Lecture 8

Numeric Techniques

Understanding numeric techniques for solving equations on the computer is an important technique that is used in almost all types of science these days. The study of numeric techniques is an entire field to itself, with people devoting huge effort to figuring ways to get the most speed and accuracy for each type of equation around. A good discussion of the techniques for doing many things numerically (not just solving differential equations, like we are doing) can be found in the various editions of "Numeric Recipes" by Press, Teukolsky, Vetterling, and Flannery (available in FORTRAN, PASCAL, and C editions, at least). The basis for this lecture is material in Chapters 16 & 19 of the FORTRAN edition of this book.

8.1 Ordinary Differential Equations

Any ordinary differential equation (ODE), of any order, can be reduced to a set of first-order ODEs (note that this will *NOT* be the case for partial differential equations, which we will treat later). For example, the generic equation

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x),$$

which is a second-order ordinary differential equation can be rewritten as two first-order equations:

$$\begin{aligned} \frac{dy}{dx} &= z(x) \\ \frac{dz}{dx} &= r(x) - q(x)z(x), \end{aligned}$$

where z is a new variable.

Thus a knowledge of how to numerically solve first-order ODEs will allow you to solve almost any arbitrary-order differential equation.

The underlying idea of any numeric solution is always the same one that we discussed earlier for *Euler's method* (and that you have used in the problem sets): rewrite the dx's and dy's as finite steps Δx and Δy and multiply the equations by Δx. This gives algebraic formulae for the change in functions when the independent variable x is "stepped" by one "stepsize" Δx. In the limit of makeing Δx very small, a good approximation to the real solution is achieved. But, according to *Numeric Recipes*, use of Euler's method (which is what we've been doing!) "is not reccomended for any practical use." What to do instead? There are three major types of practical numeric methods for solving ODEs,*Runge-Kutta* methods, *Extrapolation* methods (often called *Bulirsch-Stoer*, and *predictor-corrector* methods. We will only consider the first of these. The other two are mentioned just so that if you here the words somewhere you are least have an idea what the people are talking about.

8.2 Runge-Kutta method

Runge-Kutta is the standard numeric method that is used in probably 90% of ODE numeric integration. It is conceptually simple, straight-forward to implement, and works more often than not. Unless you have a very good reason for doing otherwise, it should be the one you use.

Once again, the formula for using the Euler method to solve $dy/dx = f(x, y)$ is

$$y_{n+1} = y_n + hf(x_n, y_n)$$

which advances the solution from x_n to x_{n+1} where $h = x_{n+1} - x_n$ (Figure 8.2). One way to see just how inaccurate this method is to realize that this method is unsymmetrical: if you were to start at x_{n+1} and try to go backwards to find the value of the solution at x_n, you would get a different answer!

How much error is there in this method? If we use a Taylor expansion, we find that

$$y_{n+1} = y_n + hf(x_n, y_n) + \frac{h^2}{2}f'(x_n, y_n) + \frac{h^3}{6}f''(x_n, y_n) + ...$$

If h is small, then h^2 is even smaller and h^3 is smaller still. The Euler method then accumulates errors that are similar in size to h^2. Because it is only accurate to the first power of h, it is called "first-order" accurate. This is often written

$$y_{n+1} = y_n + hf(x_n, y_n) + O(h^2)$$

where $O(h^2)$ means "terms of order h^2."

