17

Developing ODE software in new computing environments

L. F. Shampine
Southern Methodist University
Mathematics Department, Dallas, TX 75275, USA.
Email: shampine@na-net.ornl.gov

M. W. Reichelt
The MathWorks Inc.
24 Prime Park Way, Natick, MA 01760, USA.
Email: mwr@mathworks.com

Abstract

The MATLAB ODE Suite is used as a case study for the development of ODE software in a new computing environment. Some current work on a multimedia project for teaching ODEs is used to illustrate another environment.

Keywords

ODE software, MATLAB, simulation language, teaching package, user interface

1 INTRODUCTION

The fruits of decades of research and development of mathematical software for solving initial value problems for ordinary differential equations, ODEs, on mainframe computers are diffusing into new computing environments. Serious scientific computation has been made possible in several new environments because of the remarkable increase of computing power. Most of our attention will be directed to the solution of ODEs on microcomputers and workstations, specifically in the widely used scientific computing environment MATLAB. There are other environments that are similar, so we regard MATLAB as representative. Naturally the details of this particular environment were important in the development of the MATLAB ODE Suite (Shampine and Reichelt, 1997) that serves here as a case study.

R. F. Boisvert (ed.), *Quality of Numerical Software*

Now that significant computing power has become readily available to students, there have been many efforts to supplement traditional courses in ODEs with computational experiments. These efforts have resulted in a number of packages for the solution of ODEs. One of us is associated with an effort by C·ODE·E, the Consortium for Ordinary Differential Equations Experiments, to provide a multimedia supplement of this kind. A few observations about this project illustrate some of the different turns that software development has taken in another computing environment. Although we shall not pursue the matter here, we must observe that hand-held calculators are now sufficiently powerful to be used for solving ODEs. Though similar to the teaching environment, there are important differences that make it a distinct computing environment.

2 MATLAB

2.1 The Language

Certain language issues that are troublesome when developing software for general scientific computation are not present, or present in rather different form, when developing software in MATLAB. For one thing, there is only one source of the language, so there is no question about adhering to a standard. For another, there is only one precision, with details being specific to the particular machine used. MATLAB is a modern programming language with adequate constructs for scientific software plus some nice features that we have exploited in the ODE suite.

There is a standard MATLAB language, but it evolves, and at a pace extraordinary compared to the major languages for scientific computing. For example, the preliminary release of the Suite via `http://www.mathworks.com/tmwcontsoft.html` makes use of language features not available in versions prior to 4.2. MATLAB 5 is a richer language, and because we have used some of its new features in the formal release of the Suite as part of this version of MATLAB some capabilities are not backward compatible.

2.2 Efficiency

Certain aspects of the environment had a profound effect on the development of the Suite. One is the issue of efficiency. Naturally it is hoped to make programs as efficient as possible, but efficiency is not one of the *primary* goals of the environment. In the first instance programs are interpreted, as opposed to compiled. As a consequence, it is *roughly* true that each line of code executes in the same time. There are many built-in functions supplied in compiled form, and exploiting them is important to efficiency. In particular, the language is oriented towards matrix computation with important possibilities for vectorization.

A family of numerical differentiation formulas, NDFs, related to the BDFs popularized by Gear was developed for the solution of stiff problems in the Suite. The formulas are written in a backward difference representation and a quasi-constant step size implementation is used. A new method of changing the step size in a backward difference representation was developed that is compact and efficient in MATLAB. Details can be

found in (Shampine and Reichelt, 1997). We bring this up because a matrix that must be formed at each change of step size can be computed in a single line of code using a built-in function. The backward difference representation is then changed by two matrix multiplications in a single line of code. These aspects of the scheme make it efficient in this computing environment.

The numerical solution of stiff problems requires an approximate Jacobian. The default in the Suite is to approximate the Jacobian with differences. This involves evaluation of the function defining the differential equation at `t` and a number of arguments `y`. By taking advantage of MATLAB's built-in array operators, all these evaluations can be accomplished with a single call to a "vectorized" function. Generally it is easy to code the function in this manner because the built-in functions are vectorized and scalar arithmetic operations are made vector operations by placing a dot before the symbol. Vectorizing the function can reduce significantly the time it takes to solve a large stiff problem. As an illustration of vectorization, a scalar function for the van der Pol equation $y'' - 1000(1-y^2)y' + y = 0$ written as a first order system is

```
function dy = vdp(t,y)
dy(1) = y(2);
dy(2) = 1000*(1-y(1)^2)*y(2)-y(1);
dy = dy(:);                % Solver requires a column vector.
```

and a vectorized function is

```
function dy = vdpvec(t,y)
dy(1,:) = y(2,:);
dy(2,:) = 1000*(1-y(1,:).^2).*y(2,:)-y(1,:);
```

Notice that we do not have to know how many arguments are to be supplied by the solver in the form of columns of an array `y`. Indeed, in some circumstances the number is determined by the solver at run time.

