Computational Partial Differential Equations
Project 1: Diffusion Problem

Tudor Trita Trita
CID 01199397

March 19, 2020

This is my own work unless stated otherwise.

Structure of Coursework:

The coursework .zip file contains the following:

- (DIR) code:
  - partA.py: Code used to generate figures and output for Part A.
  - partB.py: Code used to generate figures and output for Part B.
  - partC.py: Code used to generate figures and output for Part C.
  - schemes.py: File containing all the schemes used in the coursework. Each of these is located in its own function. See docstring for function names.
- (DIR) figures: Folder containing all the figures used in the report stored as .png files.
- Proj1_01199397.pdf: (This document) Compiled LaTeX report.

Software Requirements:

The code for the coursework has been written in Python 3.7 and the following third-party packages have been used:

- Numpy (1.18.1): Used for access to fast array operations.
- Scipy (1.4.1): Used for access to sparse matrices.
- Matplotlib (3.1.3): Used for plotting capabilities.

1 Part A

1.1 Analytical Solution of the Diffusion Equation

The problem at hand is given by the differential equation

$$u_t = u_{xx}, \qquad 0 \leq x \leq 1, \tag{1}$$

subject to the initial condition $u(x,0) = 1$ for $0 < x < 1$ and the boundary conditions $u(0,t) = 0, u(1,t) = 0$ for $t \geq 0$.

We wish to find a separation of variables solution to this problem. This solution will take the form $u(x,t) = X(x)T(t)$ for functions $X(x)$ and $T(t)$ respectively depending on x and t only.

If we plug this form into equation 1, we find that $u(x,t) = X(x)T(t)$ is a solution to the heat equation iff $X(x)T'(t) = X''(x)T(t)$ or equivalently

$$\frac{X''(x)}{X(x)} = \frac{T'(t)}{T(t)}. \tag{2}$$

Upon closer inspection, we see that the LHS of equation 2 is independent of t and the RHS of equation 2 is independent of x. Hence, both sides are equal to some constant written in the form $-\lambda^2$ for convenience and we can obtain solutions to the ODEs separately.

The LHS of equation 2 can be rearranged to give $X''(x) = -\lambda^2 X(x)$, a second-order ODE with general solution of the form $X(x) = A_1 \cos(\lambda x) + A_2 \sin(\lambda x)$. Imposing the boundary conditions, we deduce that $A_1 = 0$ and $\lambda = m\pi/L$, for m any positive integer. The boundary conditions dictate that for positive λ, the solution is exactly zero, so we can ignore these in the summation term. Hence we can write this part of the solution as $X(x) = \sum_{m=1}^{\infty} = A_m \sin(m\pi x)$.

Turning to the RHS of equation 2, and substituting our form of lambda, we rearrange to get the equation $T'(t) = -T(t)\lambda^2 = -T(t)m^2\pi^2$, which we recognise it is a first-order ODE with general solution $T(t) = B\exp(-m^2\pi^2 t)$.

Plugging the forms of $X(x), T(t)$ into equation 2, and letting $c_m = A_m B$ and summing over all the possible values, we arrive at the solution.

$$u(x,t) = \sum_{m=1}^{\infty} c_m \sin(m\pi x)\exp(-m^2\pi^2 t) \tag{3}$$

We now use the initial condition giving the equation

$$u(x,0) = 1 = \sum_{m=1}^{\infty} c_m \sin(m\pi x)$$

This is a half-range Fourier series which we can can solve for c_m with the following formula

$$
\begin{aligned}
c_m &= 2\int_0^1 1 \cdot sin(m\pi x)dx, \\
&= 2\left[-\frac{\cos(m\pi x)}{m\pi}\right]_0^1, \\
&= 2\left(\frac{(-1)^m}{m\pi} + \frac{1}{m}\right).
\end{aligned}
$$

This expression is equal to zero if m is even and $4/m\pi$ if m is odd. Hence, c_m is non-zero iff m is odd, so we let $m = 2n+1$ for any integer $n \geq 0$. Hence our coefficients become

$$
c_{2n+1} = \frac{4}{(2n+1)\pi}.
$$

Substituting this back into equation 3, we arrive at the solution given in the notes

$$
\begin{aligned}
u(x,t) &= \sum_{n=0}^{\infty} \frac{4}{(2n+1)\pi} \sin((2n+1)\pi x) \exp(-(2n+1)^2\pi^2 t), \\
&= \frac{4}{\pi}\sum_{n=0}^{\infty} \frac{1}{(2n+1)} \exp(-(2n+1)^2\pi^2 t) \sin((2n+1)\pi x).
\end{aligned}
$$

1.2 Discretisation using the Explicit Scheme

Defining $h = \Delta(x)$, the x-discretisation step size and $k = \Delta t$, the time interval used in marching to our solution, we can use the following central differences for the first-order time derivative as well as the second-order space derivative

$$
u_t = \frac{U_n^{j+1} - U_n^j}{k} + O(k), \tag{4}
$$

$$
u_{xx} = \frac{U_{n+1}^j - 2U_n^j + U_{n-1}^j}{h^2} + O(h^2). \tag{5}
$$

Combining equations 4 and 5, we obtain the following approximation to the diffusion equation with $O(h^2), O(k)$ accuracies:

$$
\frac{U_n^{j+1} - U_n^j}{k} = \frac{U_{n+1}^j - 2U_n^j + U_{n-1}^j}{h^2}. \tag{6}
$$

Rearranging for U_n^{j+1} and letting $r = k/h^2$ we obtain the second-order accurate explicit scheme formula

$$
U_n^{j+1} = r(U_{n+1}^j + U_{n-1}^j) + (1-2r)U_n^j. \tag{7}
$$

From lectures, we know that the scheme is numerically stable provided that $r \leq 1/2$.

This scheme has been implemented in Python and the function that computes the solution using this scheme can be found inside the file `schemes.py` in `diffusion_explicit_scheme`.

