SciComp Coursework 2

Max Oblein

mo17165@bristol.ac.uk

December 12, 2019

1 Summary of Software

This package provides functions to allow the user to compute numerical solutions to the one dimensional (1-D) heat diffusion equation

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \tag{1}$$

using a finite difference method.

1.1 Finite Difference

This software implements the forward difference, backward difference and Crank-Nicolson schemes. They are used to solve equations with both Dirichlet and Neumann boundary conditions in the form of constants or functions.

To compute the solution the space and time domains of the system are separated into a 'grid' with $mx + 1$ points in space and $mt + 1$ points in time where mx and mt are inputs to the function. The program then uses an input of the initial temperature distribution in space and iterates this vector forward in time using the chosen finite difference scheme. The function has an optional boolean argument `Plot` which if true gives a plot of the temperature distribution at time T as shown below in Figure 1.

The forward difference method is the simplest to implement as it is an explicit technique. That is to say the temperature distribution at the next time-step can be calculated directly from known quantities. The backward difference and Crank-Nicolson schemes are slightly harder to formulate due to them being implicit techniques. This means a matrix linear solve is needed to compute the updated temperature distribution. However due to the better error traits of Crank-Nicolson its higher computational cost is justified.

One other problem with forward difference is its lack of stability as an explicit function. The scheme is only stable for

$$\lambda < 0.5, \tag{2}$$

where

$$\lambda = \kappa \frac{\Delta t}{\Delta x^2}. \tag{3}$$

This means that to use the faster forward difference method the user must carefully choose mx and mt this is not an issue for either of the implicit schemes which is why they are more widely used.

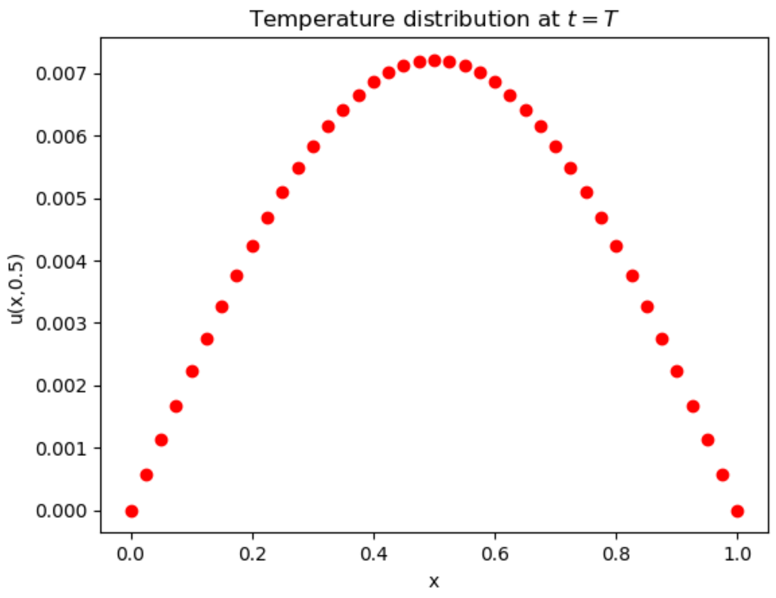


Figure 1: Example solution found using the Crank-Nicolson finite difference scheme. Generated using an initial temperature distribution of $\sin\frac{\pi x}{L}$ and boundary conditions $u(0, t) = 0$ and $u(L, t) = 0$

1.2 Test File

To test this package a simple test file is provided. Initially tests are run to check the output of `Finite_Difference` against the provided exact solution. The test checks that the error between the approximate solution and the exact solution is lower than a specified tolerance. This tolerance is higher for forward difference as, due to its instability, it is only run with 10 points in space. The other methods are run with 40 points in space and all are run with 1000 points in time. Next a test is run to check that the code performs correctly when forward difference is run with mx and mt values that would lead to instability as this could result in the user receiving a solution that does not converge. The function should print an error message and return $[1, 1]$ for this test to pass. The next test is to check that the user has inputted a correct type for both boundary conditions, once again this test only passes if the code prints an error message and returns $[1, 1]$. Finally the code is tested with an incorrect method such as ‘fu’ again the code should print an error message and return $[1, 1]$ for the test to pass.

1.3 Demo File

Provided with this software is a very simple demo file to show some of the basic functionality of the package. The file is run on the command line with either ‘FD’ to produce the plot in Figure 1 or with ‘Error’ to produce the plot in Figure 2a. The function calls in this demo file along with the docstrings in the package should allow the user to effectively use the code.

