

the next generation of Intel chips, Windows NT, and PC versions of NeXTStep and Solaris which take advantage of the new hardware.

(Editors Note: Lerner's article is the first in a series on setting up an ODE computer lab. Stay tuned for further information.) □

## The Savvy Solver II

Larry Shampine

Southern Methodist University

Dallas, TX 75275

*Ed. note: Second order linear ODEs can have solutions that decay and others that become unbounded. This divergence of solutions causes problems that require an understanding of how solvers function. Shampine's example shows one difficulty and how the savvy solver resolves it.*

**Airy Functions:** Several important functions are defined as solutions of Airy's equation,  $y'' = xy$ .  $Ai(x)$  is the solution with initial values given below, and  $Bi(x)$  is an independent solution:

$$Ai(0) = \frac{3^{-2/3}}{\Gamma(2/3)} \approx 0.355028$$

$$Ai'(0) = \frac{-3^{-1/3}}{\Gamma(1/3)} \approx 0.258819$$

$$Bi(0) = Ai(0) \sqrt{3}$$

$$Bi'(0) = -Ai'(0) \sqrt{3}$$

Airy's equation is so simple that a natural

way to compute  $Ai(x)$  is to integrate the initial value problem. I did this with "top-of-the-line" codes based on the most popular kinds of discrete variable methods, namely Runge-Kutta, Adams, and backward differentiation formulas. In each case the code was asked to produce about five correct digits at  $x=1$  and  $x=10$ . The results were:

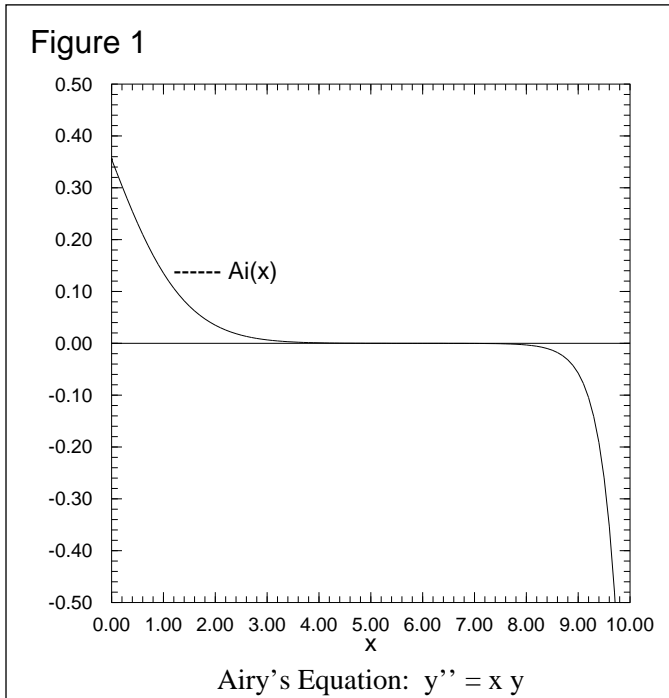
	x=1	x=10
Runga Kutta	.135300	3742.20
Adams	.135298	3140.81
back solve	.135293	548.399
$Ai(x)$	.135292	$\sim 10^{-10}$

**Bad results from good software:** The results at  $x=1$  are what you might expect, but those at  $x=10$  are terrible! The numerical solution plotted (Figure 1) was obtained using the variable step size Runge-Kutta code in the package MDEP (see the review in the last issue of this newsletter). It is just as bad as the other solutions. Towards the end of the interval it is negative and its magnitude is increasing rapidly. This despite the fact that  $Ai(x)$  is positive for  $x \geq 0$  and decays rapidly to zero. It looks like the codes all failed, but they *didn't* – they did just what they were supposed to do! The savvy solver knows that what codes do is different from what you want them to do, so let's talk about the difference.

**Forgetting the Past:** Discrete variable methods for the numerical solution of

$$y' = F(x, y), a \leq x \leq b, y(a) = A$$

start with the given value  $y_0 = A$  and then produce an approximation  $y_1 = y(x_1)$  at a point  $x_1 > a$ . The process is repeated, successively producing approximations  $y_j$  on a



mesh  $a = x_0 < x_1 < \dots < x_N = b$  that spans the whole interval.

What the codes try to do on the step to  $x_1$  is what you'd expect, but what they try to do when stepping from  $x_j$  to  $x_{j+1}$  is not so obvious. It would seem that they ought to try to approximate  $y(x_{j+1})$ , but they do **not** try to do this, at least not directly. Some numerical methods do not even "remember" results computed prior to  $x_j$ . On reaching  $x_j$ , all a code based on such a method has at its disposal is the current approximate solution  $y_j$  and the ability to evaluate  $F$ . Because of this, the best that it can do is to approximate well the **local solution**, the solution of the differential equation that has the value  $y_j$  at  $x_j$ :  $u'(x) = F(x, u(x))$ ,  $u(x_j) = y_j$ . Some numerical methods do remember results at a few points  $x_{j-k}, x_{j-k+1}, \dots, x_j$  prior to the current one, but only a few points, so that for most purposes they, too, "forget" what has transpired. Because of this, the codes all try

to control the **local error**,  $u(x_{j+1}) - y_{j+1}$ , in the step to  $x_{j+1}$ , rather than the **global error**,  $y(x_{j+1}) - y_{j+1}$ , that you would expect.