In the implementation, we make use of an (N - 2) x (N - 2) sparse tridiagonal Toeplitz matrix, say M_1, with coefficients $(1 - 2r)$ on the main diagonal and r on the diagonals immediately above and below. To calculate the next time-step, using the fact that at the boundaries $U_1, U_N = 0$, we can use matrix-vector multiplication $U^{j+1}_{2:N-1} = M_1 U^j_{2:N-1}$ where $U^k_{2:N-1} = (U^k_2, U^k_3, \ldots, U^k_{N-2}, U^k_{N-1})^T$. The module Scipy provides sparse matrices in Python (csr_matrix) with fast matrix-vector multiplication operation.

1.3 Investigation of the Explicit Scheme

In this question, we are going to investigate the performance of the explicit FD scheme with respect to the analytic solution. Precisely, we are interested in how the quantity

$$error = \log_{10} |u_{numerical} - u_{analytic}|$$

changes on varying r, k and h at both $x = 0.02, x = 0.5$ at the solution time $t = 0.075$.

From lectures, we know that the scheme is of order $O(h^2), O(k)$, and this means that we can make our error arbitrarily small, up to round-off error due to floating point arithmetic, as long as we make h, k small enough.

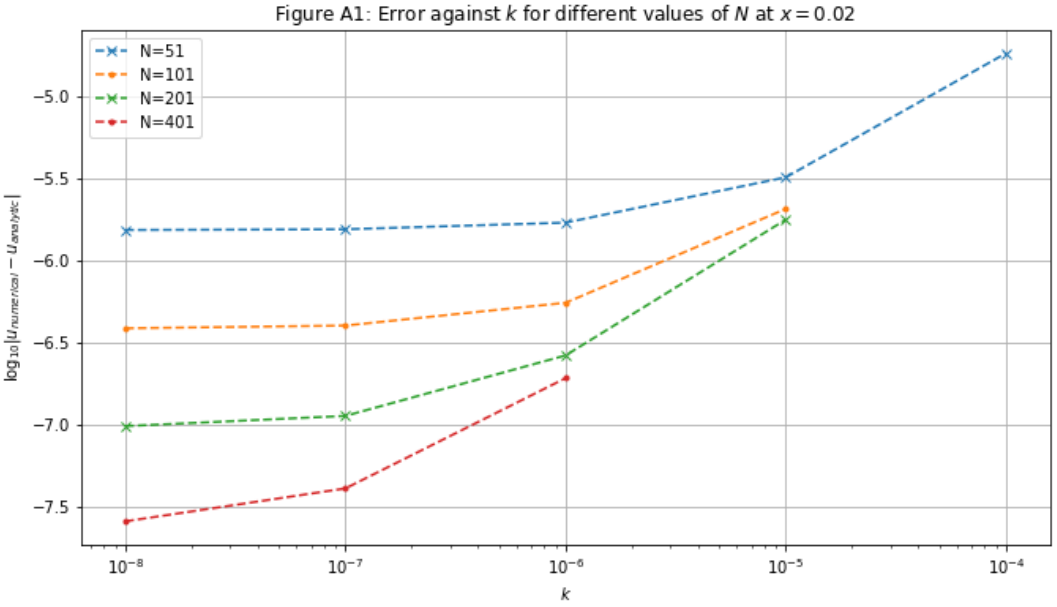


Figure A1 shows the relationship between k, N and the *error* at $x = 0.02$. Each of the individual lines represent taking a larger N, hence a smaller h. We can see that for low enough k, the error plateaus, indicating that the the error is now mainly due to the dependence on h. Taking $N = 51, h = 0.02, k = 10^{-8}$, should theoretically make the scheme have an error

of approximately 10^{-4}, but we see that the error is lower due to the proximity to the known boundary. The relationship that the accuracy is $O(h^2)$ is verified next. For $k = 10^{-8}$, we see that each time N is doubled, i.e. h is halved, the error decreases by approximately $0.6 = 10^{0.6} \approx 4$, which agrees with the theory.

If we take each of the lines separately, which is equivalent to fixing h, we see that the error increases as k increases. This increase is roughly linear in the log-log plot, agreeing with the theory that the scheme is accurate to $O(k)$. For low enough N, e.g. $N = 401$, we see that if k is too large, then the stability condition $r \leq 0.5$ is violated, so the computer program outputs an infinite value for the error as the algorithm blows up, hence values are missing for the red line for k larger than 10^{-6}.

Figure A2 shows the relationship between k, N and the *error* at $x = 0.5$. Much of the analysis done in Figure A1 still applies here, but we see that the errors are significantly higher for smaller k. This time, we do see evidence that the scheme is $O(h^2)$ accurate because at $N = 51, h = 0.02, k = 10^{-8}$, the error is approximately $-4 = 10^{-4}$, which is in the same order of magnitude as $h^2 = 4 \cdot 10^{-4}$.

The reason why the overall errors are much larger in Figure A2 compared to Figure A1 is that $x = 0.02$ is very close to the boundary, whereas $x = 0.5$ is right in the middle of the interval, so the approximation will be naturally less accurate further away from the boundaries.

Figure A3: Errors against r, showing numerical stability around the value $r = 0.5$

