Engineering Mathematics

Scientific Computing

Numerical Methods for Solving Partial Differential Equations

December 21, 2017

James Keen

Abstract

1 Summary

The purpose of this software is to provide the user with a suite of functions to solve second order Partial Differential Equations (PDE's).These are equations that take the form $Au_{xx}+Bu_{xy}+Cu_{yy}+Du_x+Eu_y+F = 0$. There are many 'famous' PDE's such as the Wave Equation[3] or the LaPlace Equation[2] but PDE's in general can be used to describe a wide variety of different processes in which some quantity changes with respect to others. This is a very common occurrence in the real world so understanding how to solve these equations is an important mission.

1.1 Parabolic Equations

There are three main types of PDE. The first that this software deals with is Parabolic PDE's. The condition that makes a PDE Parabolic is $B^2 - AC = 0$. The quintessential example of a parabolic PDE is the Heat Equation[1] which the software was trained on.

The methods used to solve Parabolic PDE's are Forward Euler (FE), Backward Euler (BE) and Crank Nicholson (CN). Each of these methods has been vectorised and implement sparse matrix operations in order to improve their efficiency. The sections of this code are as follows:

- Define initial conditions and exact solution
- Define the evolution matrices for FE, BE and CN
- Define the boundary conditions (BC's) and Right Hand side function
- Forward Euler Method
- Backward Euler Method
- Crank Nicholson Method
- Error analysis
- Main Parabolic Solver Function

The way to use this software for 1 dimension is simply to call the main function, *Parabolic_PDE()*, telling it which method you would like to use to solve the PDE as well as the type of BC's you are using. This will then implement the desired scheme and give you the solution. The approach for 2 dimensions is simply to call *Parabolic_2D()* and it will provide the solution to the PDE.

It is worth noting that both BE and CN are implicit methods and are unconditionally stable, while FE is explicit and only stable when

$$\lambda = \kappa \frac{\Delta t}{\Delta x^2} < \frac{1}{2}$$

making it one of the less useful methods. However, it is computationally cheaper as it does not involve solving a great number of equations. The accuracy of these methods will be investigated in the next section.

1.2 Hyperbolic Equations

Hyperbolic Equations are similar to Parabolic except that the condition they must meet is $B^2 - AC > 0$. The example used in this software is the Wave Equation. The explicit Finite Difference method used here is given by

$$u_{i,j+1} = 2u_{i,j} + \lambda^2(u_{i-1,j} - 2u_{i,j} + u_{i+1,j}) - u_{i,j-1}$$

The structure of the package that solves Hyperbolic Equations is much the same as for Parabolic, except that it only contains two methods, one Explicit (*Finite_Difference_EW()*) and one Implicit (*Finite_Difference_IW()*). To use these solvers, you simply call the name of the type you wish to use with the relevant PDE and type of conditions you wish to solve. As in the previous case, the explicit scheme is conditionally stable for the Courant number, λ

$$\lambda = c\frac{\Delta t}{\Delta x} \leq 1.$$

It should be noted that at $\lambda = 1$ the truncation error is exactly 0

1.3 Elliptic Equations

Elliptic equations are the final type of PDE handled by this software. They take the same form as above and satisfy the condition $B^2 - 4AC < 0$. The example this software was tested on was Laplace's Equation.[2] This file is set up more simply than previous, as it only handles Elliptic PDE's simply. It uses Successive Over Relaxation with Gauss Seidel method.[4] The update formula for this is

$$R^k_{i,j} = \frac{u^{k+1}_{i-1,j} + u^k_{i+1,j} - 2(1+\lambda^2)u^k_{i,j} + \lambda^2(u^{k+1}_{i,j-1} + u^k_{i,j+1})}{2(1+\lambda^2)}$$

$$u^{k+1}_{i,j} = u^k_{i,j} + \omega R^k_{i,j}$$

There is only one significant function in this file, namely *Elliptic_Solver()*, which will solve the Elliptic PDE you ask it to with Dirichlet BC's only. This file will give you some diagnostic information if you would like it to. For instance, if run, it will print the error between the last approximation and the one before it, the number of iterations it has taken to arrive at this point and the true error between the approximation and the solution. This code could be made significantly more efficient if it were vectorised, as well as implementing sparse matrix operations simultaneously.

2 Design

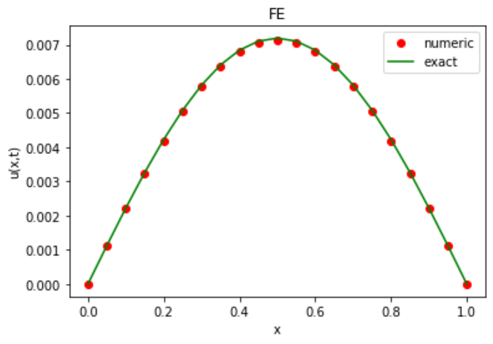
This code attempts to imitate best practise as well as it can. For instance, the vast majority of it has been split into functions that try not to rely on global variables too heavily and the plotting code has been separated from the main algorithms. It also attempts to solve these PDE's efficiently where it can although, as always, there is room for improvement.

