Are we there yet?

Zero crossing and event handling for differential equations

by Cleve Moler

Ordinary differential equations are like kids on a family vacation; they're always asking when it's time to stop. When does the bouncing Mars Pathfinder hit the surface? When is a car's engine speed close enough to its transmission speed for the clutch to engage? When does the price for an options contract hit the strike price?

The new ordinary differential equation solvers available in MATLAB 5 and Simulink 2 provide the features necessary to answer these questions. There are five new solvers. Each of them solves a system of ODEs of the form

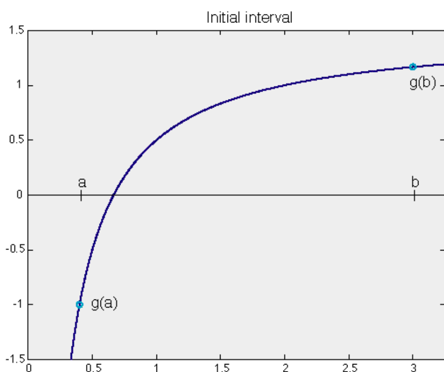
$$y' = f(t,y)$$

where f is a vector-valued function of the independent variable, t, and the vector of dependent variables, y. The initial values, y_0, are specified at t_0. The solvers compute a sequence of points t_1, t_2, …, and corresponding numerical solutions, y_1, y_2, … . The new solvers also include interpolation formulas that define a smooth numerical solution, $y(t)$, for all values of t.

A key question is: When should we stop? With the old solvers you had to specify a final value t_f. This is still possible, but it is sometimes more useful to specify a second function, $g(t,y)$, and then seek a stopping value t_s so that

$$g(t_s) = g(t_s, y(t_s)) = 0 \quad .$$

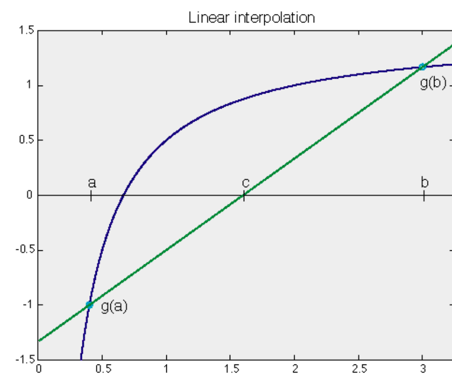
The stopping function g could be the distance from the Pathfinder to the surface of Mars, the difference between the speeds of the engine and the transmission, or the difference between the option and strike prices. The zeros of $g(t)$ are known as *zero crossings* or *event locations*. JPL's Fred Krogh, one of the first people to include such a feature in general purpose ODE software, calls them *g-stops*.



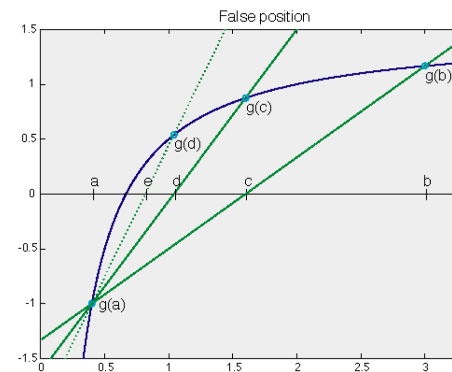
The MATLAB function `fzero` finds a zero of any function of a scalar variable. It has been in MATLAB for years and should be perfect for this job. But we didn't use it. Here's why.

The first graph shows the situation when a solver is near a stopping point. We have found an interval, $a < t < b$, where the function $g(t)$ changes sign. Our job is to find a point t_s in this interval where the function value is close to zero.

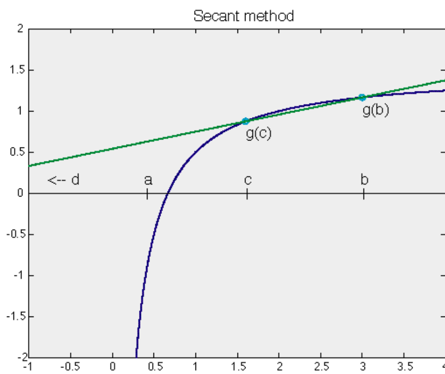
The second graph shows a simple linear interpolation step. The resulting point c is our first approximation to t_s. This works fine. What's next?