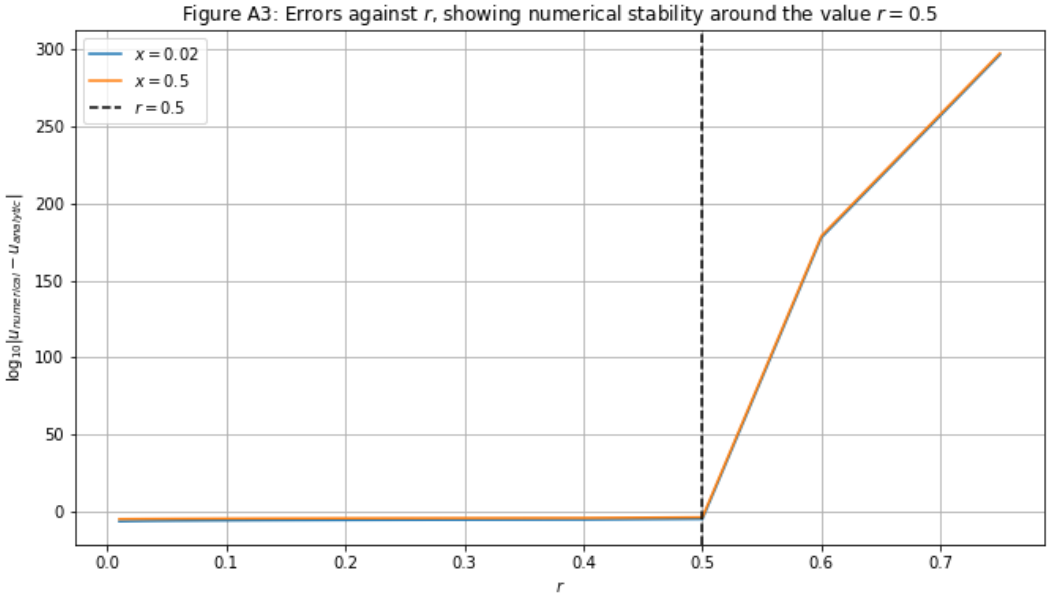
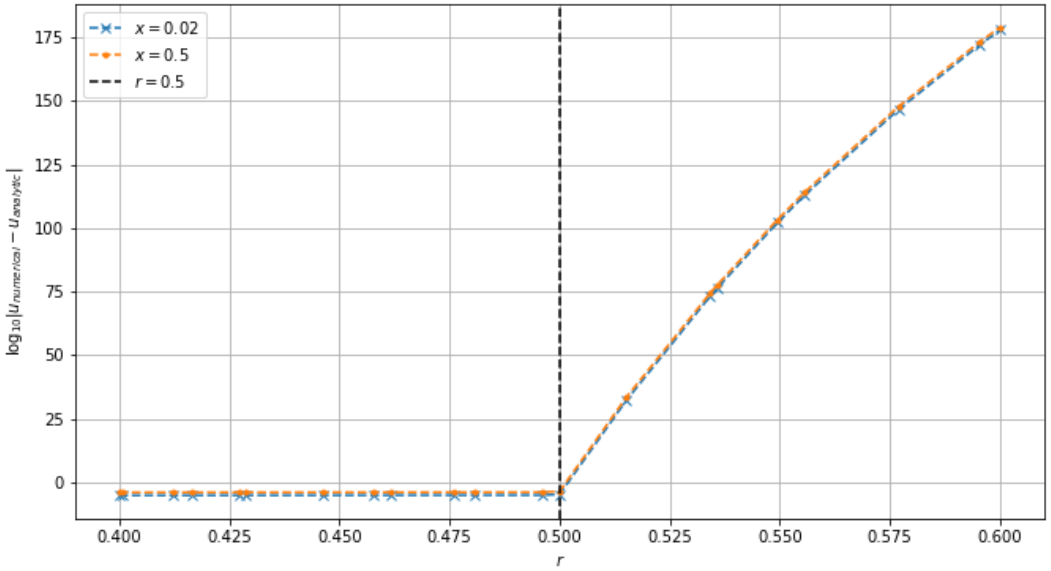


Figure A4: Errors against r, showing numerical stability finely around the value $r = 0.5$



Figures A3 and A4 clearly demonstrate that at values of $r > 0.5$, the explicit scheme becomes unstable, in agreement with the theory, and therefore the errors jump up to massive values e.g. at $r = 0.525$ the error is already higher than 10^{50}. This stability value is independent of where in the interval the solution error is calculated are, as seen by the fact that the jump in error occurs at the same point whether the solution is taken at $x = 0.02$ or $x = 0.5$.

2 Part B

2.1 Discretisation: Fully-Implicit and Crank-Nicolson

Until now, solving for the solution at time $j+1$ required knowledge of the solution at the previous time step j. With implicit methods, we make use of solutions at the current time-step $j+1$ as soon as they become available, provided we have a knowledge of the boundary conditions at our time-step. We can use formula (5.1) as seen in the lectures

$$U_n^{j+1} - U_n^j = r\left[\theta\delta^2 U_n^{j+1} + (1-\theta)\delta^2 U_n^j\right]$$

to approximate the equation $u_t = u_x x$. The operator δ^2 is the usual second-order difference operator.

This formula gives the groundwork needed to formulate implicit schemes, as long as $\theta \neq 0$. The fully-implicit scheme takes $\theta = 1$ which results in the equation $U_n^{j+1} - U_n^j = r\delta^2 U_n^{j+1}$, which after some manipulation, we arrive to the equation

$$(1+2r)U_n^{j+1} - r(U_{n+1}^{j+1} + U_{n-1}^{j+1}) = U_n^j. \tag{8}$$

Again, using the fact that at the boundaries $U_1, U_N = 0$, we can express equation 8 as the matrix equation $M_2 U_{2:N-1}^{j+1} = U_{2:N-1}^j$, where $U_{2:N-1}^k = (U_2^k, U_3^k, \ldots, U_{N-2}^k, U_{N-1}^k)^T$, where M_2 is a sparse tridiagonal Toeplitz matrix with entries: $1+2r$ on the diagonal and $-r$ on the diagonals above and below the main diagonal.

For the Crank-Nicolson scheme, we take $\theta = 0.5$, which results in the equation $U_n^{j+1} - U_n^j = r\delta^2\left[U_n^{j+1} + U_n^j\right]/2$, which after some manipulation, we arrive to the equation

$$(1+r)U_n^{j+1} - \frac{r}{2}(U_{n+1}^{j+1} + U_{n-1}^{j+1}) = (1-r)U_n^j + \frac{r}{2}(U_{n+1}^j + U_{n-1}^j) \tag{9}$$