One way to get higher order accuracy is to take a trial Euler step to the midpoint of the interval and then calculate the derivative at that point. This

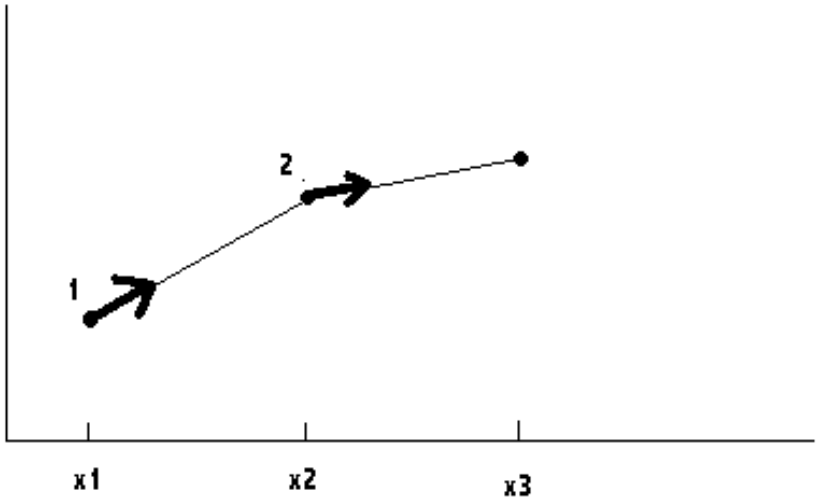

Figure 8.1: Euler's method. In this simplest (and least accurate) method for integrating an ODE, the derivative at the starting point of each interval is extrapolated to find the next function value. The method has first-order accuracy.

derivative is then used to take the full step. Figure 8.2 illustrates the idea. In eqautions,

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + h/2, y_n + k_1/2) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned}$$

As we could see from a Taylor expansion, this method is accurate to second-order in h, and so is called the *second-order Runge Kutta* method.

We could continue on forever, figuring out schemes to cancel higher and higher order errors. But most people stop after a few interations and use the classical fourth-order Runge-Kutta formula, which is a good tradeoff between complexity and power. In this method:

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + h/2, y_n + k_1/2) \\ k_3 &= hf(x_n + h/2, y_n + k_2/2) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + k_1/6 + k_2/3 + k_3/3 + k_4/6 + O(h^5) \end{aligned}$$

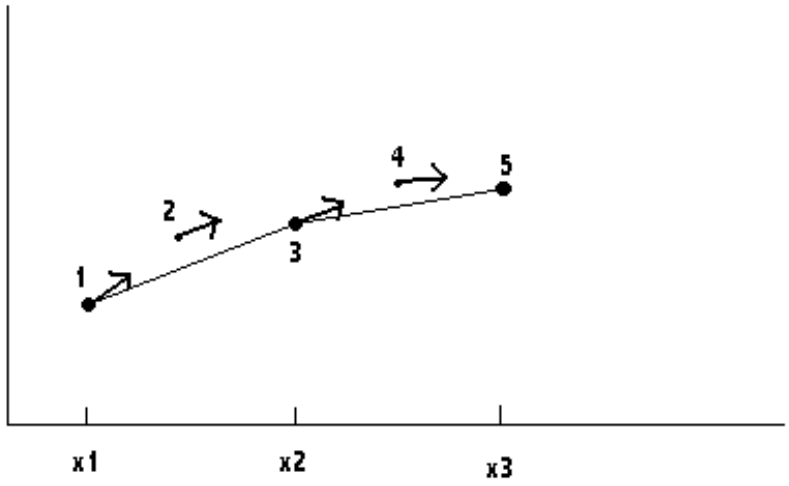

Figure 8.2: The midpoint, or second-order Runge-Kutta, method.

As you can see from the last term, this method is accurate to fourth-order in h, hence its name.

Now you may object at this point: yes, this method is more accurate, but it also requires that we compute the value of the function (which is what takes the most computer time) four times just to advance the solution to the next point, instead of simply once in the Euler method. Thus, to be useful, this method must allow us to get comparable accuracy using steps that are four times larger than those used for the Euler method. Is this the case? Almost always. It would require construction of a particularly pathological set of ODEs to find a case where the Runge-Kutta method didn't save considerable time and accuracy, though such pathologies do exist in the world.

