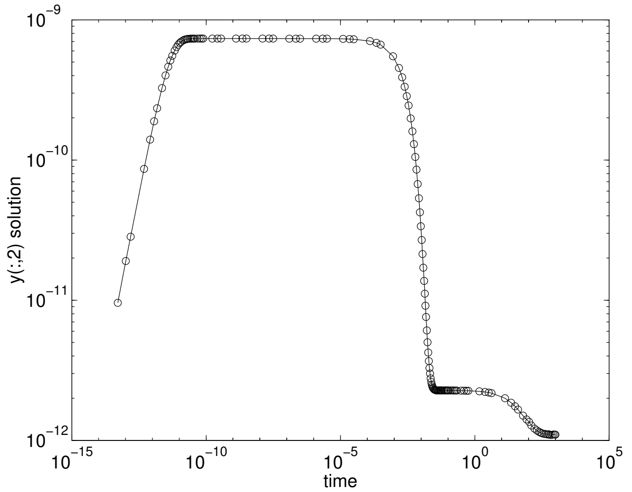

FIG. 5. *A log-log plot of the second component of the solution of* `chm6ex`.

Hindmarsh and Byrne [23] present the non-autonomous problem coded in `hb2ex` that arises in a diurnal kinetics model and they discuss its solution with EPISODE. 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. As the measures of cost given in Table 3 show, `ode15s` performs quite well in comparison to a good BDF code. For this problem, reuse of Jacobians proved to be quite advantageous. With only two equations, numerical evaluation of a Jacobian is so cheap that the difference in cost cannot be due entirely to saving Jacobians.

Example `vdpex` is 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, forming Jacobians more often would probably have been a bargain in `ode15s`.

Now we compare the modified Rosenbrock code `ode23s` to `ode15s`. Table 4 provides statistics for two stiff problems. Note that `ode15s` was run with both the NDF's and the BDF's. The fact that NDF1 and NDF2 are more efficient than BDF1 and BDF2 is 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)	32.58	1165	124	2586	1	319	2578
	`ode15s` (NDF)	25.95	936	97	2074	1	263	2066
	`ode23s`	15.38	549	17	1689	1	566	1698

As a first example we solved `vdpex` with a relative accuracy tolerance of 1.0%. At this tolerance `ode23s` was faster than `ode15s`, even though it made many more Jacobian evaluations, and the plot of $y(t)$ in Figure 6 obtained with `ode23s` is notably better than that of `ode15s`. However, when the tolerance is tightened to the default of 0.1%, `ode15s` is faster than `ode23s` and the plot of $y(t)$ is just as good.

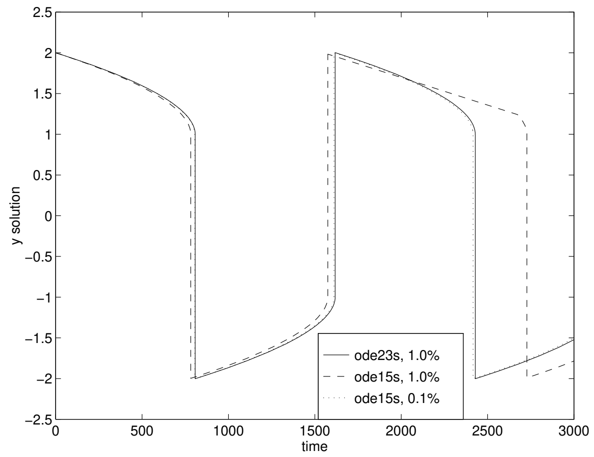

FIG. 6. *The* `vdpex` *solution computed by* `ode23s` *and* `ode15s` *at 1.0% accuracy, and by* `ode15s` *at 0.1% accuracy. The latter indicates that the 1.0% solution by* `ode15s` *is less accurate than* `ode23s`*.*

Problem `b5ex` [15] has a Jacobian with eigenvalues near the imaginary axis. The popular variable-order BDF codes do not deal with this well. Gaffney [16] compares BDF and other kinds of codes on this and similar problems. `ode15s` recognizes automatically that this problem has a constant Jacobian. For a fair comparison, we used the option of telling both codes that the Jacobian is constant. We also restricted the maximum order of the NDF's used by `ode15s` to 2, so that both codes would be L-stable. Evidently `ode23s` is to be preferred for the solution of this problem.

Supplying a function for evaluating the Jacobian can be quite advantageous, both with respect to reliability and cost. Table 5 shows the effects of using the `analyticJ` option for some of the most expensive of our examples. Because `ode23s` evaluates the Jacobian at every step, reducing this cost by means of an analytical Jacobian has an important effect on the overall cost. It 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	23.37	9.70
`chm9ex`	47.12	27.32	29.38	25.86
`hb2ex`	140.78	87.29	13.45	13.40
`vdpex`	28.61	16.36	19.33	19.21

Next, we examine the role of Jacobian sparsity. The `brussex` example is the classic "Brusselator" system modelling diffusion in a chemical reaction [21],

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

and is solved on the time interval $[0, 10]$ with $\alpha = 1/50$ and

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

There are $2N$ equations in this system, but the Jacobian is banded with a constant width 5, if the equations are ordered as $u_1, v_1, u_2, v_2, \ldots$.

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 the results for `ode23s` and `ode15s` were produced using the default numerical approximation of Jacobians. As the second columns show, the `sparseJ` option makes a tremendous difference. Until N becomes large, `ode15s` is efficient even without the `sparseJ` option because it forms relatively few Jacobians.

The `fem2ex` example involves a mass matrix. The system of ODE's, found in [41], 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 $u(t, 0) = u(t, \pi) = 0$. 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

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

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	80.00	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

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 ODE's

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

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. The problem is solved on the time interval $[0, \pi]$.

Because the mass matrix A depends on t, this equation cannot be solved directly with `ode23s`. However, scaling 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 is sparse, but in this instance we have followed [41] in taking $N = 9$, which is too small to take advantage of the sparsity. 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

8.2. Non-stiff Examples. In this section we consider four non-stiff examples drawn from the collections [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. `r3body` describes a periodic orbit for a restricted three body problem [39]. Because the problems are non-stiff, they can be solved with all the MATLAB ODE solvers. Table 8 compares the costs of solution.

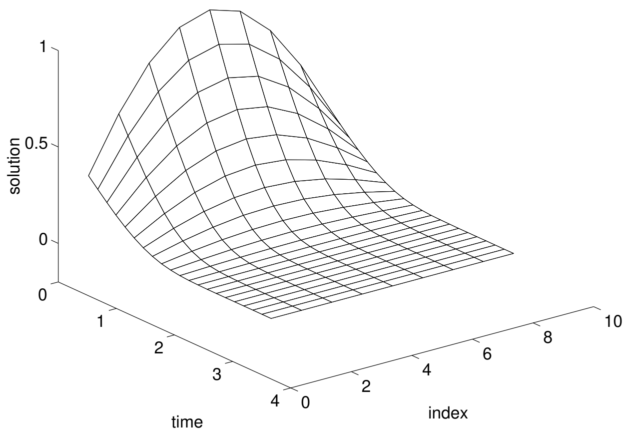

FIG. 7. *The* `fem2ex` *solution with* $N = 9$ *as computed by* `ode15s`.

9. Conclusion. 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.

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.

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.

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

[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.