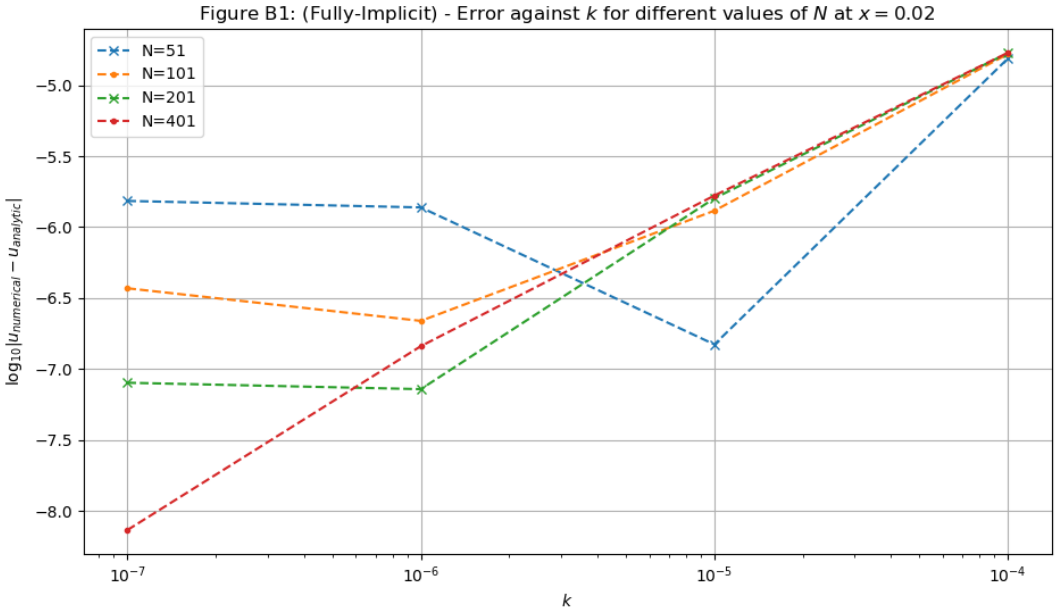
Similarly as before, we can express equation 9 as the system $M_3 U_{2:N-1}^{j+1} = M_4 U_{2:N-1}^j$, where M_3 is a sparse tridiagonal Toeplitz matrix with entries: $1+r$ on the diagonal and $-r/2$ on the diagonals above and below the main diagonal and M_4 is a sparse tridiagonal Toeplitz matrix with entries: $1-r$ on the diagonal and $r/2$ on the diagonals above and below the main diagonal.

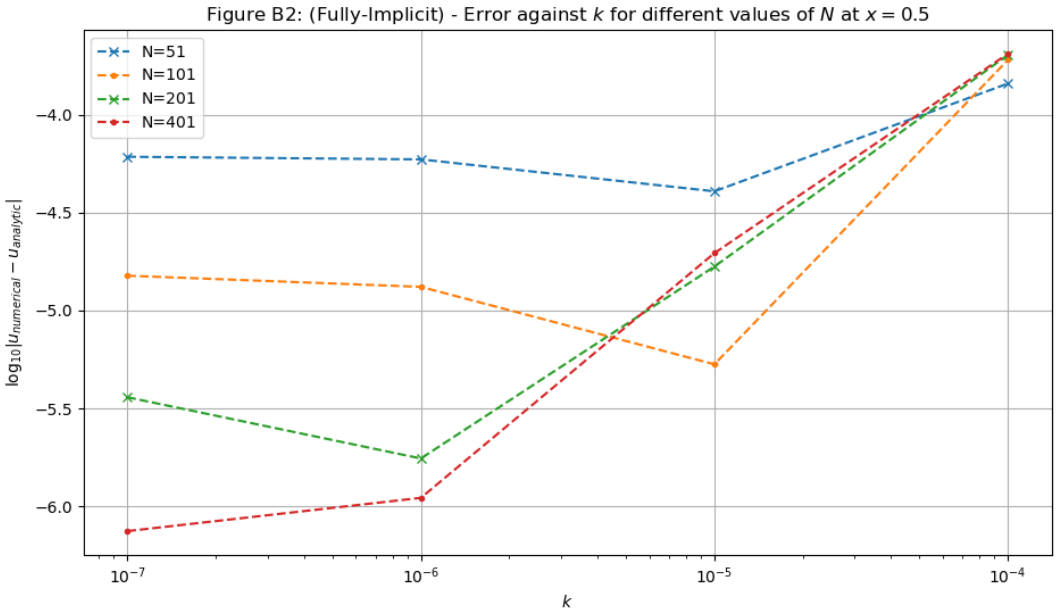
Both of these schemes involve solving a system of equations involving a tridiagonal matrix of size $N-2$. Using Gaussian elimination, this can be done with $O(N^3)$ complexity, however, we can exploit the matrix structures and use other algorithms (such as the Thomas Algorithm) for tridiagonal matrices which achieves this with a complexity of $O(N)$, i.e. linear.

In the implementation, we choose to use sparse matrices given in the Scipy module for memory optimisation, and we use the function `scipy.sparse.linalg.spsolve` which is optimised for solving sparse (tridiagonal) systems instead of using the `tridiag.m` function as it is of the same complexity and it is written in MATLAB not Python. The schemes are implemented in the file `schemes.py` in the functions `diffusion_fully_implicit` and `diffusion_crank_nicolson`.

2.2 Investigation of Implicit Schemes

We repeat our investigations of accuracy and numerical stability for the fully-implicit and Crank-Nicolson schemes. Turning to accuracy first, we obtain the following plots:





Figures B1 and B2 illustrate the accuracy of the fully-implicit ($\theta = 1$) scheme for varying k, N. The graph demonstrates that the scheme is $O(k), O(h^2)$ accurate and it's accuracy is similar to the fully-explicit scheme - the analysis for this is similar. One big difference between plots (A1, A2) and (B1, B2) is that we have small values for the error even if r is large. This demonstrates the fact that implicit schemes are unconditionally stable in terms

of r. We see that for this scheme, there is an optimal k (therefore r), at which the scheme has the lowest error. This is clearly seen on the blue line given by $N = 51$ in Figure B1.