We now quote directly from *Numeric Recipes* on the use of Runge-Kutta:

> For many scientific uses, fourth-order Runge-Kutta is not just the first word on ODE integrators, but the last word as well. In fact you can get pretty far on this old workhorse, especially if you combine it with an adaptive stepsize algorithm [we'll discuss this in the next section]. Keep in mind, however, that the old workhorse's last trip may well be to take you to the poorhouse: Bulirsch-Stoer or predictor-corrector methods can be very much more efficient for problems where very high accuracy is a requirement. These methods are the high-strung racehorses. Runge-Kutta is for plowing the

fields.

8.3 Adaptive Step-Size

A good ODE integrator should exert some adaptive control over its own progress, making frequent changes in its stepsize. Usually the purpose of this adaptive stepsize control is to achieve some predetermined accuracy in the solution with minimum computational effort. Many small steps should tiptoe through treacherous terrain, while a few great strides should speed through smooth uninteresting countryside. The resulting gains in efficiency are not mere tens of percents or factors of two; the can sometimes be a factor of ten, a hundered, or more.

Implementation of adaptive stepsize control requires that the stepping algorith return information about its performance, most important an estimation of the truncation error. Obviously, the calculation of this information will add to the computational overhead, but the investment will be repaid handsomely.

With fourth-order Runge-Kutta, the most straightforward technique by far is *step-doubling*. We take each step twice, once as a full step and then independently, as two half-steps. How much overhead is this? Each of the three separate Runge-Kutta steps in the procedure requires four evaluations of the function, but the single and double steps share a common starting point, so the number of evaluations is 11. This has to be compared not to 4, but to 8 (the two half-steps) since – stepsize control aside – we are achieving the accuracy of the smaller (half) stepsize. The overhead cost is therefore a factor of 1.375. What does it buy us?

Let's denote the exact solution for an advance from x to $x+2h$ by $y(x+2h)$ and the two approximate solutions by y_1 (one step $2h$) and y_2 (2 steps, each of size h). Since the basic method is 4th order, the true solution and the two numeric approximations are related by

$$\begin{aligned} y(x+[2h]) &= y_1 + (2h)^5\phi + O(h^6) \\ y(x+2[h]) &= y_2 + 2(h^5)\phi + O(h^6) \end{aligned}$$

where, to order h^6, the value ϕ remains constant over the step. Taylor series expansion shows that ϕ is a number of order $y^5(x)/5!$. The first expression above involves $(2h)^5$ because the stepsize is $2h$, while the second involves $2(h^5)$ because each of two stepss is h. To good approximation, the truncation error in the first term is $2^4 = 16$ times greater than that in the second, so we will use the difference in the two solutions to estimate the truncation error:

$$\Delta = y_2 - y_1.$$

It is this difference that we shall endeavor to keep to a desired degree of accuracy, neither too large (because the answer would be too inaccurate) nor too small

(because the computation would take too much time). We do this by adjusting h.

From the above equations, we see the Δ scale as h^5. Thus, if we take a step h_1 and produce and error Δ_1, but wish to produce and error Δ_0 instead, we could simply rescale h_1 to obtain a new stepsize, h_0, equal to

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2}. \tag{8.1}$$

Henceforth we will let Δ_0 denote *desired* accuracy. Then the above equation is used in two ways: if Δ_1 is larger than Δ_0 in magnitude, the equation tells us how much to decrease the stepsize *when we retry the present (failed!) step.* If Δ_1 is smaller than Δ_0, on the other hand, the equation tells us how much we can safely increase the stepsize *for the next step.*

Practical matters: When writing a computer program to implement this adaptive-stepsize algorithm, there are two practical things that make things work a little smoother. First off, if we implement equation 8.1 literally, we run the danger of increasing the stepsize to just a tiny bit above the exact right amount, which means that the next step taken will be too big and will fail and have to be redone, wasting time. If, however, we implement a *safety factor*, S, that we multiply the increase in the stepsize by, we make it much less likely to have this problem. So instead of equation 8.1, we use

$$h_0 = S h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2}.$$

where a good value for S tends to be something like 0.9.

