

The Savvy Solver

Larry Shampine

Modern codes for the numerical solution of an initial value problem (IVP) ask as little of you as possible. About all you have to do is define the mathematical problem by providing \mathbf{F} , $[a,b]$, and $\mathbf{y}(a)$ in

$$\mathbf{y}' = \mathbf{F}(t, \mathbf{y}), \quad a \leq t \leq b, \quad \mathbf{y}(a) \text{ given}$$

and define the computational problem by stating where you want answers and how accurate they should be. Although the better codes are reliable, efficient, and easy to use, the savvy solver knows that sometimes the codes need a little help. Ed Spitznagel presented an exercise on two-compartment pharmacokinetics to the Harvey Mudd Workshop that illustrates one of those times.

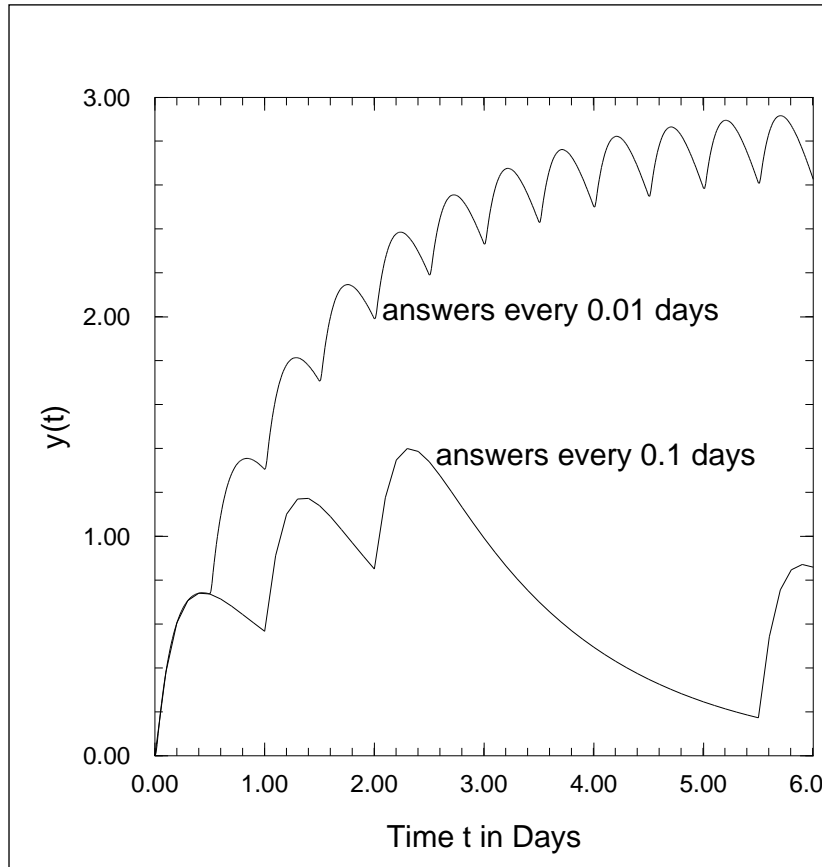
Example with Puzzling Results. A model for transport of a drug (with just under a three hour half-life in the GI tract and 24 hour half-life in the blood) when one tablet is taken every twelve hours is

$$\begin{aligned} \mathbf{x}' &= -5.6\mathbf{x} + 48 \text{ pulsep}\left(t, \frac{1}{48}, \frac{1}{2}\right), & \mathbf{x}(0) &= 0, \\ \mathbf{y}' &= 5.6\mathbf{x} - 0.7\mathbf{y}, & \mathbf{y}(0) &= 0. \end{aligned}$$

The time t is measured in units of days. The function $\text{pulsep}(t, w, p)$ defines a pulse of period p . The pulse is of unit height for $0 \leq t \leq w$ and zero for $w < t < p$, and extended periodically into $t > p$. The term involving $\text{pulsep}(\bullet)$ in the first differential equation corresponds to a pill that is released uniformly over half an hour ($= 1/48$ day), and the factor of 48 accounts for 1 pill in $1/48$ a unit of time (*Ed: see article by E. Spitznagel, this issue*).

The book by D. Kahaner, C. Moler, and S. Nash, **Numerical Methods and Software**, Prentice-Hall, 1989, provides a good survey of numerical analysis at an intermediate level and a diskette containing a small library of quality mathematical software. The library includes an excellent code, **DDRIV2**, for the IVP. This code implements both Adams-Moulton formulas and backward differentiation formulas (BDFs). The figure shows the level of the medication in the blood, $y(t)$, over a period of 6 days as computed with the Adams formulas. One curve was obtained when answers were requested every 0.01 days, and the other when they were requested every 0.1 days. The *only* difference between the two runs was how frequently answers were requested. Although the results are dramatically different, the code claims the same accuracy for both runs, so what is going on?