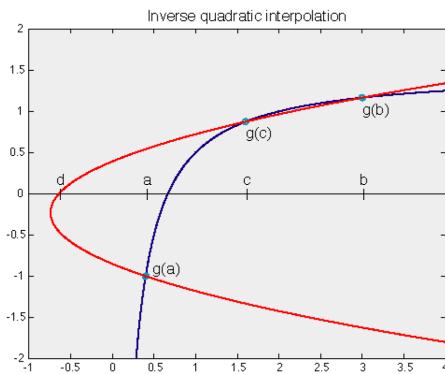
If we confine ourselves to linear interpolation using function values of opposite sign, the result is the method of false position or "regular falsi." The third graph shows that for a convex function, one "old" value, a in our example, is used over and over again. The resulting convergence is so slow that the method is not useful.



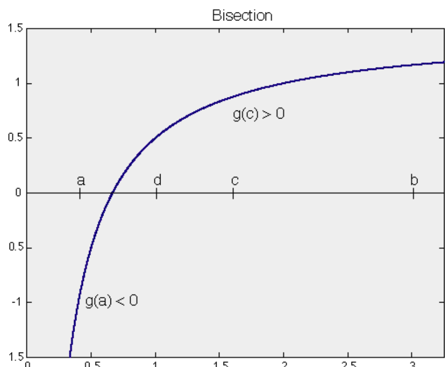
The secant method has a faster convergence rate than false position. It involves linear interpolation using the two most recently computed function values, even when they have the same sign. It's very effective when you're close to the zero. But the fourth graph shows that the extrapolated zero can be far outside the interval.



Inverse quadratic interpolation, IQI, is another rapidly convergent algorithm. It involves fitting a "sideways" parabola through three function values and then evaluating that parabola at zero to get a fourth point. The fifth graph shows that this point can also be outside the interval. When we get closer to the solution, so that the curvature doesn't vary so much, this method can also be very effective.



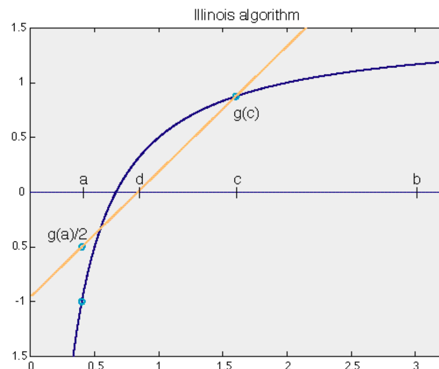
The ultimate in reliability is interval bisection, shown in the sixth graph. Only the signs of the function values are used. With our example, the function changes sign between *a* and *c*, so the next iterate is taken to be the midpoint of this interval. This algorithm is guaranteed to find a tiny interval on which



the function changes sign. But, if our convergence criterion is that the uncertainty in the computed t_s should be less than the IEEE floating-point roundoff error in the original *a* and *b*, then bisection will take 52 steps to finish. That's too many.

MATLAB's `fzero` combines secant, IQI and bisection. It uses secant or IQI if they are within the interval and bisection if they aren't. It combines the reliability of bisection with the ultimate speed of the other two algorithms. Originally called `zeroin` or Dekker's algorithm, it has a long and interesting history, dating back to the early 1960's, with contributions by Dekker, van Wijngaarden, Zonneveld, Dijkstra and Wilkinson. Important refinements were made in 1973 by Richard Brent. It has been featured in numerical methods textbooks by Shampine and Allen, Forsythe, Malcolm and Moler, and Kahaner, Moler and Nash. It's a terrific algorithm–but only for scalar-valued functions of a scalar variable.

In our differential equations context, we may have several different stopping criteria. The function $g(t)$ may be a vector-valued function. Extending the logic of `fzero` to the vector case is impractical because different components of g may take different paths through the algorithm. In other words, `fzero` can't be *vectorized*.



So Larry Shampine, our advisor on ODE solvers, resurrected an even older algorithm, probably invented in the 1950's at the University of Illinois, in one of the world's first computer centers. The Illinois algorithm is discussed in a survey paper by P. Jarratt in a 1970 book, *Numerical Methods for Nonlinear Algebraic Equations*, edited by Philip Rabinowitz. The basic idea is illustrated by our final graph. Instead of bisection in *t*, it uses a kind of bisection on the values of *g*. When an old point, like *a*, is reused in a linear interpolation step, its function value is cut in half. The Illinois algorithm shares the reliability that the bisection algorithm obtains by staying in an ever shrinking interval, but avoids the slow convergence that plagues the method of false position. And it can be vectorized. ■

Cleve Moler is chairman and co-founder of The MathWorks. His e-mail address is moler@mathworks.com.