2.3 Storage

Storage is handled in a very convenient way in MATLAB that obviates the complicated declarations and allocations seen in general scientific computing, simplifying *considerably* the use of solvers and reducing the time required to develop the software. For instance, it is not necessary to tell a solver the number of equations or the sizes of arrays because these numbers are available by means of intrinsic functions when actually needed. The preceding example illustrates how much easier this can be for users when coding even a very simple problem. A matter of considerable significance is that sparse matrix technology has been fully integrated into MATLAB.

The natural mode of output in this environment is to report the vector of solution components corresponding to each step of the integration. Because it is not necessary to declare the sizes of arrays in advance, this is accomplished easily. To be sure, an *efficient* implementation preallocates storage in blocks so as to reduce the overhead of continually

increasing the size of the array, but this is done out of sight of the user and does not complicate the codes greatly.

When solving stiff problems there is considerable advantage to be gained from storing approximate Jacobians and reusing them. With the virtual memory of the MATLAB environment, this strategy was easy to implement in the stiff solvers. Because it is generally not feasible to solve large stiff problems without accounting for the structure of the Jacobian matrix, the stiff solvers will use structural information if it is provided. This, too, was easy to implement because of the transparent way that sparse matrices are handled in MATLAB. Although there are two distinct types of matrices, dense and sparse, after declaring a matrix to be sparse, all further manipulation of the matrix is the same as for a dense matrix. In particular, the function for computing an LU decomposition recognizes automatically whether the matrix to be factored is dense or sparse, chooses an appropriate method, and returns the factors in appropriate form.

The solvers for stiff problems allow problems of more general form, namely

$$\boldsymbol{M}(t)\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$$

In other environments, storage issues become quite complicated for such problems. Typically the user must inform the solver whether the mass matrix $\boldsymbol{M}(t)$ is dense, banded, or sparse, and accordingly provide either the size of the array in which it is stored, the upper and lower bandwidths, or information about the data structure. Additionally, the user must answer the same questions about the Jacobian of $\mathbf{f}(t, \mathbf{y})$ and decide how storage is to be handled for the pair of matrices jointly. But all of this is *much* less complex in the MATLAB environment because the language sorts out many of these issues automatically.

2.4 User Interface

In the MATLAB context it is essential that solving ODEs be as easy as possible. By means of defaults, it possible to reduce the demands on the user merely to specifying what the problem *is*. To solve an initial value problem $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ for $t0 \leq t \leq tfinal$, with $\mathbf{y}(t0)$ given, the function $\mathbf{f}(t, \mathbf{y})$ is first defined in a file. Proceeding in this manner the full power of the language can be brought to bear and there is no restriction on how complicated the function can be. If the ODE file is called, say, `f.m`, then on defining vectors `tspan=[t0; tfinal]` and `y0` =$\mathbf{y}(t0)$, the initial value problem can be solved by typing

```
[t,y] = ode45('f',tspan,y0);
```

at the command line. Here the solver happens to be `ode45`, but by design, all five solvers in the Suite can be used in *exactly* the same way. Each row of the array `y` contains the vector of solution components corresponding to the time in the array `t`. For most of the solvers, the times `t` are those of the steps taken by the solver as it integrates from `t0` to `tfinal`. The solution is accurate to a default relative tolerance of 10^{-3} and a default absolute tolerance of 10^{-6} on each component. It is easy to extract solutions at times of interest and to manipulate the solution as a whole. Very often in this context users examine the solution by means of the standard plotting tools of MATLAB, e.g., plotting all solution components with `plot(t,y)`.

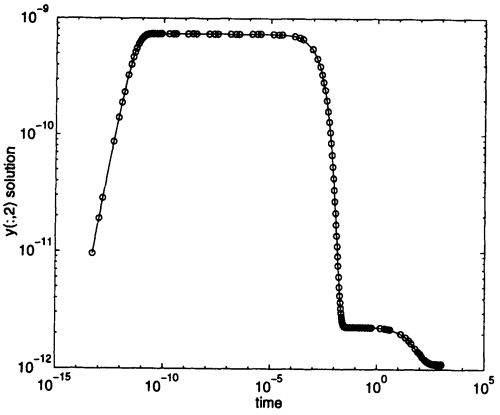

Figure 1 A log-log plot of the second component of the solution of CHM6.

