# Scientific Computing - Partial Differential Equations (PDEs)

Jack Healey

December 13, 2019

University of Bristol

## 1 Aim

The process is centred around solving parabolic PDE problems, using the finite difference method and its various schemes. The numerical methods addressed include the forward Euler scheme, the backward Euler scheme, and the Crank-Nicholson method. The performance of each method is analysed, showing how each method ranks against the others. The order of the truncation error for each method was chosen for this purpose. The aim was to make the solver adaptable to a variety of conditions and other potential factors.

## 2 Process

Many methods are available to solve particular PDE problems. The following steps were undertaken in this method:

1. Modularise the skeleton forward Euler code.

   The skeleton code provided the foundation of how to solve a parabolic PDE with homogeneous Dirichlet boundary conditions in a bespoke manner. To improve the code's adaptability and functionality, the code was modularised into smaller individual functions, where ideally each function has its own task. This helps the user alter certain inputs for example the initial temperature distribution or the known solution, and also debug the code with greater ease should an error arise. It is also good practise to avoid the use of global variables, to generalise the code for wider, less problem-specific use, and this can be seen throughout the submitted code. The result of this step was the function 'myForwardEuler1DDiffusion.py'.

2. Modify the code for alternate numerical schemes.

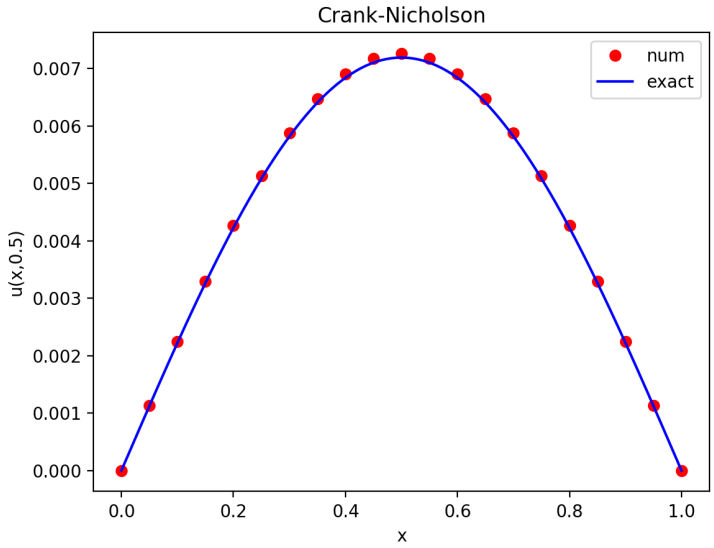
$$\text{Forward Euler method stable for:} \quad 0 < \lambda < \frac{1}{2} \quad \Longleftrightarrow \quad \kappa \frac{\Delta t}{\Delta x^2} < \frac{1}{2} \tag{1}$$

   It is known that the forward Euler method is an explicit, conditionally stable scheme (eq. 1). Therefore, other methods should be explored in order to find an optimum method for the solver. The code was modified for the backward Euler method and the Crank-Nicholson method by studying the alternatives and deriving equations similar

to that used for the forward Euler case. The result of this step was the function 'myBackwardEuler.py'.

3. Modify numerical methods for greater efficiency.

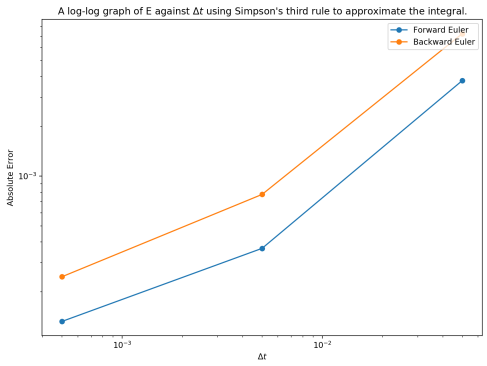
   Once the solutions of the two alternatives were compared to the forward Euler case and checked for accuracy, more intricate modifications could be made to improve efficiency and functionality. With a greater understanding of the numerical methods, certain processes could be enhanced. Examples of this include the use of sparse matrix operations when manipulating tridiagonal matrices, and the performance of a coded Thomas algorithm over python's inbuilt 'linalg.solve' function in the same scenario. These benefit the functionality of the code as time is not wasted running over zeros in a large matrix, improving the efficiency overall. Having implemented the mentioned modifications, the solution using the Crank-Nicholson method could be visualised against the exact solution (below). The accuracy for this can be seen to be satisfactory.