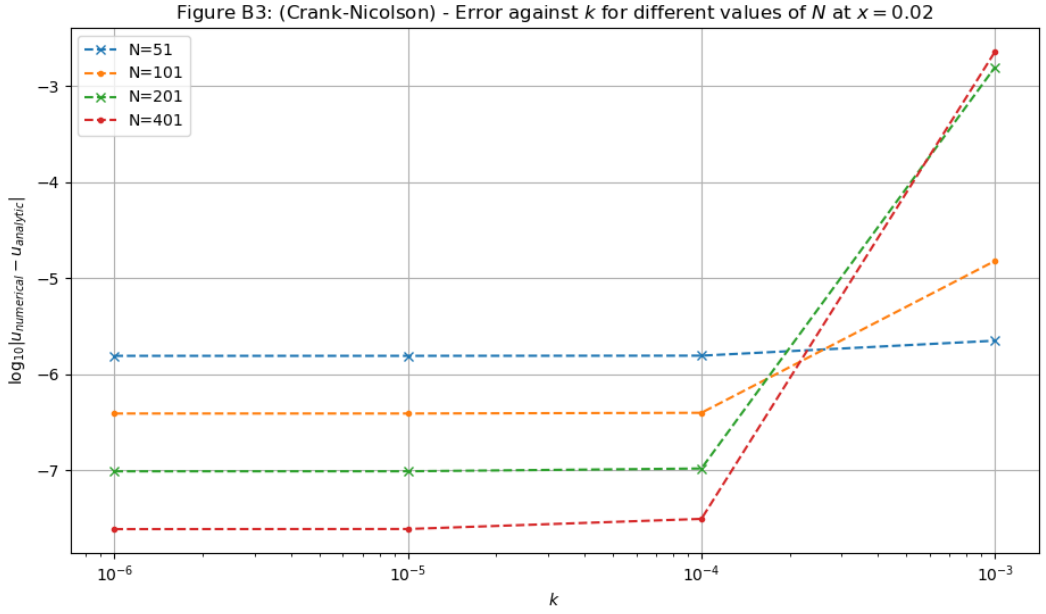


Figure B3: (Crank-Nicolson) - Error against k for different values of N at $x = 0.02$

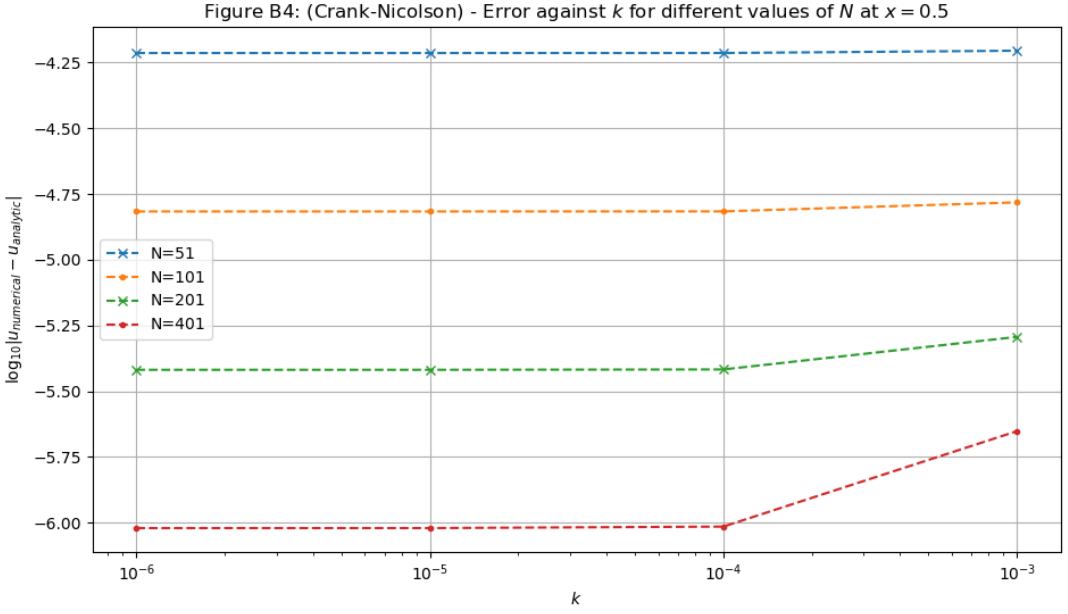


Figure B4: (Crank-Nicolson) - Error against k for different values of N at $x = 0.5$

Figures B3 and B4 demonstrate the numerical accuracy of the Crank-Nicolson in the same way as plots (A1, A2, B1, B2). This scheme seems to have the best numerical accuracy for larger k from all of the schemes considered so far, e.g. we can take $k = 10^{-4}$ and get error measures of ≤ 0.000001 meaning that we can use less time-steps. However, the Crank-Nicolson involves more operations per time-step, as it involves solving a system of (N-2) equation as well as multiplication of a (N-2) matrix with a vector.