A standard test problem in chemical kinetics called CHM6 (Enright and Hull, 1976) will serve to make a number of points. The equations are simple enough:

$$
\begin{aligned}
y_1' &= 1.3 * (y_3 - y_1) + 10400 * K * y_2 \\
y_2' &= 1880 * (y_4 - y_2 * (1 + K)) \\
y_3' &= 1752 - 269 * y_3 + 267 * y_1 \\
y_4' &= 0.1 + 320 * y_2 - 321 * y_4
\end{aligned}
$$

where $K = \exp(20.7 - 1500/y_1)$. They have almost this form when coded in a MATLAB ODE file. The system is to be integrated over `tspan = [0; 1000]` with initial condition `y0 = [761; 0; 600; 0.1]`. All we wish to do right now is point out that the solution components are scaled very differently and some reactions proceed very quickly. Figure 1 is a log-log plot of the second solution component. Its behavior cannot be discerned from a simple linear plot of all the components, so it is useful that the numerical solution can be manipulated and plotted in different ways to gain an understanding of the behavior of the solution without having to solve the problem repeatedly.

An interesting possibility peculiar to the MATLAB environment is to invoke a solver without specifying output arguments `[t,y]`. The solvers are coded to interpret this as an instruction to plot the solution as it is computed. This is natural for classroom purposes, but it can also be illuminating in scientific computation by drawing attention to points in the interval where the code has difficulty. For example, the integration slows to a crawl when a solver appropriate for non-stiff problems is used to integrate a problem that becomes stiff. This is a good indication that a stiff solver should be used!

The decision to make it possible to use all solvers in the same way implies that the two solvers in the Suite intended for stiff problems must by default approximate Jacobians numerically. This was done in part to make it easy to use the solvers, but in part it was done to make it easy to try a solver appropriate for stiff problems if one intended for non-stiff problems should prove unsatisfactory.

The CHM6 example illustrates why it is necessary to allow users to provide optional input. Default error tolerances are used to make life as easy as possible for the user, but they cannot always be appropriate, especially the absolute error tolerances. As Figure 1 shows, the second component of the solution of the CHM6 problem is never greater than about 7×10^{-10} — the default absolute error tolerance of 10^{-6} is much too big. Stiff problems like this one place special demands on the solvers. Providing additional information such as a function for evaluating the Jacobian analytically can increase substantially the reliability and efficiency of solution. The trick is to make it possible for a user to gain some control of the solution process or to assist the solver without the options being obtrusive when solving routine problems.

The solvers exploit the possibility of optional arguments in MATLAB to allow users to provide additional information. An options argument is created with a function `odeset` that uses keywords to make specification of options easy. Suppose, for example, that we integrate the CHM6 problem with equations defined in a file `chm6.m` and we realize that we ought to reduce the absolute error tolerance to 10^{-13}. Further, we decide to assist the solver by providing a function for evaluating the Jacobian analytically. We would first define

```
options = odeset('jacobian',1,'atol',1e-13);
```

and then invoke the solver with

```
[t,y] = ode15s('chm6',tspan,y0,options);
```

Here the value 1 for the keyword `Jacobian` is the value the language uses for "true". Setting this option true instructs the solver that the ODE file `chm6.m` is coded so that `chm6(t,y,'jacobian')` evaluates the Jacobian analytically. The `atol` option illustrates that arguments can have different data types. Supplying a scalar as the value of the keyword `atol` is interpreted to mean that this value is the tolerance for all components. Supplying a vector is interpreted as supplying tolerances applicable to corresponding components of the solution. As illustrated here, `odeset` exploits optional arguments to make it possible for users to specify only the options that are relevant to the problem at hand.

We have already mentioned that the solvers intended for stiff problems allow a more general form of the initial value problem involving a mass matrix. The solver is informed of this by means of the options argument and the mass matrix is evaluated in the file defining the problem in a way entirely analogous to the way analytical Jacobians are handled. The options interface makes this capability invisible to the person who does not need it.

Optional arguments are exploited for output as well as input. For instance, the solvers monitor the various costs of the computation such as the number of function evaluations and the number of LU decompositions. One way to access this information is by means of an additional output argument, `[t,y,stats]`. Another is to request that the options be printed by setting the `'stats'` option to 1 in the options argument.