2.1 Parabolic

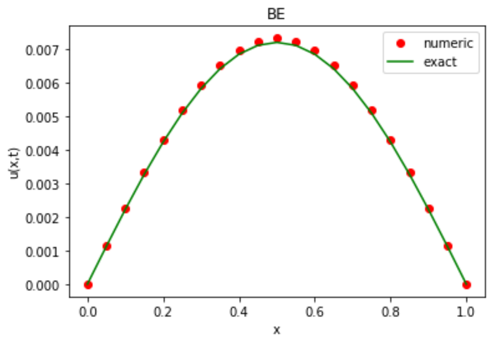
The 1D Heat equation is a simple one by the standards of PDE's. As such it can be relatively easily solved in a variety of different ways. The three methods used were Forward Euler, Backward Euler and Crank Nicholson. The solutions provided using these methods can be seen in Figures 1 and 2.

This code was written with efficiency in mind. Instead of using expensive for loops, this implements the methods above using as many vectors and matrices as possible. For instance, the matrix A_{FE} is used every timestep to calculate the solution for every point in space simultaneously in one operation as opposed to one operation for each point. Given that this A_{FE} matrix is mostly comprised of zeros, except along the three main diagonals, it makes sense to make it a sparse matrix, in order to increase efficiency for a vast number of calculations.

I decided to separate the boundary conditions off into their own functions to try and keep the section where the PDE is solved as clean as possible. In order to switch the solver between solving for Dirichlet or Neumann BC's, I made it so that there is an input flag that switches the behaviour of the solver slightly to adapt the matrices A or B to account for each type of BC's.



(a) Solution of the Heat Equation using Forward Euler.



(b) Solution of the Heat Equation using Backward Euler.

Figure 1

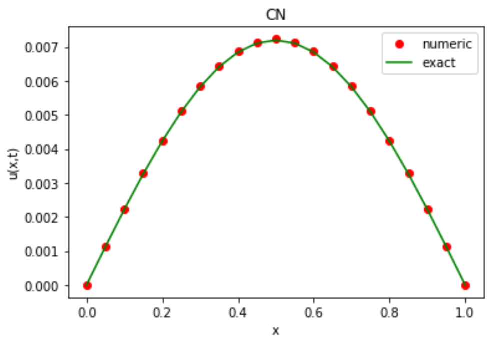


Figure 2: Solution of the Heat Equation using Crank Nicholson.

Given that there was a selection of methods used to solve in this section, I determined that there ought to be a main function that would enable the user to govern the whole script more readily. This lead to *Parabolic_PDE()*. This came in handy when it came to the error analysis as it lets the user easily switch between methods for simple solutions or a comparison of their error for a variety of different values of λ. This exact comparison can be seen in Figures 3. This shows that Forward Euler is actually the most accurate over this small selection of Δx's followed by Crank Nicholson then Backward Euler however, once λ gets above its critical value of $\frac{1}{2}$, FE becomes unstable and is no longer useful though it is quick (0.016 seconds), leaving just the two implicit methods to be compared. The error for these schemes will eventually begin to increase again as it runs into problems with rounding errors.

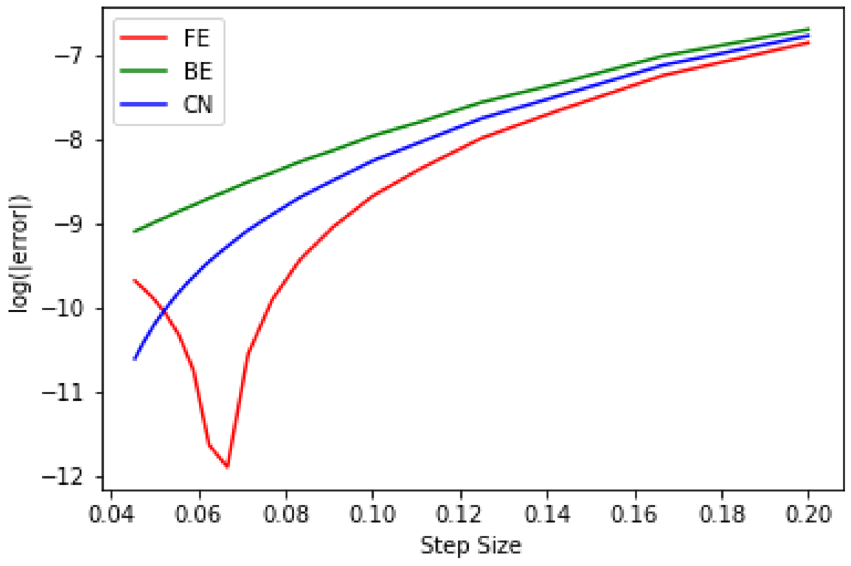


Figure 3: Comparison of the error for the three schemes. Forward Euler in red, Backward Euler in green and Crank Nicholson in blue.

