Golden ODEs

New ordinary differential equation solvers for MATLAB and SIMULINK

by Cleve Moler

After matrices, ordinary differential equations–ODEs–are MATLAB's most important mathematical objects. ODEs are also at the heart of SIMULINK's modeling and simulation capabilities.

For many years, MATLAB has had only two ODE solvers available, ode23 and ode45. Even though they employ fairly simple algorithms, they have proved remarkably effective. SIMULINK provides some additional methods, but they are not easily accessible to MATLAB users. We have recently developed a suite of new, more powerful, ODE solvers for MATLAB. A preliminary version of the suite is available from The MathWorks FTP site and the complete suite will be included in MATLAB version 5 and SIMULINK version 2.

The authors of the ODE suite are Larry Shampine and Mark Reichelt. Shampine is a professor at Southern Methodist University, the author of two books and several dozen papers on the numerical solution of ODEs, and a consultant to The MathWorks. Reichelt is a member of The MathWorks development staff.

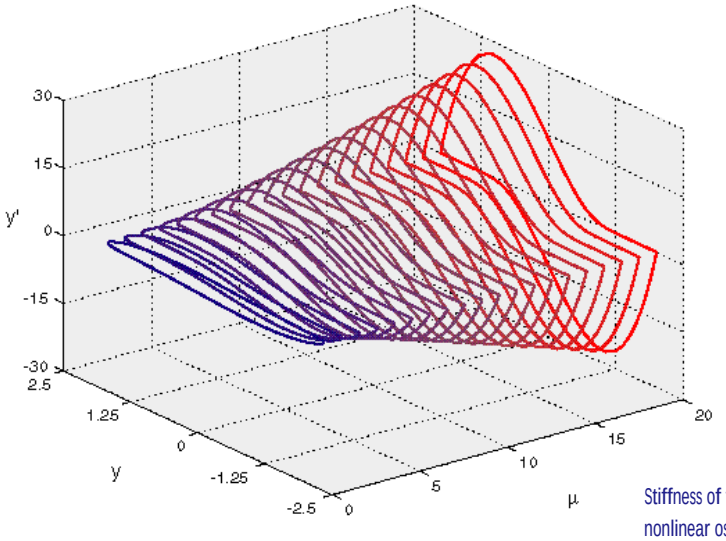
There are five new routines in the suite; each of them is the method of choice for a particular set of problems. Two of the routines are intended to replace ode23 and ode45; they are based on new methods and have significant new capabilities. Two other routines are intended for *stiff* problems. The fifth routine is the only one in the suite that is based on a classic formula, whereas the other four are based on new formulas developed by Shampine and others specifically for the MATLAB and SIMULINK environment.

All five new routines share a common calling sequence. All are intended to solve a general system of differential equations of form

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

where t is the scalar independent variable, $y = y(t)$ is the vector of dependent variables, the prime denotes differentiation with respect to t, and f is a vector-valued function defined in an M-file.

All five new routines approximate the solution by sampling it at a sequence of discrete time values, t_n, and all five automatically vary the step size, $h_n = t_{n+1} - t_n$, in such a way that an estimate of the error in the solution is less than a specified tolerance.



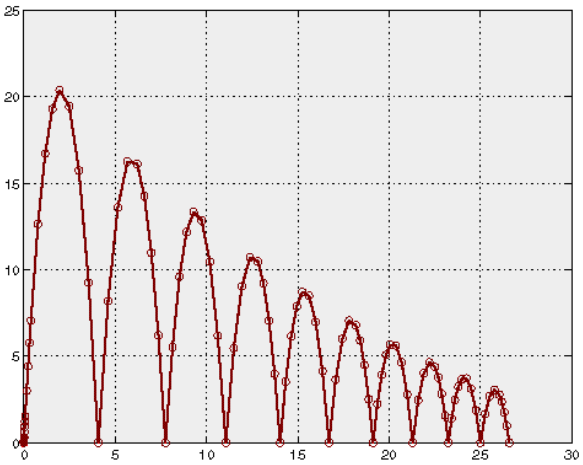
Stiffness of the van der Pol nonlinear oscillator

$y'' = \mu(1 - y^2)y' - y$

increases as μ increases.

The digits in each routine name, like the 2 and 3 in ode23, refer to the *order* of the underlying method. So, ode23 employs two approximations, one of order 2 and one of order 3, and uses their difference to estimate the error in each step. All the methods are based on local polynomial or Taylor series approximations. The order is roughly the degree of the polynomial or the number of terms in the Taylor series. More precisely, the order is p if the local error is proportional to h^{p+1} and the overall error is proportional to h^p. For a third order method, if the step size is cut by a factor of 1/2, the overall error will be reduced by a factor of about 1/8.