A much more substantial use of both input and output is in event location. An event is said to occur at time t^* when the solution $\mathbf{y}(t)$ of the initial value problem is such that

$g(t^*, \mathbf{y}(t^*)) = 0$ for a given event function $g(t, \mathbf{y})$. A classic example of event location is the bouncing ball. A ball is thrown into the air and its height computed as a function of time. The solver must determine when the ball hits the ground. Note that an initial value problem might have many event functions associated with it. Like an analytical Jacobian or a mass matrix, the event functions are coded in the ODE file, and the solver is informed that there are event functions by means of the input options argument.

In addition to defining the event function equations, it is necessary to specify the character of each event. Some events are passive in the sense that the solver is merely to report when the event occurred and the value of the solution then. Other events are terminal, meaning that the integration must stop when, or if, the event occurs. For the bouncing ball example, hitting the ground is a terminal event. Another characteristic is the directionality of the event, i.e., whether the event is to be detected when the event function decreases to zero, or increases to zero, or both. The bouncing ball example requires this information because it is necessary to distinguish hitting the ground from leaving it.

When event functions are defined, the solvers of the ODE Suite produce the output `[t,y,te,ye,ie]`, giving the times `te` at which events occurred, the values of the solutions `ye` at those times, and an index vector `ie` specifying which events occurred. By making all this available by means of optional output arguments, the user who wishes to solve a problem with no event functions is not inconvenienced in any way by the presence of this powerful and useful capability.

2.5 SIMULINK

Associated with MATLAB is a dynamic system simulation package, SIMULINK, that played, and continues to play, an important role in the development of the Suite. Although the Suite was developed in MATLAB with the aim of enhancing the capabilities of its solvers, the solution of ODEs is fundamental to SIMULINK and it was intended from the start that the solvers respond to the needs of this package. There are some important differences from the MATLAB environment. Initial value problems are solved considerably faster in SIMULINK because the solvers have been translated into C and, mainly, because the package compiles the equations, leading to a considerable reduction in the cost of function evaluations.

A continuous simulation language places certain constraints on the solution of ODEs, constraints that favor certain kinds of methods and ways of using them. The event location mentioned earlier is a capability required for SIMULINK. To locate events efficiently and accurately, it is necessary that a method allow a continuous extension so that it is possible to approximate very inexpensively the solution between time points. For this reason we considered for implementation in the Suite only methods with continuous extensions. Modified Rosenbrock methods are attractive in the context of a simulation package because they are one-step and do not require an exact Jacobian. With this in mind, we developed an effective modified Rosenbrock formula and a continuous extension for it. A simulation language also needs a capability of solving differential-algebraic equations, a capability that is the object of current research and development.

Providing for the requirements of SIMULINK led in several instances to a restructuring of the solvers of the MATLAB ODE Suite. In SIMULINK, additional information about a

problem such as the initial condition or the simulation interval is typically stored in a single file. This allows the user to specify and *encapsulate* an entire initial value problem in a single file. For the sake of this encapsulation and to achieve some compatibility with SIMULINK, the MATLAB solvers were redesigned so that they call a single user-supplied function to retrieve *all* information about a problem.

Using a single "gateway" function in the file defining the ODE differs markedly from the interface typical of existing scientific computing packages, but it has proved to be a convenient design. We mentioned earlier an example of the design when we indicated that invocation of the function defining the differential equation, with `chm6(t,y,'jacobian')` should cause it to return the Jacobian. This is coded easily with the string 'jacobian' being used in a case statement to call another function that evaluates the Jacobian or to select a block of code where the Jacobian is evaluated. And, the flexibility of MATLAB with respect to output makes it possible to return the Jacobian matrix, in either dense or sparse form, instead of the usual vector **f**(t,**y**). Another example is event location. Event location is common and it involves specification of perhaps several event functions, whether the corresponding events are passive or terminal, and the directionality desired of the event function. It is convenient to have all this information available in a single place, especially the evaluation of all the event functions.

