

Mat1062: Computational Methods for PDE

Mary Pugh

January 29, 2008

1 How to use truncation errors to test your code

Let's step back and look at what we've done in proving convergence for the explicit euler scheme for the heat equation $u_t = Du_{xx}$. Our finite difference scheme had two parts: the spatial discretization and the temporal discretization. And so if $v(x, t)$ is a smooth solution of the heat equation then

$$\frac{v_{j+1}^n - 2v_j^n + v_{j-1}^n}{h^2} = v_{xx}(x_j, t_n) + \frac{1}{12}h^2 v_{xxxx}(\tilde{x}, t_n), \text{ for some } \tilde{x} \in (x_{j-1}, x_{j+1})$$

and

$$\frac{v_j^{n+1} - v_j^n}{k} = v_t(x_j, t_n) + \frac{1}{2}kv_{tt}(x_j, \tilde{t}), \text{ for some } \tilde{t} \in (t_n, t_{n+1})$$

If we further assumed that $v_{xxxx}(x, 0)$ is uniformly bounded on \mathbb{R} then there is some $M < \infty$ such that $|v_{xxxx}(x, t)| \leq M$ and $|v_{tt}(x, t)| \leq M$ for all $x \in \mathbb{R}$ and all $t > 0$. As a result,

$$\left| \frac{v_{j+1}^n - 2v_j^n + v_{j-1}^n}{h^2} - v_{xx}(x_j, t_n) \right| = \mathcal{O}(h^2)$$

and

$$\left| \frac{v_j^{n+1} - v_j^n}{k} - v_t(x_j, t_n) \right| = \mathcal{O}(k).$$

Recall that " $A = \mathcal{O}(k)$ " means there exists a K and a constant C such that $|A| \leq Ck$ for all $k < K$. Note that because the bounds on v_{xxxx} and v_{tt} are independent of x and t , the constants C in the $\mathcal{O}(h^2)$ and $\mathcal{O}(k)$ bounds are independent of j and n .

From the above,

$$\left| \frac{v_j^{n+1} - v_j^n}{k} - D \frac{v_{j+1}^n - 2v_j^n + v_{j-1}^n}{h^2} - (v_t(x_j, t_n) - Dv_{xx}(x_j, t_n)) \right| = \mathcal{O}(k + h^2)$$

If we make the further assumption that $k = \lambda h^2/D$ then the right-hand side is $\mathcal{O}(h^2)$.

Using these local truncation errors and the stability of the scheme, we were able to control the true error $e_j^n = u_j^n - v(x_j, t_n)$:

$$\|e^n\|_\infty = \max_j |e_j^n| \leq h^2 t_n MD(\lambda + 1)$$

where $t = nk$ and $k = \lambda h^2/D$ and h is small enough so that $h < H$ and $k < K$ where H and K are such that the bounds (1) and (2) from the January 24 notes hold.

What does this mean? It means that at any point (x_j, t_n) in space time, if k (and hence h) is small enough then

$$|u_j^n - v(x_j, t_n)| \leq h^2 t_n MD(\lambda + 1), \implies u_j^n = v(x_j, t_n) + \mathcal{O}(h^2).$$

I can use this to test if I've written my code correctly. To do this, I consider the heat equation $u_t = u_{xx}$ on $[0, 1]$ with Neumann boundary conditions and initial data $u_0(x) = \cos(3\pi x)$. The exact solution is

$$v(x, t) = e^{-(3\pi)^2 t} \cos(3\pi x)$$

I choose the initial time is 0 and the final time is 1/10. I compute five discrete solutions: u_1 with $h_1 = 1/8$ and $k_1 = 1/160$, u_2 with $h_2 = h_1/2 = 1/16$ and $k_2 = k_1/4 = 1/640$, u_3 with $h_3 = h_2/2$ and $k_3 = k_2/4$, u_4 with $h_4 = h_3/2$ and $k_4 = k_3/4$, and u_5 with $h_5 = h_4/2$ and $k_5 = k_4/4$. That is, I'm dividing the space-step by two and the time-step by 4 with each refinement. They are chosen so that $\lambda = 2/5 < 1/2$.