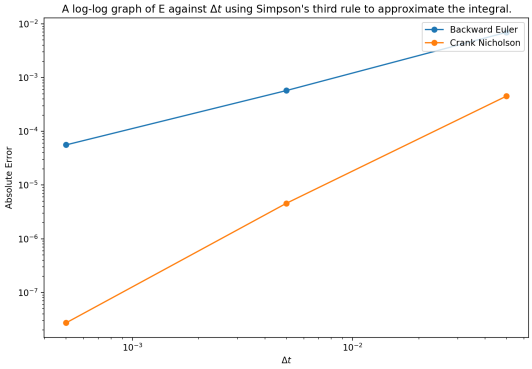

4. Compare the order the truncation error to find an optimum method.

   The performance of the improved methods can then be compared using the difference in their individual results and the known solution. The method chosen was to find the areas under the two graphs and calculate the absolute difference at different values of $\Delta t$. The absolute error can then be plotted for varying $\Delta t$ in a loglog plot, where the gradient of this provides the order of the truncation error for each method. The file 'Errors.py' was used to carry out this method for PDEs with homogeneous Dirichlet boundary conditions, as the exact solution was known.

   It can be seen from the figures below that the orders of the truncation error for the forward and backward Euler methods are equal, as they behave the same with varying $\Delta t$. This order can be calculated to $\mathcal{O}(\Delta t)$ with varying $\Delta t$. The two alternative schemes can then be compared, where the Crank-Nicholson method can be seen to have a superior order of $\mathcal{O}(\Delta t^2)$ with varying $\Delta t$, performing better than both the forward and backward Euler methods. To produce a solver that performs better with




Forward Euler vs. Backward Euler




Backward Euler vs. Crank-Nicholson

smaller time points, the Crank-Nicholson method should be used. This method is therefore used throughout the rest of the process (the file 'myCrank_Nicholson.py').

5. Extend scope of the optimum method for different possible boundary conditions and additional factors

   The optimum method can then be adapted to widen the scope of its abilities. This includes the ability to handle different boundary conditions, heat sources, and variable diffusion coefficients. The following extensions can be handled by the code:

   - Homogeneous Dirichlet boundary conditions
   - Non-homogeneous Dirichlet boundary conditions
   - Homogeneous (or not) Neumann boundary conditions
   - Heat sources inside the domain
   - Non-constant diffusion coefficient

   The user interface is improved as a result of this as it widens the range of parabolic PDE problems that can be solved by the code. This requires user input therefore there is a healthy amount of usage instructions necessary in the form of docstrings.

6. Final code improvements

   The final steps include checking the docstrings include sufficient information for a third party user to digest the code, in addition to further functionality edits where available, and deleting unnecessary code.

## 3 Summary of Software

In terms of using the code to solve a specific PDE problem, the files to focus on are 'myCrank_Nicholson.py' and 'usage.py'. When running the code for a specific PDE problem load 'usage.py' and input the relevant functions with the help of the usage docstrings. Then it is possible to input the corresponding variables to the main solver function. When run, this produces the solved PDE along with the mesh points in space, allowing the user to plot the solved results.