3 TEACHING PACKAGES

Soon after microcomputers made it possible to solve quickly an ODE and display the solution graphically, packages for teaching ODEs with computer experiments began to appear. Notable examples are Phaser (Koçak, 1989) and MDEP (Buchanan, 1992) for PCs and MacMath (Hubbard and West, 1992) and Differential Systems (Gollwitzer, 1991) for Macintosh machines. In this context the emphasis is almost entirely on graphical display of solutions and on the easy definition of initial value problems. The packages cited limit the possibilities so as to facilitate this. As one example, several packages contain more than one piece of software with particular items of software aimed at a quite restricted class of problems, e.g., a single equation or a pair of equations. This is quite natural in the context of teaching ODEs because it is easy for students to visualize the solutions of such problems and to understand their properties. From an implementation point of view, allowing only one or two equations facilitates developing an interface that is easy to use. This, for example, makes it natural to input initial conditions by identifying points with a mouse and clicking. It also makes possible a particularly easy way to define the problems, namely a "fill in the blanks" approach. There are literally blank spaces on the screen where the user types a line of code or an initial condition. All the packages cited are stand-alone programs that parse the equations. Typically the equations are limited to a single line and the vocabulary is restricted. Although many interesting equations can be solved, the complexity possible is quite limited. For example, the packages make no provision for conditionals and loops. Because the solvers are part of the package and storage was a problem at the time the packages cited were written, a premium was placed on simple solvers.

As faster microcomputers with much more memory have become available, teaching packages have begun to expand their capabilities. It is interesting to contrast a teaching

package with a scientific computing environment like MATLAB. That they are rather different in purpose and approach is made clear by Polking's teaching package (Polking, 1995) which is a graphical user interface to MATLAB. In teaching packages the student has to learn some rules of syntax and some kind of simple editor, but the emphasis is on simplicity of description and interpretation. As a consequence the packages are restricted to simple problems and limited in the way the results can be analyzed. It is striking that none of those cited provides for the solution of stiff problems, presumably because of the premium placed on simple, short programs. Indeed, it is not clear that "difficult" problems can be solved with the packages, difficult, that is, in the sense that step sizes ranging over many orders of magnitude are necessary. In the case of the kinetics problem leading to Figure 1 the `ode15s` code used step sizes that ranged from 5×10^{-14} to 10^2! Despite this, the problem is not a hard one for a suitable integrator, as made evident by the fact that `ode15s` needed only 139 steps to solve the problem. Such problems certainly arise in the more advanced ODE classes, and the packages have not provided for them.

A different tack is being taken in a multimedia teaching project (ODE Architect, 1997) by C·ODE·E, the Consortium for Ordinary Differential Equations Experiments. It relies upon state-of-the-art integrators, specifically RKSUITE (Brankin et al., 1992) for its explicit Runge-Kutta (4,5) pair and DVODE (Brown et al., 1989) for its Adams-Moulton and BDF options. RKSUITE and DVODE differ in a number of respects, e.g., the form of the equations, the error control, and the diagnostics, but with a few changes and a thin interface they look the same to the graphical user interface. The solvers are compiled and the actual solution of an ODE is fast. Problems are defined using a small editor so they can be more substantial than is usual in teaching packages, but the vocabulary is restricted to facilitate parsing of the equations in the package. Exploiting existing software is appropriate, but the fact that RKSUITE and DVODE were written for a different computing environment has led to some difficulties that we take up now.

Like all the popular codes for general scientific computation, RKSUITE and DVODE monitor the work by either counting the number of function evaluations or the number of steps. This is inappropriate in the present context for several reasons. For one, function evaluations do not provide a realistic measure of the cost of solving a problem. For another, the absolute cost is not the issue, rather it is a question of how long a user is willing to wait for the solution to be completed. Much the same is true in MATLAB. In the latter environment users can interrupt the computation at any time. By exercising the option to display the solution as it is computed, the user can recognize that a solver has gotten stuck somewhere. Something of this kind is the standard way to solve problems in ODE Architect. The solution itself is not displayed until the whole integration is completed, but the fraction of the interval completed is always displayed in real time. If the solver gets stuck, the user can click on a "stop sign" icon to terminate the integration and display the portion of the solution that has been computed.

Integrators that can solve hard problems like those used in the ODE Architect are not enough in themselves. Hard problems may exceed the display capabilities of typical teaching packages because it may be necessary to plot solution components of greatly different scales separately and it may be desirable, or even necessary, to resort to logarithmic scales as in Figure 1. There is no intrinsic limitation on the number of equations, but there is a qualitative difference in the input of the equations and the display of the results when more than a few are treated. ODE Architect consists of a set of modules, each for supple-

menting the teaching of a single topic, along with tools for defining, solving, and displaying the solution of a system of ODEs. It is appropriate in some modules to provide a "fill in the blanks" definition of the small systems of equations that arise in the topic. The tools themselves are more-or-less general purpose. The tasks of the modules, the number of equations that can be displayed easily on a single screen, the complexity of an interface for specifying which solution components are to be displayed, and the limited complexity imposed by the vocabulary led to a decision to allow no more than 10 equations.