We've proven that the scheme converges and because we know the exact solution, we can actually look at the true errors. Below, I present the errors at the final time $t = 1/10$:

	error	ratio
$\ u_1^{16} - v(\cdot, 1/10)\ _\infty$	1.2022e-04	2.7052
$\ u_2^{64} - v(\cdot, 1/10)\ _\infty$	4.4441e-05	3.6644
$\ u_3^{256} - v(\cdot, 1/10)\ _\infty$	1.2128e-05	3.9160
$\ u_4^{1024} - v(\cdot, 1/10)\ _\infty$	3.0970e-06	3.9790
$\ u_5^{4098} - v(\cdot, 1/10)\ _\infty$	7.7834e-07	

(recall that it took 16 time steps for u_1 to reach $t = 1/10$). The first thing you notice is that the errors are decreasing. This is good. But you can get more out of the errors. The second column is the ratio of subsequent errors. You note that as k (and hence h) gets smaller, the ratios appear to be getting closer to 4. This is what you're looking for.

What's going on? We know that

$$\|u_1^{16} - v(\cdot, 1/10)\|_\infty = \max_j |u_1^j - v(x_j, 1/10)| \sim (h_1)^2$$

Similarly,

$$\|u_2^{64} - v(\cdot, 1/10)\|_\infty \sim (h_2)^2 = (h_1)^2/4.$$

Combining the two,

$$\frac{\|u_1^{16} - v(\cdot, 1/10)\|_\infty}{\|u_2^{64} - v(\cdot, 1/10)\|_\infty} \sim \frac{(h_1)^2}{(h_2)^2} = 4.$$

And so as the space-step and time-step get smaller and smaller (but always by factors of 2 and 4, respectively) the ratios of the errors get closer and closer to 4. Note: there's nothing magical about factors of 2 and 4. If I'd taken the space-steps smaller and smaller by factors of 3 and the time-steps smaller by factors of 9 then the ratios get closer and closer to 9.

This looks all well and good, but what if I don't know the exact solution v ? In fact, you can still find what you're looking for:

	norm	ratio
$\ u_1^{16} - u_2^{64}\ _\infty$	7.5782e-05	2.3452
$\ u_2^{64} - u_3^{256}\ _\infty$	3.2313e-05	3.5782
$\ u_3^{256} - u_4^{1024}\ _\infty$	9.0307e-06	3.8948
$\ u_4^{1024} - u_5^{4098}\ _\infty$	2.3187e-06	

Again, the ratios are getting closer and closer to 4. Note: I should be clear about what I mean when I talk about the difference of u_1 and u_2 . Recall

that u_1 is defined at 9 grid points ($x_j = j/8$) and u_2 is defined at 17 grid points ($x_j = j/16$). In the above, I take the differences on the coarser mesh:

$$\|u_1^{16} - u_2^{64}\|_\infty := \max_{0 \leq j \leq 8} |u_1^{16} - u_2^{64}|$$

Why are those ratios going to 4? Loosely speaking,

$$\begin{aligned} \|u_1^{16} - u_2^{64}\|_\infty &\sim \|u_1^{16} - v(x_j, 1/10)\|_\infty + \|u_2^{64} - v(x_j, 1/10)\|_\infty \\ &\sim (h_1)^2 + (h_1/2)^2 = (h_1)^2(1 + \frac{1}{4}) \end{aligned}$$

and

$$\begin{aligned} \|u_2^{64} - u_3^{256}\|_\infty &\sim \|u_2^{64} - v(x_j, 1/10)\|_\infty + \|u_3^{256} - v(x_j, 1/10)\|_\infty \\ &\sim (h_1/2)^2 + (h_1/4)^2 = (h_1/2)^2(1 + \frac{1}{4}) \end{aligned}$$

and so

$$\frac{\|u_1^{16} - u_2^{64}\|_\infty}{\|u_2^{64} - u_3^{256}\|_\infty} \sim \frac{(h_1)^2(1 + \frac{1}{4})}{(h_1/2)^2(1 + \frac{1}{4})} = 4$$

One can make the above more rigorous.