Second, when adjusting the stepsize, we run the risk of being in a relatively calm section of a function not knowing that there is a cliff ahead. If stepsize is allowed to increase quickly it might accidentally increase too much. So a good practical thing to do is to *never* increase the stepsize by more than a factor of say 5 at any one time.

With these two practical matters in hand, you should now be able to sit down and write your own stepsize control program.

8.4 Partial Differential Equations

The Runge-Kutta method is a great for use in ODEs and is probably the only thing you will ever need. Partial differential equations (PDEs) are another matter entirely. Where for ODEs we could come up with clever schemes to adapt stepsizes and to make solutions more accurate, for PDEs there is little we can do to speed things along. In practical terms, this fact means that the grids used to explore solutions to PDEs are often quite small. As you say in

the problem set last week, even a 100 by 100 grid is enough to make things run quite slowly.

Let's again consider the diffusion equation:

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2}$$

where we have incorporated the ρc term and k (which we are implicitly consider spatially constant) in D, the diffusion coefficient.

As we found in a previous problem set,

$$\begin{aligned}
\frac{\partial^2 T}{\partial x^2} &= \frac{\partial}{\partial x}\frac{\partial T}{\partial x} \\
&= \frac{\partial}{\partial x}\lim_{dx\to 0}\frac{T(x) - T(x+dx)}{dx} \\
&= \lim_{dx'\to 0}\lim_{dx\to 0}\frac{(T(x-dx') - T(x)) - (T(x+dx-dx') - T(x+dx)}{dx\ dx'}
\end{aligned}$$

which we can transform to small steps by assuming $dx = dx' = \Delta x$', so

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T(x-\Delta x) - 2T(x) + T(x+\Delta x)}{\Delta x^2}.$$

But what do we do about the fact that for a PDE, T is now a function of both t *and* x? In a previous problem set, we considered the simplest (so, once again, least accurate) way of doing this. We simply calculated this second spatial derivative at each point and each time and then propagated the solution using this calculation. This method is precisely analogous to Euler's method, discussed above, and is refered to as a FTCS implementation (Forward Time Centered Space).

In this implementation, the diffusion equation becomes

$$u_j^{n+1} = \Delta t D\left[\frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{(\Delta x)^2}\right] + u_j^n,$$

where the superscript refers to the spatial dimensions and the subscript refers to the time dimension.

This numeric solution is stable if

$$\frac{2D\Delta t}{(\Delta x)^2} \leq 1.$$

The physical interpretation of this restriction is that the maximum allowed timestep is approximately the diffusion time across a cell of width Δx.

Consider now a form of implementation that is almost as obvious

$$u_j^{n+1} = \Delta t D\left[\frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{(\Delta x)^2}\right] + u_j^n.$$

This is exactly like the FTCS scheme except that the spatial derivatives on the right-hand side are evaluated at timestep $n+1$. Schemes with this character are call *fully explicit* or *backwards time*, by contrast with FTCS (which is called *fully explicit*). To solve this equation, one has to solve a set of simultaneous linear equations at each timestep for the u_j^{n+1}.

So what good is this now much more complicated equation? Just like in the Runge-Kutta scheme, we can now average the fully implicit and fully explicit implementations to cancel out the second-order error to make the implementation accurate to second order:

$$u_j^{n+1} = \frac{\Delta t D}{2}\left[\frac{(u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}) + (u_{j+1}^n - 2u_j^n + u_{j-1}^n)}{(\Delta x)^2}\right].$$

This scheme is called *Crank-Nicholson* differencing. Honestly, though, unless you are doing serious numeric solutions of seconds order PDEs, you are unlikely to find use of this. And if you *are* doing serious numeric solutions, you will want to look up the details in a book like *Numeric Recipes*. So mostly the point here is to make you familiar with some of the methods of these sorts of numeric solutions so you'll have some ideas what to do if you ever run into them.