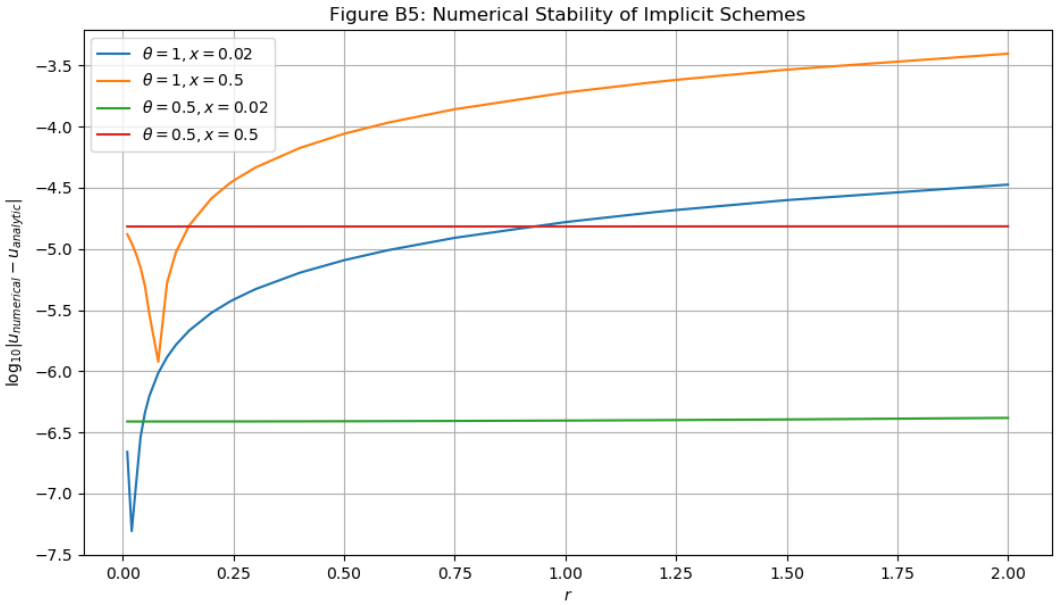
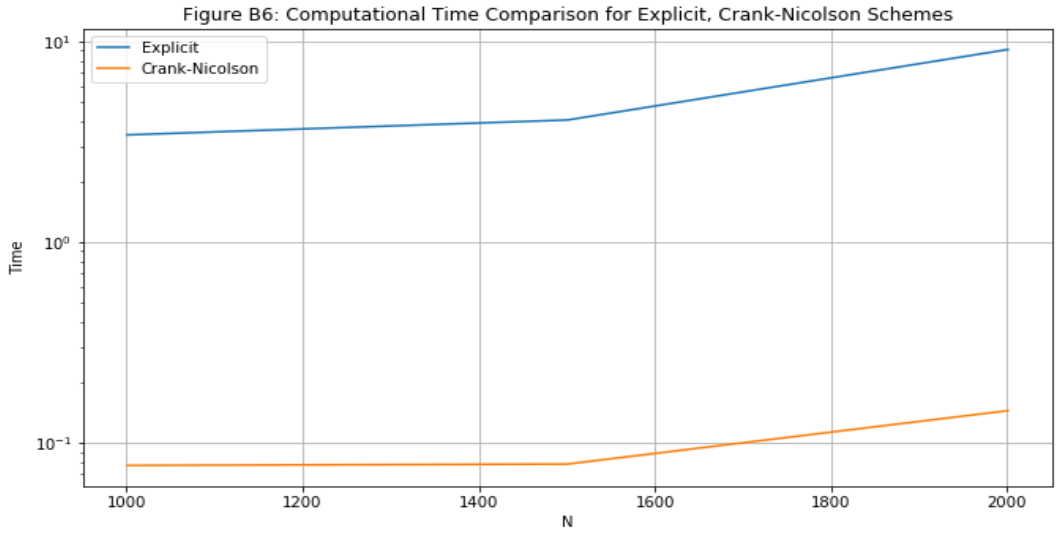
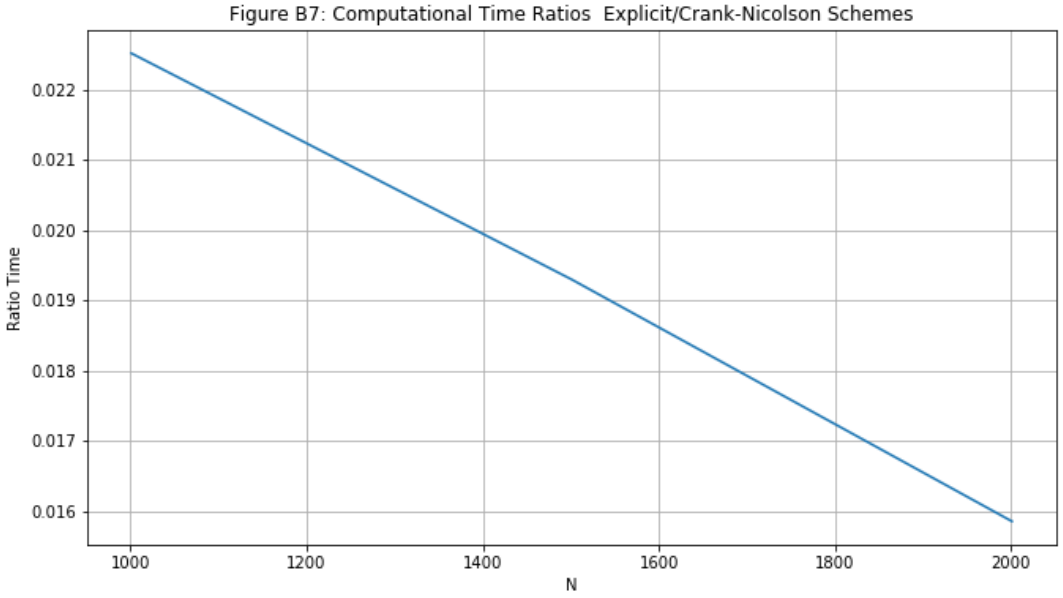


Figure B5 demonstrates the error for various values of r. We can see that the error is constant for all values of $r \leq 2$ for the Crank-Nicolson scheme, implying that it is unconditionally stable on r. For the fully-implicit scheme, we see that the there is a minimum in error for small r, corresponding to an optimal value of r to take, with the error then increasing as r increases but remaining bounded, i.e. does not explode as it is the case of the explicit scheme, implying that it is stable for $r > 0.5$.

2.3 Computation Times Comparison

In this question, we compare the computational times (in seconds) to arrive at the solution for the explicit and semi-explicit (Crank-Nicolson) schemes with a similar level of accuracy ($\approx 10^{-6} error$).