I will now try to convince you of the power of all those ratios. Let's imagine that there's a bug in my code, somewhere in the spatial discretization, for example. For example, let's imagine I have an even number of subintervals in space and that at the middle grid point $x_{N/2}$ I mistyped and as a result I've coded up

$$u_{xx}(x_{N/2}) \sim \frac{u_{N/2+1} - 2u_{N/2} + 0u_{N/2-1}}{h^2}$$

(Note that there's a coefficient of zero there, instead of 1.) As a result, one of the $N + 1$ equations determining u^{n+1} is incorrect. How big of an effect could that have? It's only at one point, after all! The result of this error is that at the $N/2$ grid point we'll have

$$\left| \frac{v_{N/2+1}^n - 2v_{N/2}^n + 0v_{N/2-1}^n}{h^2} - v_{xx}(x_j, t_n) \right| = \mathcal{O}(h)$$

That is, the consistency at that grid point is $\mathcal{O}(h)$ rather than the $\mathcal{O}(h^2)$ we have at all the other grid points. But recall that the convergence arguments were in the supremum norm and so not only will what's going on at this

one grid point matter but it'll be what's most important because $h^2 < h$ for h small. If we follow through the argument as before we'll find that this buggy code does converge to a solution but that the error estimate is now

$$\|e^n\|_\infty = \max_j |e_j^n| \leq htMD(\lambda + 1)$$

Note that right-hand side is now $\mathcal{O}(h)$ rather than $\mathcal{O}(h^2)$!

Is this all in theory or does it show up in practice? To check this, I coded up this buggy code and repeated the ratio testing from before:

	error	ratio
$\ u1^{16} - v(\cdot, 1/10)\ _\infty$	5.1947e-02	1.6507
$\ u2^{64} - v(\cdot, 1/10)\ _\infty$	3.1470e-02	1.8815
$\ u3^{256} - v(\cdot, 1/10)\ _\infty$	1.6726e-02	1.9584
$\ u4^{1024} - v(\cdot, 1/10)\ _\infty$	8.5404e-03	1.9843
$\ u5^{4096} - v(\cdot, 1/10)\ _\infty$	4.3041e-03	

Note that not only are the errors larger than before but the ratios are going to 2. Because the ratios are going to 2 rather than 4, we know with 100% confidence that our code is not doing what we wanted it to do. Time to go bug hunting!

Remember back when we looked at how to code up the Neumann and Robin boundary conditions? We used

$$v_x(0) \implies \frac{v_1 - v_{-1}}{2h} = 0 \implies v_{-1} = v_1$$

This finite-difference approximation of v_x is $\mathcal{O}(h^2)$ close to v_x . And it works great in the code – it's what I used at each endpoint in the code that generated the first set of tables. What if we hadn't done this? What if we'd done

$$v_x(0) \implies \frac{v_0 - v_{-1}}{h} = 0 \implies v_{-1} = v_0$$

instead? This finite-difference approximation of v_x is $\mathcal{O}(h)$ close to v_x . And so we can't hope for any thing better than in our buggy code above. Here's what we get when implementing the Neumann boundary condition sub-optimally at $x = 0$:

	error	ratio
$\ u1^{16} - v(\cdot, 1/10)\ _\infty$	1.1542e-01	1.9517
$\ u2^{64} - v(\cdot, 1/10)\ _\infty$	5.9142e-02	1.9864
$\ u3^{256} - v(\cdot, 1/10)\ _\infty$	2.9773e-02	1.9963
$\ u4^{1024} - v(\cdot, 1/10)\ _\infty$	1.4914e-02	1.9989
$\ u5^{4098} - v(\cdot, 1/10)\ _\infty$	7.4613e-03	

Again, note that not only are the errors larger than before but the ratios are going to 2. Because the ratios are going to 2 rather than 4.