2 Usage

The main function provide by this package is `Finite_Difference()` its inputs and outputs are shown below in Table 1.

Input	Description
`initial_cond`	The initial temperature distribution (callable).
`bc`	The boundary conditions for the problem ([LHS,RHS] can be int, float or callable).
`mx`	The number of grid points in space (int).
`mt`	The number of grid points in time (int).
`params`	The parameters of the problem (Tuple (kappa,L,T)).
`Method`	The finite difference scheme to be used (str). defaults to 'cn' Crank-Nicolson also available 'fd' and 'bd'.
`b_type`	The type of each boundary condition, 0 for Dirichlet and 1 for Neumann ([LHS,RHS]) default to [0,0].
`u_exact`	Optional. The exact solution to the problem (callable) default is None.
`plot`	Optional. Boolean argument as to whether to show a plot of the solution at time $t = T$.
Outputs	
`u_T`	Solution at time $t = T$ (ndarray).
`diagnostics`	System parameters useful for error analysis ([Δx, Δt, λ]).

Table 1: Usage for `Finite_Difference()`.

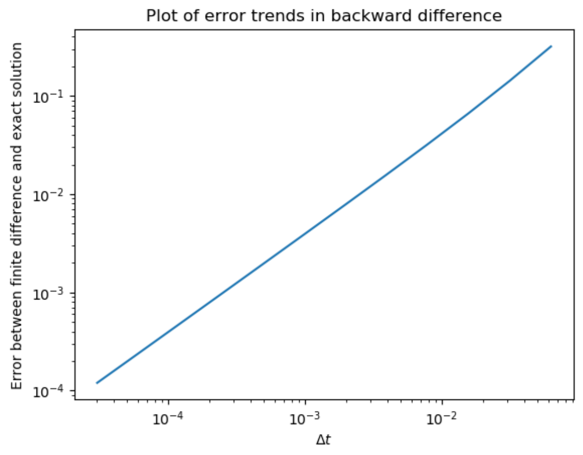
3 Error Analysis

As with all numerical analysis errors in the discretisation of the problem will give rise to errors within the result for a given problem. For the finite difference schemes provided in this software package the truncation errors are as follows:

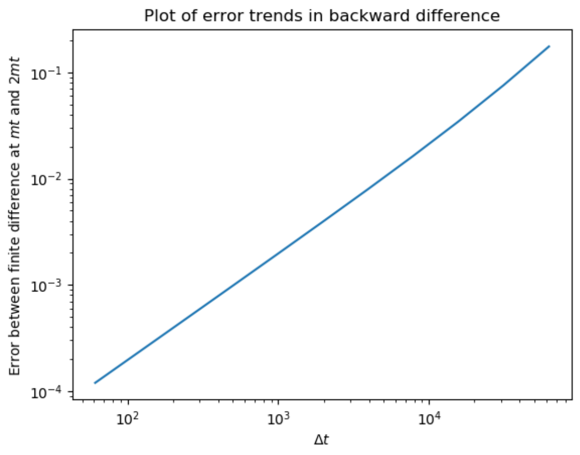
Scheme	Error
Forward difference	$\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2)$
Backward difference	$\mathcal{O}(\Delta t) + \mathcal{O}(\Delta x^2)$
Crank-Nicolson	$\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^2)$

Table 2: Error contributions of Δt and Δx for all schemes.

The errors for all schemes are plotted on a log-log graph for increasing mt, which gives decreasing Δt. The error can be calculated with respect to an exact solution (if provided) or as the difference between successive iterations of the scheme. In Figure 2 the error plots with and without an exact solution can be seen. Since the plots are on a log-log scale the gradient represents the order of the error contribution both are very close to 1 so agree with the known error which is proportional to Δt for the scheme.



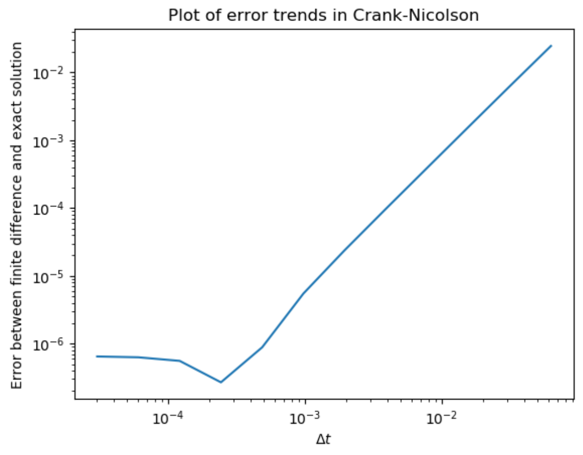
(a) Error plot when an exact solution is given. Gradient = 1.025.