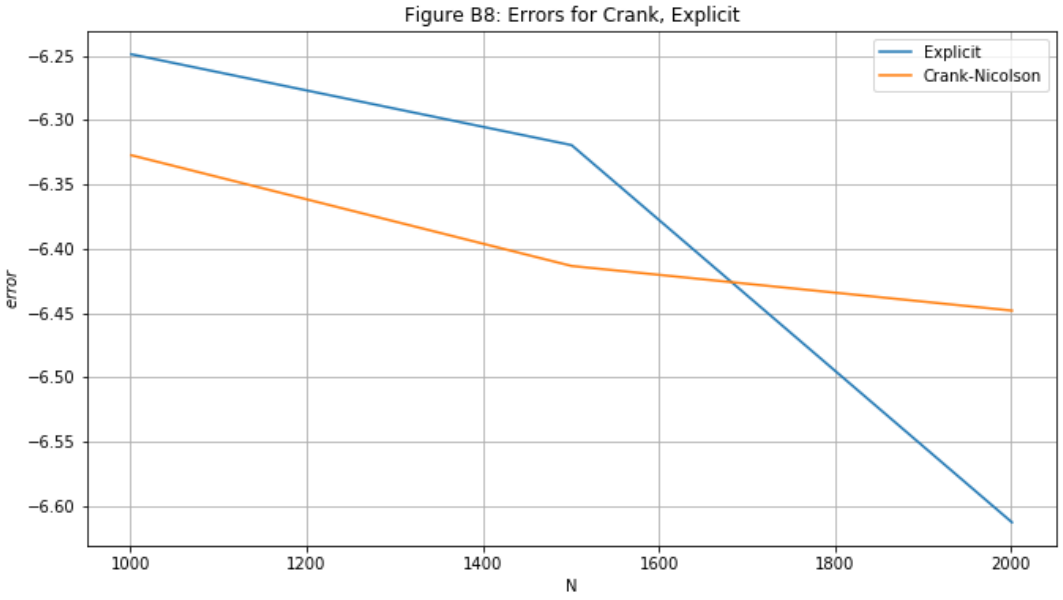


Figure B6 shows absolute times that each of the schemes takes to compute the solution, with their corresponding errors (shown in Figure B8) being approximately equal. It is clear, from Figure B7, that the explicit scheme is much slower, being approximately $0.022^{-1} \approx 45$ times slower for $N = 1000$ and $0.016^{-1} \approx 63$ times slower for $N = 2000$. This is to be expected because even though the Crank-Nicolson scheme involves more operations per iteration, the explicit scheme takes far more steps to arrive at the solution with the same accuracy resulting in a greater time overall.

2.4 Alternative 3-Point Time-Discretisation

To use this 3-point second-order accurate time-discretisation formula, we will need to substitute it into our implicit scheme as follows:

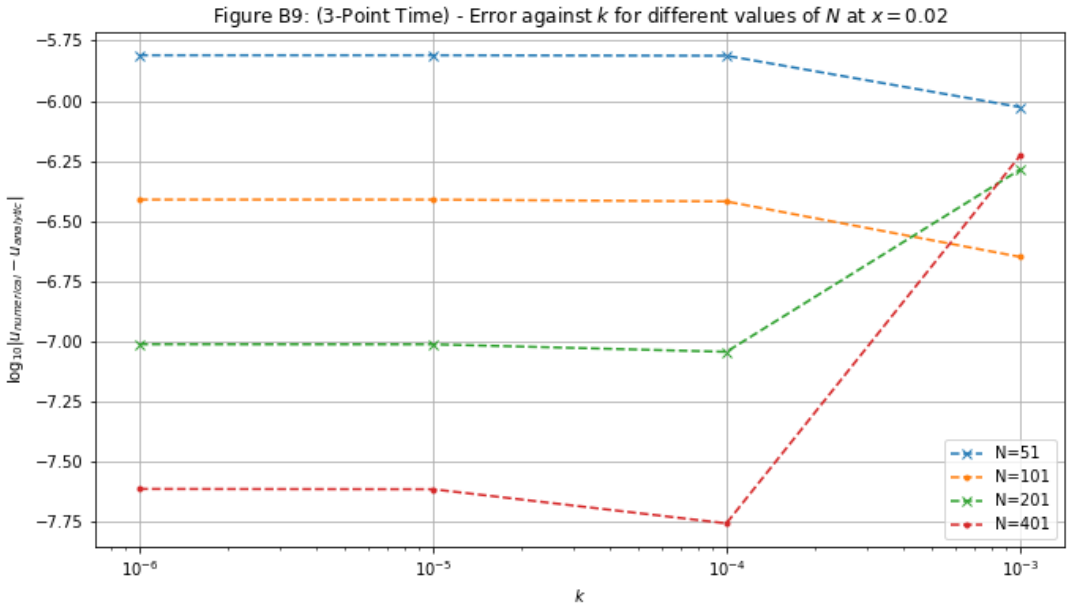
$$\begin{aligned}
&\frac{3U_n^{j+1} - 4U_n^j + U_n^{j-1}}{2k} = \frac{U_{n+1}^{j+1} - 2U_n^{j+1} + U_{n-1}^{j+1}}{h^2}, \\
\implies &3U_n^{j+1} - 4U_n^j + U_n^{j-1} = 2r(U_{n+1}^{j+1} - 2U_n^{j+1} + U_{n-1}^{j+1}), \\
\implies &(3+4r)U_n^{j+1} - 2r(U_{n+1}^{j+1} + U_{n-1}^{j+1}) = 4U_n^j - U_n^{j-1}.
\end{aligned}$$

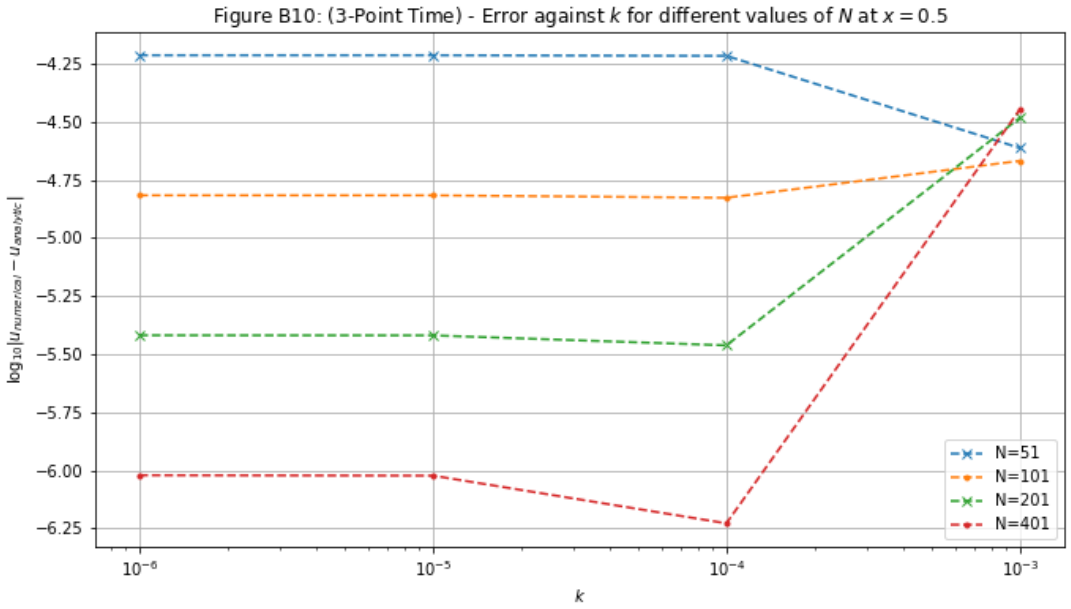
This involves solving a system of equations involving a tridiagonal matrix of size $N-2$. We can express it as the system $M_5 U_{2:N-1}^{j+1} = 4U_{2:N-1}^j - U_{2:N-1}^{j-1}$ where M_5 is a sparse tridiagonal Toeplitz matrix with entries: $(3+4r)$ on the main diagonal and $-2r$ on the diagonals above and below the main diagonal.