Okay, so do we really care about $\mathcal{O}(h^2)$ versus $\mathcal{O}(h)$? If we want to achieve an error of 10^{-9} we can do this with either code. So what's the big deal? This is where the runtime¹ shows up:

	error	runtime (sec)
$\ u2^{64} - v(\cdot, 1/10)\ _\infty$	5.9142e-02	4.0368e-03
$\ u3^{256} - v(\cdot, 1/10)\ _\infty$	2.9773e-02	6.2507e-02
$\ u4^{1024} - v(\cdot, 1/10)\ _\infty$	1.4914e-02	1.1203e+00
$\ u5^{4098} - v(\cdot, 1/10)\ _\infty$	7.4613e-03	5.2689e+01
$\ u6^{16384} - v(\cdot, 1/10)\ _\infty$	3.7313e-03	1.6711e+03

Note that

$$\text{err}(h) \sim ch^\alpha \implies \log_{10}(\text{err}(h)) \sim \log_{10}(c) + \alpha \log_{10}(h).$$

Doing a least-squares fit on the above data, I find $\alpha = 1.0$ (as expected) and $c = 0.94$. This implies that to get an error of $10e-08$ I would have to take $h = 1.06e-08$. Turning to the runtime,

$$\text{run}(h) \sim dh^\beta \implies \log_{10}(\text{run}(h)) \sim \log_{10}(d) + \beta \log_{10}(h).$$

Doing a least-squares fit on the above data, I find $\beta = 4.7$ and $d = 5.5e-09$. And so to reach an error of $1e-08$ it would take $2.4e+29$ seconds ($7.6e+21$ years). Which is a long time.

If I do the same calculations with the code where the boundary conditions were both implemented in an $\mathcal{O}(h^2)$ manner then I would have gotten

¹The runtime is done on my mac and I was using more RAM than CPU. Ideally, I'd want to get flop counts since this would be platform independent. But one can't get flop counts in matlab.

	error	runtime (sec)
$\ u2^{64} - v(\cdot, 1/10)\ _\infty$	4.4441e-05	3.6236e-03
$\ u3^{256} - v(\cdot, 1/10)\ _\infty$	1.2128e-05	5.9179e-02
$\ u4^{1024} - v(\cdot, 1/10)\ _\infty$	3.0970e-06	1.0569e+00
$\ u5^{4098} - v(\cdot, 1/10)\ _\infty$	7.7834e-07	5.0538e+01
$\ u6^{16384} - v(\cdot, 1/10)\ _\infty$	1.9484e-07	1.6319e+03

There's no marked change in the runtimes, obviously. The change in boundary conditions doesn't affect the code — it just affects the accuracy. Doing a least-squares fit as above, I find $\alpha = 2.0$ (as expected) and $c = 1.1e-02$. This implies that to get an error of $1e-08$ I would have to take $h = 9.5e-05$. Turning to the runtime, I find $\beta = 4.7$ and $d = 5.1e-09$. And so to reach an error of $10e-08$ It would take $8.3e+05$ seconds. Which is about ten days.

What could we do to make things go faster? As you'll see in your homework, if we'd done things via Crank-Nicolson then the truncation error would be $\mathcal{O}(k^2 + h^2)$. And rather than taking k smaller by factors of 4 each time, we'd take them smaller by factors of 2, just like h . This will make things run much faster.

A word of experience: if your time-stepping is $\mathcal{O}(k)$ then a bug in the implementation of the time-stepping will likely leave it $\mathcal{O}(k)$. Similarly, using accuracy to go bug hunting won't have an effect if the spatial discretization is $\mathcal{O}(h)$. So not only is getting higher accuracy a good idea for accuracy's sake but it provides a useful tool for bug-hunting.

Another word of experience: sometimes you'll have a code for which convergence has not been proven. But if you do runs and show errors that are decreasing and ratios that are going to a constant then it's consistent with a numerical scheme that's convergent. The next thing you need to convince yourself of is that what it's converging to is actually a solution of the problem you're interested in.