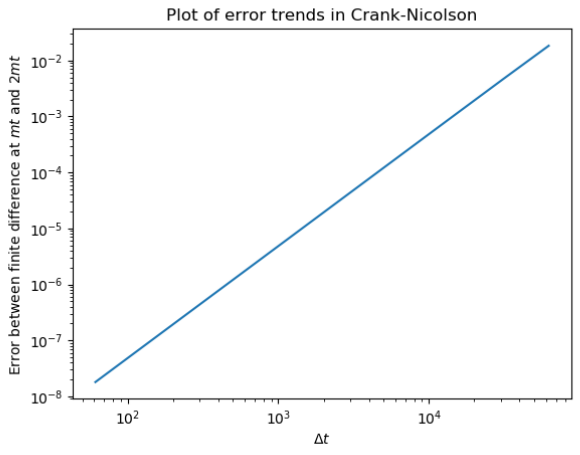
(b) Error plot when an exact solution is not given. Gradient = 1.041.

Figure 2: Error trends for varying mt with the backward difference scheme.

The Crank-Nicolson scheme has a better error with respect to Δt than the other two schemes, its error plot using successive iterations in Figure 3b shows a gradient of almost 2 which makes sense as this scheme has an error proportional to Δt^2. The plot for error between successive iterations is the best representation of the error trends here as the error will continue to decrease as Δt decreases. The anomaly on Figure 3a is due to the dominance of the constant error contribution of Δx compared with the near zero contribution from Δt. This gives rise to constant error between the approximation and the exact solution. This same property would show in Figure 2a if the scheme was run for more iterations but since backwards difference converges slower this point is not reached for the range of n used.



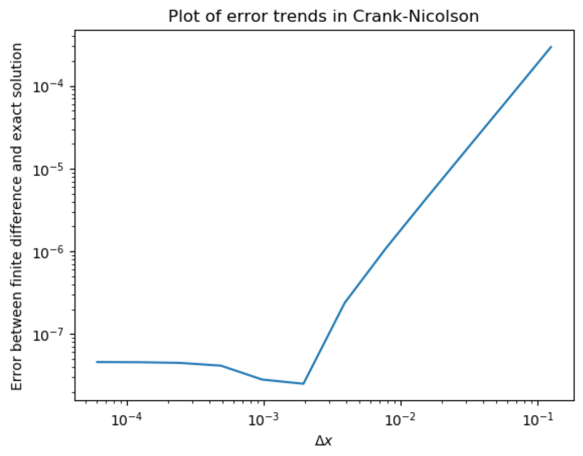
(a) Error plot when an exact solution is given. Gradient undefined.



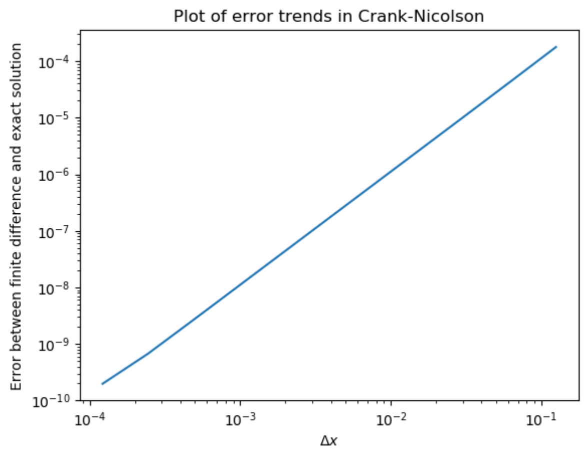
(b) Error plot when an exact solution is not given. Gradient = 1.998.

Figure 3: Error trends for varying mt with the Crank-Nicolson scheme.

All the schemes provided in this software have an error proportional to Δx^2 therefore only the trend for Crank-Nicolson is plotted in Figure 4. Once again the error against previous iterations shows the correct trend with a gradient of approximately 2 which agrees with the known relationship that the error is proportional to Δx^2. However the error against the exact solution is again constant after a point as the error due to the constant Δt begins to outweigh the error contribution from the varying Δx leaving a constant error between the approximation and the exact solution even as Δx is decreased further.



(a) Error plot when an exact solution is given. Gradient undefined.



(b) Error plot when an exact solution is not given. Gradient = 1.989.

Figure 4: Error trends for varying mx with the Crank-Nicolson scheme.