How ODE Codes Work. To understand the figure, we first need to talk a little about how the codes work. All the popular codes are based on methods that start with the given value $\mathbf{y}(a)$ at $t_0 = a$ and then step to b producing successive approximations $\mathbf{y}_j \approx \mathbf{y}(t_j)$ on a mesh $a = t_0 < t_1 < \dots < t_N = b$. They select automatically the size of the step from t_j to t_{j+1} . To minimize the cost, the codes try to take steps that are as long as



possible, but, of course, the steps must be short enough that each approximate value y_j have the required accuracy. There are two ways to get answers at specified points, and the savvy solver knows that how this is done can really matter. One way is to adjust the step size so that the specified points are in the mesh. This is typical of Runge-Kutta codes. Carefully done, it works fine, but asking for a “lot” of answers at specified points can put the cost up enormously and have other undesirable consequences that we’ll not take up here. (Savoir-faire: Since the codes produce

answers frequently where the solution changes rapidly, if you just want to see how the solution behaves, the best way to use such a code is to accept answers at the points t_j selected by the code rather than interfere with the integration by specifying where answers are to be computed.) In the other way of getting answers, the code interpolates the y_j to produce answers at the specified points. This is inexpensive, and in principle, it scarcely matters how many answers you want and where they are. As is typical of Adams and BDF codes, **DDRIV2** obtains answers this way.

Do Codes See Pulses? Using only a finite number of samples, a differential equation solver tries to approximate the solution throughout the interval. Clearly this is possible only if we make strong assumptions about how the solution behaves between samples, i.e., about how smooth the solution is. All methods sample at least once in a step, and most sample several times. What has happened with the example is that the step size selection algorithm has lost the scale of the problem. The code starts with a step size small enough that it “sees” the first dose by evaluating the right hand side of the differential equation during the time the pill has an effect. The equations are easy to integrate, so the code increases the step size rapidly, so rapidly that in one of the runs, the code doesn’t “notice” the second dose because it does not evaluate F during the time the pill is being absorbed. After a couple of days, the integration is going so well that the code skips over a good many doses. Familiarity with step function input causes people to underestimate the strain they place upon an integrator. Because the change is discontinuous, the code has no inkling from the preceding computations that a dramatic change is

about to occur, and it can easily lose the scale of the problem. Though not necessary in principle, codes with an interpolation capability do pay some attention to where answers are specified because generally this conveys a measure of scale. Here requesting answers every 0.1 days seems frequent, yet the absorption of a pill takes place in 0.02 (= 1/48) days, a rather shorter time scale, and the code gets into trouble. On the other hand, requesting answers every 0.01 days should be frequent enough to see the effects of the pills, and that proves to be the case.

Savvy Solver to the Rescue. The savvy solver informs the code about the time scale on which phenomena occur. Just how this is done depends on the code, and the fact is, there is more than one way to do it when using **DDRIV2**. Some codes let you specify a maximum step size, a lower level subroutine called by **DDRIV2** being a case in point. The trouble with this is that a maximum step size applies to the whole run, and it may be needlessly small where the solution is smooth. If your code produces answers at specified points by stepping to those points, all you have to do is ask for answers whenever you expect rapid change. Here, for example, asking for an answer when a pill is swallowed and when it has been completely digested will convey the necessary scale information. This is usually good enough for codes that do interpolation, too, but there is a way you can be sure. Because it is not always permissible for a code to step past a certain point and get an answer there by interpolation, all quality codes with interpolation give you the option of preventing the code from stepping past a specified point. You can use this option to make sure that the code “notices” very sharp changes like the discontinuities of the example. You don’t have to get involved with step size selection, just help the code recognize a drastic change of scale---it will refine the step size as necessary to deal with the change.

There is a closely related difficulty that the savvy solver will keep in mind. The first step is particularly difficult for the codes as they try to determine automatically the scale of the problem. The codes exploit all the information available to them, but at the initial point this information is very limited and sometimes even the best codes are fooled. The distance to the first output point is one of the things that codes use to recognize the scale. The savvy solver will specify a first output point that indicates the scale on which phenomena occur near the initial point even when an answer at this point is of no interest in the application. To be concrete, if things can happen in seconds and you are interested in answers at intervals of weeks, you should tell the code to produce an answer after the first few seconds even though you have no use for the answer. Doing this won’t cost you a thing, and it may make the difference between reliable results and nonsense. □

*Editorial Comment (SMH): The graph accompanying this article was developed in a two step process. The actual numeric solutions were generated by the author using **DDRIV2**, but to create the PostScript output he saved the data in a file and used it as input to **MDEP** (see review this issue). The hardware used was a Dell 310 IBM compatible 386 machine with a 387 coprocessor. The two step method of producing output was necessary only to make it possible to include the graph in this publication.*

This method of manipulating output is very common, and many problems in computational mathematics require the use of multiple programs to create a numeric solution, display it, and finally create an output that can be printed or shared with others.