Numerical methods for ODEs can be classified as *single step* or *multistep*. The best known single step algorithms are the classic Runge-Kutta methods. Carl Runge was one of the fathers of modern applied mathematics; he published several differential formulas for the numerical solution of ODEs in 1895. Wilhelm Kutta independently introduced the general method in 1901. The basic idea is to treat $f(t,y)$ as a function of two independent variables, sample the function for several different values of these variables, combine the samples to approximate the Taylor series, and integrate the resulting



Interpolation and event handling facilities allow the efficient simulation of a bouncing ball.

approximation over one time step. Each step of the classic method begins anew; nothing is remembered from previous steps. There is an interesting, but complicated, relationship between the number of function evaluations per step and the resulting order. One, two, three or four evaluations lead to methods of order one, two, three or four. But six evaluations are required to achieve a fifth order method.

The classic multistep methods were developed by J. C. Adams. His first publication was in 1883, although he apparently knew about the techniques for many years before then. The idea is to use the values of y at several previous time steps as the basis for a polynomial approximation over the next time step. In principle, the order can be increased without increasing the number of function evaluations per step by simply using more previous values. In practice, modern multistep methods automatically determine the order, as well as the step size, and rarely use orders above 12 or 13.

Numerical methods for ODEs can also be classified as *explicit* or *implicit*. Explicit methods simply evaluate the function $f(t,y)$ for values of t and y determined earlier in the algorithm. Implicit methods solve nonlinear systems of simultaneous equations derived from the function f to obtain the next value of y at each time step. The solution process uses matrix computations involving the Jacobian matrix of partial derivatives, $\partial f/\partial y$. In some special cases, the functions in the Jacobian are easy to find and an M-file can be provided which evaluates them. But, more often, it is inconvenient to generate the Jacobian and so it must also be approximated numerically.

Implicit numerical methods are important for solving *stiff* problems. Stiffness is an elusive and sometimes misunderstood concept. Roughly, it means there are components of the solution that are varying on such widely different time scales that the size of the step taken by explicit methods is unacceptably small. A stiff problem is not unstable, or ill-conditioned, or even difficult to solve; it just takes classic Runge-Kutta or Adams methods an impossibly long time to achieve reasonable accuracy. We'll have an example later. The first widely used software for stiff problems was published by C. W. (Bill) Gear in 1971. Gear's software uses a class of variable order, multistep, implicit methods known as "backwards differentiation formulas," or BDFs.

Modern numerical methods should also come with *interpolants*; these allow the solution $y(t)$ to be evaluated for any value of t without evaluating $f(t,y)$. They also allow efficient *event handling* and *zero crossing*, which involves finding a value of t for which $y(t)$ is equal to a specified value. Adams and BDF methods have natural interpolants, but developing interpolants for Runge-Kutta methods is a contemporary research topic.

With this background, we can now describe the five new ODE routines. All five include error estimation, step size control, and interpolation.

`ode23`	A new pair of explicit Runge-Kutta formulas, of orders 2 and 3.
`ode45`	A new pair of explicit Runge-Kutta formulas, of orders 4 and 5.
`ode113`	A new implementation of the explicit Adams predictor-corrector methods, with variable order from 1 to 13.
`ode23s`	A new pair of linearly implicit Runge-Kutta formulas, of orders 2 and 3.
`ode15s`	A new class of implicit multistep methods, "numerical differentiation formulas," NDFs, with variable order from 1 to 5.

The letter s stands for stiff. The s methods can be used for nonstiff problems, but they are inefficient because each step requires solution of a linear equation involving the Jacobian. The non-s functions can be used for stiff problems, but they are inefficient because each step is too small.

One difficulty with the old `ode45` is that it is "too accurate." Its natural step size choice produces output which is too widely spaced to give good, smooth plots of the solution. With its interpolation feature, the new `ode45` can produce smooth output at reasonable cost.

In his talk at last fall's MATLAB Conference, Shampine

DE suite is available
The MathWorks. Point
Web browser to the
Works home page
/www.mathworks.com
en click on "Software
y, MathWorks
s." A postscript copy of
r by Shampine and
lt is available in the
directory.

e Moler is chairman
co-founder of
MathWorks. His
il address is
r@mathworks.com.