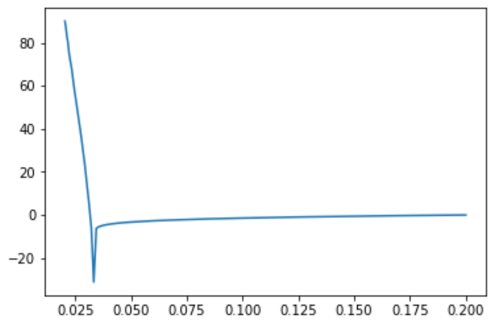
Parabolic2D.py implements the same Forward Euler method mentioned above but in 2 dimensions, as you might expect. I had problems implementing this code as nicely as I would have liked. As it stands, this code solves a Parabolic PDE (the Heat Equation) iteratively using the forward Euler approximation. Firstly, FE in 2D is conditionally stable for

$$\kappa \frac{\Delta t}{\Delta x^2 + \Delta y^2} \leq \frac{1}{8},$$

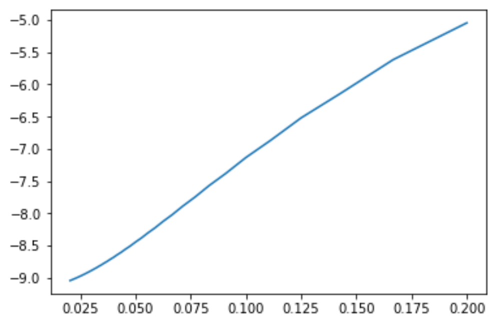
making it less useful for significant accuracy. Secondly it does not run as efficiently as it could. It proved to be a struggle to create a block matrix made up of matrices as would be required to make this 2D scheme more efficient as suggested in Worksheet 1.

2.2 Hyperbolic

The code in Hyperbolic1D.py also implements the same efficiency shortcuts with vectorisation and sparse matrices described in the previous section. However, the most significant difference is the lack of a main function. This is because there are only two different methods to potentially apply here so it is trivial for the user to switch one letter between *Finite_Difference_EW()* and *Finite_Difference_IW()*. As mentioned in the previous section on Hyperbolic PDE's, the explicit method is stable for $\lambda \leq 1$, while the implicit method is unconditionally stable. This can be shown by Figures 4a, 4b and 5. They show that at a specific value for stepsize the error for the explicit shrinks to zero but then begins to climb as the instabilities in the scheme take over. However, the error for the implicit scheme in Figure 5 shows the error consistently decreasing. At some point, this too will run into rounding error that is unavoidable in a computation such as this.



(a) Error of the explicit Finite Difference Method on the Wave Equation.



(b) Error of the implicit Finite Difference Method on the Wave Equation.

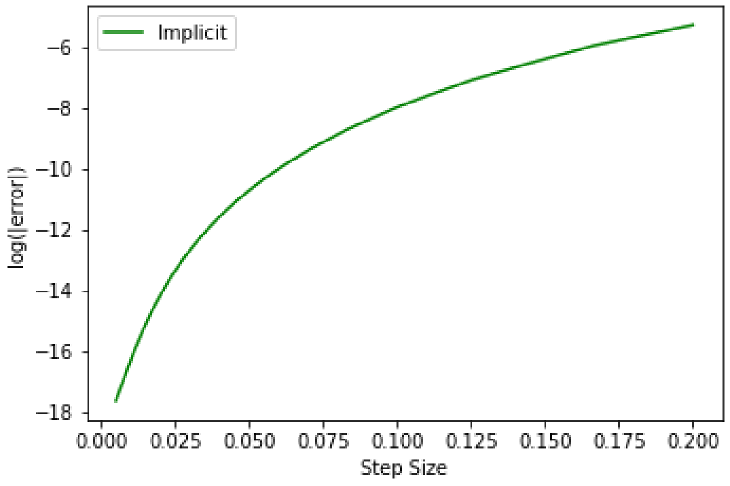


Figure 5: Error of the explicit Finite Difference Method on the Wave Equation.

Another significant difference is the *animate()* function. My understanding is that this function functions, however, it has proven impossible to test due to the fact that my Spyder refuses to behave like a GUI and actually display the animations. I tried to save the animation but this requires ffmpeg.