### 3.1 Format

The two files mentioned feed into each other, but are split up for the optimal user experience. It was designed so that the functions are split between ones that require interaction, and ones that do not. The 'myCrank_Nicholson.py' file contains the "behind the scenes" work that requires no editing. This file is essentially the cogs behind the machine, where the functions found here are called in the 'usage.py' to neaten the code and improve the modularisation of the whole work. The functions in the file 'usage.py' contains mostly files that directly require user input. The functions built for user input include:

1. The known solution (if known).
2. The initial temperature distribution.
3. The function for a heat source inside the domain.
4. The function for a variable diffusion coefficient.

The remaining functions in the 'usage.py' file are not made to be altered by the user, however they were deemed essential in helping the user understand how the solver works, and how the inputted functions are used in the process. The functions 'lmbda_calc' and 'tridiag_kappa' depend heavily on the function for a variable diffusion coefficient, and the 'Crank_Nicholson_solver' function wraps up the entire work into a final solution, and contains the usage text for when calling the solver.

### 3.2 Input

The 'usage.py' script assumes the user has a certain level of prior knowledge about the PDE they wish to solve. To produce an accurate result, the 'Crank_Nicholson_solver' function must be fed:

1. The initial temperature distribution: **u_I**

   The function defined previously in the file.
2. The number of grid spaces in space and time: **mx**, **mt**
3. The size of the domain: **L**
4. The desired computation time: **T**
5. The boundary conditions (scalar or function): **u_0**, **u_T**

   Either a scalar, for example where **u_0**, **u_T** = 0, or a lambda function, such as '**u_0** = lambda t_j : np.sin(pi*2*t_j)'.
6. The label for the boundary condition: **bCond**

   Options include 'D_hom', 'D_non_hom', and 'Neum'.

### 3.3 Functionality

The focus when developing the code was to maintain modularisation that compartmentalised large sections of code yet also comprehensible for the user. The aim for the functionality was for each function to have an individual task, where these functions may interact. This is also the reason the code was split into the two files. The individual functions are then combined in the main solver function 'myCrank_Nicholson.py'

The main functionality changes from the original file arose from incorporating the extensions into the code.

### 3.3.1 Boundary conditions

To include the option of different boundary conditions, the method chosen was to split the code into three branches using 'if' statements, detailing:

1. Homogeneous Dirichlet boundary conditions
2. Non-homogeneous Dirichlet boundary conditions
3. Homogeneous (or not) Neumann boundary conditions

This method was chosen as each route has its unique calculation, where each calculation is concise, therefore this method was not deemed too costly in terms of efficiency.

### 3.3.2 Heat source in the domain

The concept of a potential heat source in the domain was taken into account by using the formula below (eq. 2). With this method the heat source could be predefined and added to the right hand side of the equation of all boundary condition options. If no heat source exists, usage instructions in the heat source equation help equate y[$i$] to zero, which has no effect on the rest of the system, which is a good sign in terms of functionality.

$$\mathbf{A_{CN}}\ \mathbf{u}^{j+1} = \mathbf{B_{CN}}\ \mathbf{u}^{j} + \frac{\Delta t}{2}(\mathbf{F}^{j} + \mathbf{F}^{j+1}) \tag{2}$$

Calculating the heat source may lead to losses in efficiency when there is no heat source, as a vector for its value of zero is still created. This was deemed necessary however, as the accuracy and adaptability benefits of widening the scope of the solver out way the relatively minor efficiency losses of its calculation.

### 3.3.3 Variable diffusion coefficient

In order to take a variable diffusion coefficient into account, the parameter $\lambda$ had to be vectorised, where

$$\underline{\lambda} = \kappa(\underline{\mathbf{x}}\,)\frac{\Delta t}{\Delta x^2}. \tag{3}$$

The adaptability of the code is improved here as there is the option to leave $\kappa(x)$ equal to a constant, such as $\kappa = 1.0$, leaving the regular Crank-Nicholson method. This increases the storage of the code however once more the range of the solver is prioritised over storage issues that are deemed more insignificant.