illustrated the new ODE suite with an interesting example, a scalar, nonlinear equation

$$y' = y^2(1 - y)$$

This equation is a simple model of flame propagation. (See R. E. O'Malley, *Singular Perturbation Methods for Ordinary Differential Equations*, Springer, 1991). The initial condition prescribes a small value for y and the integration is carried out over an interval whose length is twice the reciprocal of the initial value

$$y(0) = 10^{-4}$$

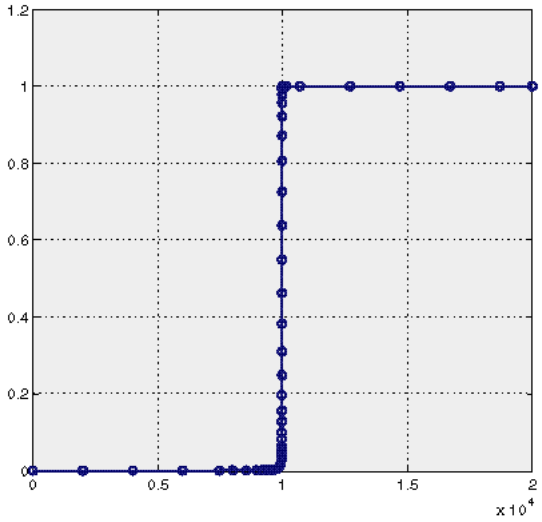
$$0 <= t <= 2 \cdot 10^4 \quad .$$

If the equation were simply $y' = y^2$, the solution would be $y(t) = 1/(10^4 - t)$, which becomes infinite in finite time as t approaches the midpoint of the interval. But the $(1 - y)$ factor prevents $y(t)$ from becoming larger than 1. The first graph shows the solution computed by `ode23s`.

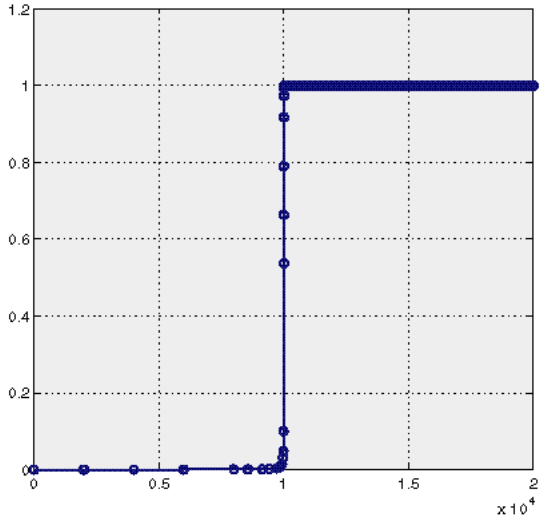
This problem is interesting because it changes from being nonstiff to stiff halfway through the interval. Stiffness is often defined by saying the Jacobian has widely varying eigenvalues, but in this case, the Jacobian is 1-by-1; there is only one eigenvalue. On the second half of the interval the solution is very stable. It behaves like $y(t) = 1 - e^{-t}$, which doesn't blow up and doesn't decay; it just sticks close to $y = 1$.

Both two stiff solvers handle this problem easily. `ode23s` takes only 55 steps to traverse the interval. `ode15s` takes a little longer, 103 steps. The three nonstiff solvers do not fare so well on the last half of the interval. The second graph shows the output produced by `ode45`. It covers the first half of the interval more quickly than the stiff solvers, but then requires over 3000 steps and over two minutes of computer time to complete the job. The computed result is within the prescribed tolerance (which is 10^{-3}); it just takes too long to compute.

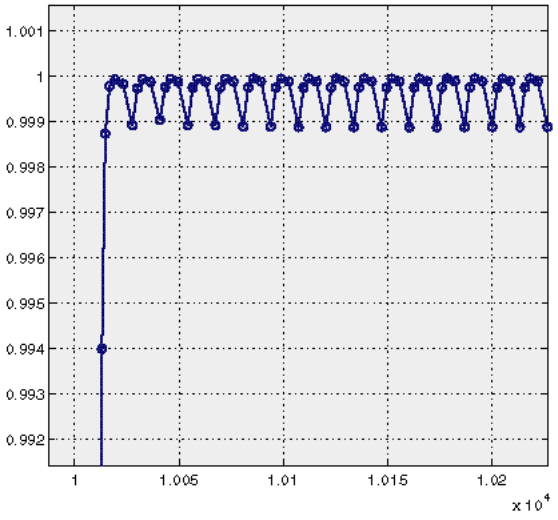
The third graph shows stiffness in action. We have zoomed in the portion of the `ode45` graph where $y(t)$ first approaches 1. Think of yourself hiking along a narrow canyon with steep walls on either side. If you keep looking straight ahead and anticipate where you are going, you will not have any trouble. This is what the implicit methods are capable of doing. But if you sample the terrain by stepping to either side, the steep gradients will force you to oscillate back and forth across the desired trajectory. You will eventually reach your destination, but it will probably be long after dark when you get there. ■



`ode23s` solves a stiff problem easily.



`ode45` takes over 3000 steps.



Detailed look at `ode45` behavior.