4 Design Decisions

Throughout the project the software was developed numerous times after design choices had been made to either speed up the code or cut down on repeated code within the package.

4.1 Finite Difference Methods

The skeleton code provided for the project implemented the forward difference scheme in an element-wise format. This was then converted to a matrix form to save on computation time and to allow the code to be adapted to use both the backward difference and Crank-Nicolson schemes as these schemes are implicit and hence need a matrix solve to update the temperature distribution at each time-step. This was implemented using sparse matrices because all matrices for finite difference methods are tridiagonal. The use of sparse matrices allows for faster running code and less memory use.

The package originally involved three different functions, one for each of the three schemes to be used. However as the software developed these were condensed into one function. To achieve this the update step for was moved into its own function and the method is used to decide which type of update step to use. This did however mean creating a function to initialise the correct matrices for each method and then input these into the solver. After this I decided to make the method an optional input with a default as Crank-Nicolson as this method has the best convergence rate. All of this was done to reduce the amount of repeated code in the package which is good coding practice.

To interpret the output of `Finite_Difference()` it is highly beneficial to have a plot of the solution at time T therefore a boolean option `plot` was added to allow the user the option to visualise it. This option defaults to `False` as when running the function for multiple iterations plots should not be shown for example when calculating the errors for Section 3 the function is called multiple times and having to close a plot each time would be highly inconvenient.

To implement the different boundary condition types an input of `b_type` was added for the user to specify either Dirichlet (0) or Neumann (1). This meant that the decision to condense the code into one function for all methods saved a lot of repeated code as each boundary condition combination needed to be individually specified. These conditions could also be integers, floats or functions. Therefore, it was necessary to get them into the same form, functions were created to represent both integer and float boundary conditions, to be used later in the code and for further generalisation of the method.

4.2 Error Analysis

When analysing the errors of the methods provided in this package I decided to start with varying Δt as this gives a different trend for both backward difference and Crank-Nicolson. This was achieved by running `Finite_Difference()` with

$$mt = 2^n \text{ for } n \in [3, 15]. \tag{4}$$

This range of n was chosen to eliminate trivial cases of too few grid points at the low end and to avoid an unnecessarily high computational cost at the high end. I also chose to exclude forward difference from any error analysis as it has only a small range of λ, determined by mx and mt, in which the method is stable and this made error analysis difficult.

Originally all analysis was done in reference to a known solution. In practice one rarely has a true solution when performing numerical methods. Therefore, the code was adapted to also calculate the error between successive iterations of the function. This method does well at demonstrating the expected trends in the error whereas it doesn't show the convergence to the true solution. So only really gives the user an idea of how increasing grid density affects error.

I then developed a second function to analyse the error in varying Δx this was done in a similar way to varying Δt in that `Finite_Difference()` was run with

$$mx = 2^n \text{ for } n \in [3, 15]. \tag{5}$$

This was separated from the calculations for varying Δt as in every iteration the number of points in the solutions vector changes meaning that extra care must be taken when calculating error between successive iterations. This challenge was overcome by comparing the trapezium rule integral for each iteration using its Δx value.

5 Reflective Learning Log

In the last project the main point for improvement I highlighted was the planning process involved in writing code. So in this project I took extra care to plan the structure of my code before writing it. This meant that the process of generalising the code was much faster and smoother.

In future MDM projects it is highly likely that I will have to deal with PDE systems. Therefore, the numerical approaches I have learnt in this project will be useful to calculate approximate solutions where it is infeasible to find an analytic solution.

I also learnt the importance of choosing the right scheme due to its error properties. Initially forward difference seemed the easiest method to implement. However, when looking at the errors for different methods it was clear that for the little extra effort the Crank-Nicolson scheme provides a more accurate approximation to the solution.

Also when trying to generalise my code I found that it is sometimes better to be less general whilst maintaining a good computational efficiency. I had thought that I could generalise the update step and do a linear solve for all schemes, however in practice this needlessly increases the computational time for the forward difference method.

When using version control in this project I aimed to have a good frequency of commits. This could have been improve by committing smaller changes more frequently. I also made good use of the ability to revert to old commits as in some cases I broke my code and it was extremely helpful to reset to the last working iteration of my code. In future I would like to have a deeper understanding of git and its features possibly looking at branching.

This project has given me further understanding of numerical methods and the challenges with extracting useful results from the program. One must understand the error properties of a given algorithm before deciding to use its result. As far as software development, this project highlighted similar issues to the last one in that writing general code is difficult and requires lots of planning. I believe this time round the planning I did made for a much smoother coding process.