2.3 Elliptic

As with the other attempt to implement some kind of numerical solver for 2D PDE's, the challenge was to create a "tridiagonal" matrix of matrices. Unfortunately this proved not to be a viable solution so the Gauss-Seidel (GS) method has been implemented iteratively. This is inefficient but works well to arrive at a solution. When the Successive Over Relaxation (SOR) method is applied in addition to GS we can increase the accuracy of this method. When running just GS ($\omega = 1$) it takes this software 289 iterations to get within 10^{-4} between its second last and last approximations. However, when $\omega = 1.79$ this scheme satisfies the same error constraint in just 51 iterations. This value for ω is reasonable as the true value for the optimal ω is[5]

$$\omega \approx \frac{2}{1+\frac{\pi}{mx}} = 1.8$$

in the situation outlined in Worksheet 3. The solution can be seen in Figure 6.

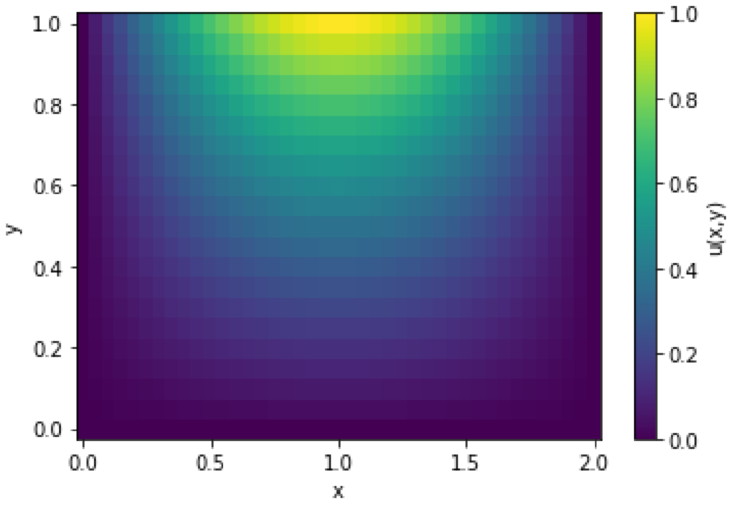


Figure 6: The solution fo Laplace's equation under the circumstances described in Worksheet 3.

3 Reflective Learning Log

This assignment has exposed me to a lot of similar material to Numerical Methods in Matlab but in a much more advanced way. I learned about a great number of methods for approximating the solution to PDE's that make a lot of intuitive sense and do a surprisingly good job, given their conceptual simplicity. I struggled most with the application of Boundary Conditions which proved to be less intuitive to me than the general solution. It sounds weird to say that I enjoyed the elegance of the vectorised method but I genuinely did appreciate how neatly it worked and how much it improved the code both aesthetically and in terms of efficiency. The sparse matrices will also no doubt prove to be very useful for an MDM project where we are working with matrices that may reach dimensions in the millions. These methods could well also prove to be immensely useful in an MDM setting as I know that some groups ended up having their projects revolving around solving a PDE these past few weeks.

I have learned about the benefits of git. I started off finding it to be slightly tedious and like an added complication that I did not understand and did not see a need for but after three weeks of editing code, parts of it stopped working and it was a great help to find that I could return to the version I had created days ago with minimal effort. I have since begun using it for a variety of different group projects in MDM and Machine Learning though I still have plenty to learn.

In future, I'm not totally sure what I would do differently. I kept up to date with the work as it was set, leaving myself time to go into some of the extensions such as the 2D solver for Parabolic equations. I think it would have been beneficial to actually go through the maths of why some of these methods worked because it should be simple and I'm sure that if I sat down and followed through a derivation it would click into place but some of the intricacies have fallen out of my head since it was first taught.

All in all this has been one of my favourite units which also came as a surprise to me because it has quite likely been one of the hardest and most work intensive and I'm a student; we only like complaining about work. I enjoyed the challenge and it was nice to feel like i was making progress when I spent time on the problems, not just bashing my head into the walls. The support from both lecturers and Oliver has been fantastic and I feel like I have learned a lot, which has served to illustrate that there's so much more to learn. Annoyingly.

References

[1] Wikipedia contributors. Heat equation — wikipedia, the free encyclopedia, 2017. [Online; accessed 21-December-2017].

[2] Wikipedia contributors. Laplace's equation — wikipedia, the free encyclopedia, 2017. [Online; accessed 21-December-2017].

[3] Wikipedia contributors. Wave equation — wikipedia, the free encyclopedia, 2017. [Online; accessed 21-December-2017].

[4] Seokkwan Yoon and Antony Jameson. Lower-upper symmetric-gauss-seidel method for the euler and navier-stokes equations. *AIAA journal*, 26(9):1025–1026, 1988.

[5] David M Young. A bound for the optimum relaxation factor for the successive overrelaxation method. *Numerische Mathematik*, 16(5):408–413, 1971.