**Unstable ODEs:** The savvy solver understands that what matters is not **the** solution  $y(x)$  of the given initial value problem, but rather the **family** of solutions that start out near  $y(x)$ . At each step the numerical method tries to approximate one of these solutions, namely the current local solution. Generally it makes a small error, and the numerical result lies on a nearby solution curve, the next local solution. The cumulative effect of these local errors depends on the behavior of the family of solution curves. When solutions spread apart rapidly, even one small local error can lead to large global errors. It is simply not possible to solve unstable differential equations accurately with discrete variable methods. On the other hand, when solutions come together, the effect of a small local error is damped out in the global error. Because solution curves might come together for  $x$  in one portion of  $[a, b]$  and spread apart in others, the behavior of the global error that interests us can be complex.

**Trouble in Airy City:** In computing  $Ai(x)$  numerically, suppose that at some point  $x_j$  we compute  $y_j = Ai(x_j)$  and  $y_j' = Ai'(x_j)$ . From  $x_j$  on, the best we can hope to do is to compute exactly the local solution  $u(x)$ , the solution of Airy's equation with initial values  $y_j, y_j'$  at  $x_j$ . The two solutions  $Ai(x)$  and  $Bi(x)$  are linearly independent, so all solutions of the equation can be written as linear combinations of these two, and in particular,  $u(x) = \alpha Ai(x) + \beta Bi(x)$ . Of course, we could work out what  $\alpha$  and  $\beta$  are here, but all we need do is observe that

if  $y_j$  and  $y_j'$  are good approximations to  $Ai(x_j)$  and  $Ai'(x_j)$ , then  $\alpha$  is close to 1 and  $\beta$  is close to 0. The solution  $Ai(x)$  that we wish to approximate decays rapidly for large  $x$ ,

$$Ai(x) \sim \frac{1}{2}\pi^{-1/2}x^{-1/4} \exp\left(-\frac{2}{3}x^{3/2}\right)$$

In contrast,  $Bi(x)$  grows rapidly,

$$Bi(x) \sim \pi^{-1/2}x^{-1/4} \exp\left(\frac{2}{3}x^{3/2}\right)$$

Regardless of how small  $\beta$  is, as  $x$  gets large, the rapidly growing term  $\beta Bi(x)$  in  $u(x)$  swamps the term  $\alpha Ai(x)$ , and the numerical result is useless as an approximation to  $Ai(x)$ . It is important to understand this: No matter how accurate the approximations at  $x_j$  are, eventually the numerical solution will be useless - even if the integration is exact for **all**  $x > x_j$ . On the interval  $[0, 1]$ , the initial value problem for  $Ai(x)$  is moderately stable. Because of this, control of the local error implies that the global error can at worst grow slowly. Accordingly, the codes are able to produce accurate results. For larger  $x$ , the problem is unstable and the global error can grow rapidly even when all the local errors are small. The inaccurate results produced by the codes are a consequence of an unstable problem, not a failure to control the local error.

**Go Backward!** Because the initial value problem is unstable, we cannot compute  $Ai(x)$  accurately for large  $x$  with a discrete variable method. However, there is a trick that makes this possible. Besides illuminating the role of stability, it exemplifies a standard device for computing certain special functions, e.g. Bessel functions, by recur-

rence. **The trick is to reverse the direction of integration.** Suppose we start at some "large"  $x_0$  with arbitrary initial values  $y_0, y_0' \neq 0, y_0'$  and integrate to the origin. The solution of Airy's differential equation with these initial conditions is a linear combination of  $Ai(x)$  and  $Bi(x)$ . When integrating towards the origin, it is  $Ai(x)$  that strongly dominates  $Bi(x)$ , so whatever the initial values, a code will soon be approximating a multiple of  $Ai(x)$ , say  $\gamma Ai(x)$ . Suppose  $y_N$  is the computed solution at  $x = 0$ . Since  $y_N \approx \gamma Ai(0)$  and we know  $Ai(0)$ , we can deduce the factor  $\gamma$ . For  $x_j$  far enough from  $x_0$ , and given the computed  $y_j \approx \gamma Ai(x_j)$ , all we have to do is divide  $y_j$  by  $\gamma$  to get an accurate solution for  $Ai(x_j)$ .

**MDEP is great!** It allows you to define constants, so I defined  $Ai(0)$  and  $Ai'(0)$  so that they would be available for other purposes. Because you can integrate in either direction, I could start out at  $x_0 = 10$  with a guessed solution value of  $10^{-6}$  and a slope of 0, and integrate to the origin. The package allows access to the computed value  $y_N$  and I defined it to be another constant.

MDEP has a calculator that allows constants to be used in the expression to be evaluated, making it easy to calculate  $\gamma$ . Rather than use  $\gamma$  to obtain accurate solutions at particular points, I used the calculator to compute the more accurate value of  $2.227 \times 10^{-10}$  for starting the integration at 10. A second integration with this value and a slope of 0 provided an accurate solution that had all the expected qualitative properties of  $Ai(x)$  on  $[0, 10]$ .  $\square$