MATH 211 LECTURE 12

Numerical Methods, Part II

There is more to discuss about numerical solvers. Today we highlight the remaining topics.

1. Our big problem–Error

We are always balancing a couple of troubles when we use a numerical method.

Computing time: The number of computations we have to do adds up and costs real time. The most difficult thing is usually the number of times we have to evaluate the function. In Euler's method, this is one time per step. In the Runge-Kutta methods it is 2 and 4 times per step.

truncation error: All of the known methods use Taylor's theorem one way or another to approximate complicated f's. But a whole power series is hard to compute, so usually only a couple of terms are used. By truncating the series intentionally, we introduce errors.

propagation of error: Errors we make at the last step lead to even more error in future steps. The picture is the best way to understand, but the idea is that starting with the second step we are not actually approximating our solution, but another solution which is close to the one we want. This can really add up over a long interval. There is basically nothing to be done about this. Long intervals are hard to work with.

step size: Using smaller step size increases the number of computations we have to do. However, we expect that in general smaller step sizes reduce error.

Computer roundoff error: Computers can only really deal with a finite set of rational numbers. Usually, decimals with a fixed number of digits. MATLAB typically is capable of using 24 digits. Every time a computation is done, the result has been rounded off. Usually this is so small as to be not so significant compared to the truncation error. However, it can lead to a problem not apparent at first: can't choose arbitrarily small step size.

kinky output: If the step size is not small enough (or even if it is), we can often get outputs from a numerical method which are decidedly not smooth curves. Some modern solvers fill in extra data by interpolating from the points it found and some knowledge of the differential equation. This is sometimes called "dense output". It makes things more palatable, and is usually pretty accurate.

With these lessons learned: One can estimate the truncation error for different methods using the error terms in Taylor's theorem. This is what leads to the notion of *order* for a numerical solution method. Euler's method has total truncation error bounded by something like

$$\text{error} \approx\leq C \cdot h,$$

where C is a function which depends only on the function f in the region of approximation, and h is the step size. This does not account for internal cancellations, so usually we do a little better than $C \cdot h$. So halving the step size about halves the error. Hence, Euler's method is a first order solver: h appears to the first power.

Perhaps discuss how this happens with Taylor's theorem (local error vs. total error).

Similarly, Runge-Kutta methods of order 2 and 4 above are as advertised because the respective errors are $C \cdot h^2$ and $C \cdot h^4$. Notice that this leads to a significant reduction in error for halving the step size. The price to be paid is computation time.

There are MATLAB m-files for these three methods available on the web. Get them and type `help euler` or `help rk2` to find out how they work.

Discuss the examples from the homework problems and show graphs–problems 3, 8, 11.

Your homework this week involves some problems which study the error in the methods we have described. One uses log-log graphs for this. Explain why this is typical for situations where we expect power laws.

2. Variable step solvers

One trick to bypass some of the difficulties is to use a variable step solver. In Euler, rk2, rk4, the step size was fixed over the whole procedure. Here we will allow it to vary from step to step. In fact, we will require our algorithm to choose the next step size. It should do this in a way to minimize the local truncation error at each step. Instead of specifying the step size in such a routine, one specifies the maximum allowable local truncation error.

Typical scheme: Use two solver schemes, but at a common set of points (to reduce the number of times we must evaluate the function). The two methods should have different error orders. Using the local error estimates for each solver, we can come up with an equation for the error being made in terms of the size of the step we want. If the error is small enough, we keep the step. If it is too large, we throw away the computation and reduce the step to an appropriate size and recompute.

The Dormand-Prince solver in `dfield` uses a version of this, so does `ode45` in MATLAB.

3. Stiff equations

Some equations have things happening on vastly different time scales. This can fool fixed step size solvers completely, and it can cause a variable step solver to overheat

with enormous amounts of calculation. Such equations require specially conditioned solvers, which are very inefficient in comparison to the standard methods for most equations. So, if you suspect that your equation is *stiff* it is worth trying one of these special stiff solvers. MATLAB has one in its ODE suite called `ode15s`. Stiff equations happen fairly easily in systems, so this will be an important consideration later in the semester.

Example. The equation $y' = e^t \cdot \cos(x)$ is stiff. Problem: derivative is very small some places, very large in others, even over short intervals. Try to use `dfield` on this one with the different methods. Use a window like $0 \leq t \leq 10$, $0 \leq x \leq 20$.

Note that the existence and uniqueness theorems apply on the whole plane. Draw the level curves $cos(x) = 0$, these are equilibrium solutions. (Why?) Then see how Euler and the Runge-Kutta methods we have used give bad answers. Next show how Dormand-Prince (variable step) bogs down.

4. Available MATLAB routines

Recall that we have several things to use. Our list is basically:

m-files for basic routines: `euler, rk2, rk4`

fancy MATLAB routines: `ode45, ode15s`, Dormand-Prince in `dfield`.

Basic Advice:

- Start with a flexible variable step solver of intermediate order. Like the Dormand-Prince solver in `dfield`, or `ode45`.
- Look at results skeptically. If things seems strange, or even if they don't and the equation is unfamiliar and you don't know what to expect, start with a reasonable step size or local error bound, then cut that in half and rerun. If the results are significantly different, cut in half and rerun again. Keep at it until things stabilize.
- Always be aware that the numerical output could be total gibberish.