A difficulty in displaying the solution of hard problems is in deciding where to compute and report results. Simplicity has favored reporting results at equally spaced times, even when a variable step size is used for computing these results. A rather nice feature of the ODE Architect computational engine is the possibility of animating the displays to show the effects of changing parameters. The way this is done argues strongly for reporting tables of solutions at equally spaced time steps. However, if sharp changes in the solution are possible, and this is typical of stiff problems, a table of solution values at equally spaced time steps may not reproduce faithfully the behavior of the solution. Figure 1 is a concrete example. The matter is exacerbated in the phase plane plots that are quite important in teaching ODEs nowadays. A natural remedy is to proceed as in the MATLAB ODE Suite and record results at every time step (and more often with the (4,5) Runge-Kutta pair of RKSUITE). This introduces the complication of not knowing in advance how much storage will be needed. MATLAB deals with this by virtual storage. In a smaller package like ODE Architect, this is another reason for limiting the number of equations. With a limit on the number of output points, it is possible that the code will return without having reached the end of the interval. In the context of a teaching package, an acceptable amount of storage ought to deal with problems that can be integrated in an acceptable amount of real time.

REFERENCES

Brankin, R.W., Gladwell, I., and Shampine, L.F. (1992) RKSUITE: a suite of Runge-Kutta codes for the initial value problem for ODEs. Softrept. 92-S1, Math. Dept., SMU, Dallas.

Brown P.N., Byrne G.D., and Hindmarsh A.C. (1989) VODE: a variable-coefficient ODE solver. *SIAM Journal on Scientific and Statistical Computing*, **10**, 1038–51.

Buchanan, J.L. (1992) MDEP Midshipman Differential Equations Program, v. 2.28. Math. Dept., U.S. Naval Academy, Annapolis, MD.

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

Gollwitzer, H. (1991) Differential Systems User Manual, v. 3.0. Dept. Math. & Comp. Sci., Drexel Univ., Philadelphia, PA.

Hubbard, J.H. and West, B.H. (1992) MacMath 9.0 A Dynamical Systems Software Package for the Macintosh. Springer, New York.

Koçak, H. (1989) Differential and Difference Equations through Computer Experiments, 2nd ed. Springer, New York.

ODE Architect, Consortium for Ordinary Differential Equations Experiments. Wiley, New York, to appear.

Polking. J.C. (1995) MATLAB Manual for Ordinary Differential Equations, Prentice-Hall, Englewood Cliffs, NJ.

Shampine, L.F. and Reichelt, M.W. (1997) The MATLAB ODE Suite. *SIAM Journal on Scientific Computing*, **18**(1), to appear.

DISCUSSION

Speaker : L. Shampine

J. de Swart : For the more general case $M(t,y)y = f(t,y)$, how is the index of the variables determined?

L. Shampine : At present only non-singular $M(t)$ are permitted. It is planned to add some capabilities for DAEs to the MATLAB ODE suite.

J. Reid : I am surprised to hear that you ask users to supply the sparsity pattern of the Jacobian. Why not obtain it by automatic differentiation?

L. Shampine : For the reader, I should point out that users are not required to supply this. At present MATLAB has no provision for automatic differentiation. Personally I'd be glad to have the capability.

D. Higham : The provision of a suite of ODE solvers with the same interface is clearly useful. However, to what extent is the accuracy achieved (for a given problem and tolerance) insensitive to a change in the ODE solver?

L. Shampine : It isn't possible to match perfectly the response of solvers based on quite different algorithms, but they are "tuned" so as to be comparable for routine problems.

R. Brankin : Could you elaborate on the event location technique used?

L. Shampine : Events are recognized by a change of sign of an event function in the course of a step. Zeros are then located by a combination of the Illinois algorithm and bisection applied to all the event functions and the zero occurring first is returned.

M. Gentleman : Other than mathematical and numerical problems you ran into, would you like to comment on software problems you had with your development, and particularly what kind of bugs you experienced?

L. Shampine : From the perspective of a long-time FORTRAN programmer, I found working in MATLAB to be more productive because it is a higher level language and it is an environment with excellent run-time diagnostics and on-line help. The bugs that arose were mainly due to learning MATLAB as I was developing code. The language expertise necessary for such a project was supplied by my co-author.