### 3.3.4 Usage and docstrings

The functionality changes above require some user input. Therefore, docstrings are necessary to guide the user through the process, so that reaching a solution is a smooth process. The functions in the 'myCrank_Nicholson.py' file require no maintenance so any commenting or docstrings are solely to help the user understand the process. In 'usage.py', the docstrings are extensive yet necessary as the inputs are vital to solving a given problem with ease.

When the user inputs the functions that represent their specific problem, as well as the suitable variable values, the result is the solved PDE along with the mesh points in space, allowing the user to plot the solved results.

# 4 Key Software Design Decisions

The following section details the main decisions made during the process that would affect the efficiency or accuracy of the code when producing a solution to a given PDE problem.

## 4.1 Focus on the Crank-Nicholson method

After modularising the skeleton forward Euler code and the alternate numerical schemes for a suitable level of efficiency, a decision was made to analyse the performance of the different numerical methods. A combination of prior knowledge and research led to the use of the order of the truncation error in differentiating the three schemes. The Crank-Nicholson method was found to perform the best with varying $\Delta t$ (shown in the figures referenced previously). Therefore, the decision was made that it was unnecessary to develop all three methods to a complex extent, as similar results would be produced but with reduced accuracy. This would also create an unnecessary amount of code, where the user could get lost unless handled with great caution. The result of this was that, having discovered the Crank-Nicholson method was the most accurate, only this solver would be used to solve the most complex PDEs.

## 4.2 Error calculation

When deciphering the numerical method that performs the best in this process, the order of the truncation error was evaluated. Initially, a Euclidean distance function was created, producing results that appeared marginally inaccurate. As 'mx' was increased, the error of an individual point decreased, however the number of points increased, meaning that with increasing 'mx' there were benefits and costs when producing that error in a visual. The Euclidean distance function was replaced with a Simpson's third rule function. Unlike the Euclidean distance, the accuracy of the process improves with more 'mx' grid points in space. This improved the visualisation and led to the observation that Crank-Nicholson had a truncation error order of $\mathcal{O}(\Delta t^2$ compared to $\mathcal{O}(\Delta t^2$ of the backward and forward Euler methods, as $\Delta t$ was varied.

## 4.3 Linear matrix equation solver

When dealing with linear matrix equations, it is common to use the inbuilt python function 'linalg.solve'. In this process however, the matrices being dealt with on the left hand side of the equation ($A_{CN}$ for Crank-Nicholson) are tridiagonal.

$$A_{CN} = \begin{pmatrix} 1+\lambda & -\frac{\lambda}{2} & & 0 \\ -\frac{\lambda}{2} & \ddots & \ddots & \\ & \ddots & \ddots & -\frac{\lambda}{2} \\ 0 & & -\frac{\lambda}{2} & 1+\lambda \end{pmatrix}, \qquad B_{CN} = \begin{pmatrix} 1-\lambda & \frac{\lambda}{2} & & 0 \\ \frac{\lambda}{2} & \ddots & \ddots & \\ & \ddots & \ddots & \frac{\lambda}{2} \\ 0 & & \frac{\lambda}{2} & 1-\lambda \end{pmatrix}.$$

This opens up the opportunity to use sparse matrix operations. Efficiency is boosted here as the use of sparse matrix operations can reduce storage by factors of hundreds. Two options were considered:

1. In-built python function 'sparse.linalg.spsolve'.

2. A coded Thomas algorithm solution.

If the left hand side matrix $A_{CN}$ is tridiagonal (as is the case in the Crank-Nicholson method), the Thomas algorithm finds a solution in operations of order equal to the number of elements in the matrix. For this reason, the Thomas algorithm was selected as the go to linear matrix equation solver.

### 4.4 Calculation of the tridiagonal matrix

The storage of the calculation of the tridiagonal matrix could be reduced in the same way as the linear matrix equation solver through sparse matrix operations. The use of the in-built python function 'sparse.diags' provides a tridiagonal matrix with constant diagonals which is generally useful. When adapting the code to incorporate the variable diffusion coefficient, the function 'tridiag_kappa' was coded to take its place in order to understand how the variable diffusion coefficient affects the calculation. The in-built function here is beneficial due to efficiency reasons however in order to widen the scope of the solver, the coded function was preferred. With more time these two ides could be combined. The decision was made that increasing the scope and accuracy of the solver with various problems was more important than the storage usage.

### 4.5 Tackling boundary conditions

In terms of dealing with different boundary conditions, two options were considered:

1. Separate functions for each condition.

2. Three branches 'if' statement.

The benefits of the separate functions method are that the code is more modularised, and the code in the 'if' loop looks neater. The second method where the boundary condition methods are detailed in the loop was preferred. The reason for this is when the method is listed underneath the boundary condition label in the 'if' loop, the structure is easier to follow. In addition, the code that comes under each section is not extensive so the concise nature of the rest of the code is maintained.

### 4.6 Adding complexity of the extensions

Incorporating additional extensions into a solver decreases the speed of a solver and increases the storage it requires. The decision here therefore is whether the benefits of the extensions out way the aforementioned costs. The benefits include the solver having the ability to handle more complex PDE problems. The ability to solve these problems rather than produce errors is vital in a complex functional PDE solver. For this reason, it was the aim in this process to stretch the complicity of the solver to its limits, and take on the computational costs where necessary.

# 5 Reflective Learning Log

| | |
|---|---|
| What did I learn about the mathematical algorithms? | The most important concept that was learnt was maintaining generalised code. When using the Thomas solver, keeping the values a, b, c, and d constant meant that errors arose with non-homogeneous Dirichlet boundary conditions and also when dealing with the variable diffusion coefficient. The code for the Thomas algorithm then had to be changed to loop over the input matrix. Initially I thought the calculations used for incorporating boundary conditions in the forward Euler scheme would be mirrored in the Crank-Nicholson method, however the latter must be derived using a similar process to produce a slightly different result. In addition, when solving PDEs with coded numerical methods, accuracy is not the only considerable factor. The Thomas algorithm and the python in-built sparse matrix operations are beneficial in this process to standard linear matrix equation practises. This is due to their greater efficiency in storage use. |
| What did I learn about software engineering? | In coding projects prior to this one, the code and results have taken precedence over user interface. That has changed after this unit, as it is vital the user is able to understand how to use the provided code when adapting your generalised code to solve their own PDE. I also learnt the benefits of testing framework to maintain performance throughout the process of editing code, where in this project it was difficult to test any results other than the homogeneous Dirichlet boundary conditions. Therefore, tests were limited, but can be found in 'test.py'. Modularisation of the code makes it easier to debug code should an error arise, improving functionality for the coder and the user alike. Version control software such as Git has been useful in keeping track of small checkpoints and milestones, so when picking up work after a break it is easy to find the next step. Furthermore, I have improved the functionality of my code, in terms of using no global variables and the modularisation of my code. |
| What are the implications of what I've learnt? | In the short term, this project has given me improved coding abilities, with a wider focus on usability and functionality, along with accuracy of results. These skills are easily transferable to other modules such as Mathematical and Data Modelling, where PDEs are also addressed. In the long run, these skills improve my employment prospects as I am now able to handle more complex coding tasks with a more professional structure in terms of providing a client interface. |

| | |
|---|---|
| What would I have done differently if I started the unit over again/in the future? | Although a strong relationship with Git was maintained throughout, the commits could be more detailed. A third party user may not understand the syntax in some of the commits, as they relate to specific updates made to the code. To prevent confusion in this case, a commit could explain generally what has been altered, and then give more specific details on changes to certain functions. When adding a new function, a docstring should be produced straight away to detail its functionality. It is easy to get carried away with coding and lose the user interface, so in the future it would be beneficial to maintain this balance. |