We have an issue at the very first time-step. Given some initial conditions, we know the solution U^0, but to calculate U^1, we would have to have knowledge of U^{-1}. This is not possible, so for the implementation, we have decided to use the fully-implicit scheme for the very first time-step and then proceed to use the 3-point scheme from times $t > 1$.

The implementation for this scheme can be found inside the file `schemes.py` under the function `diffusion_fully_implicit_alternative`.

Numerical Stability and Accuracy of Scheme:





We can see that this alternative 3-point second-order accurate time-discretisation has greatly improved the accuracy compared with the fully-implicit scheme. This is because the accuracy for this scheme is $O(k^2) + O(h^2)$, compared to $O(k) + O(h^2)$ for the fully-implicit scheme. This allows us to take larger k, e.g. $k = 10^{-4}$. This allows us to take a larger time-step and therefore arrive to the solution in a quicker time without sacrificing accuracy. The errors seen in the graph are mainly due to h. As we can see, the scheme is numerically unconditionally stable for r, as it is an implicit scheme.

3 Part C

3.1 Lagrangian Interpolation - FD Coefficients

An alternative to using Taylor series to compute the finite difference coefficients is to use Lagrangian interpolation. In lectures we saw that we can develop a 2nd-order accurate FD scheme using a quadratic interpolating polynomial $u(x_n) \approx f(x_n) = a + bx_n + cx_n^2$, for some coefficients a, b, c, which can be found by solving the corresponding matrix system.

We can extend this method to find a 4th-order accurate FD scheme using a quadratic polynomial $u(x_n) \approx f(x_n) = a + bx_n + cx_n^2 + dx_n^3 + ex_n^4$, for coefficients a, b, c, d, e. We can find 4th order accurate approximation to the second derivative by differentiating this expression twice, giving $f''(x_n) = 2c + 6dx_n + 12ex_n^2$.

The resulting matrix system to be solved is

$$\begin{pmatrix} 1 & x_0 - 2h & (x_0 - 2h)^2 & (x_0 - 2h)^3 & (x_0 - 2h)^4 \\ 1 & x_0 - h & (x_0 - h)^2 & (x_0 - h)^3 & (x_0 - h)^4 \\ 1 & x_0 & x_0^2 & x_0^3 & x_0^4 \\ 1 & x_0 + h & (x_0 + h)^2 & (x_0 + h)^3 & (x_0 + h)^4 \\ 1 & x_0 + 2h & (x_0 + 2h)^2 & (x_0 + 2h)^3 & (x_0 + 2h)^4 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \\ e \end{pmatrix} = \begin{pmatrix} f_{-2} \\ f_{-1} \\ f_0 \\ f_1 \\ f_2 \end{pmatrix}. \tag{10}$$

Once this system has been solved for the parameters $(a, b, c, d, e)^T$, we can find the central difference by the expression $f''(x_0) = 2c + 6dx_0 + 12ex_0^2$, and the side differences by the expressions $f''(x_0 - h) = 2c + 6d(x_0 - h) + 12e(x_0 - h)^2$ and $f''(x_0 + h) = 2c + 6d(x_0 + h) + 12e(x_0 + h)^2$.

Using the fast numerical implementation in the function `fd_coefficients` we obtain the following coefficients:

```
Central Differences:
[-0.0833  1.3333 -2.5     1.3333 -0.0833]
Backward Differences:
[ 0.9167 -1.6667  0.5     0.3333 -0.0833]
Forward Differences:
[-0.0833  0.3333  0.5    -1.6667  0.9167]
```

Therefore, transforming these into fractions we get the coefficients

$$\begin{aligned} &a = -\frac{1}{12h^2}, \quad b = \frac{4}{3h^2}, \quad c = -\frac{5}{2h^2}, \\ &d = \frac{11}{12h^2}, \quad e = -\frac{5}{3h^2}, \quad f = \frac{1}{2h^2}, \quad g = \frac{1}{3h^2}, \quad j = -\frac{1}{12h^2}, \\ &l = -\frac{1}{12h^2}, \quad m = \frac{1}{3h^2}, \quad n = \frac{1}{2h^2}, \quad p = -\frac{5}{3h^2}, \quad q = \frac{11}{12h^2}. \end{aligned}$$

as required. These will be used in the code to develop higher order FD schemes.

3.2 Higher Order Explicit Scheme Formulation

Taking an N-point discretisation of the function u, $U^j = (U_1^j, U_2^j, \ldots, U_{N-1}^j, U_N^j)^T$ at time j. The 4th-order central difference explicit discretisation is given by the formula

$$U_n^{j+1} - U_n^j = r\left[-\frac{1}{12}(U_{n-2}^j + U_{n+2}^j) + \frac{4}{3}(U_{n-1}^j + U_{n+1}^j) - \frac{5}{2}U_n^j\right],$$

which upon rearranging it becomes, for $3 \leq n \leq (N-2)$,

$$U_n^{j+1} = -\frac{r}{12}(U_{n-2}^j + U_{n+2}^j) + \frac{4r}{3}(U_{n-1}^j + U_{n+1}^j) + \left(1 - \frac{5r}{2}\right)U_n^j. \tag{11}$$

The 4th-order forward difference discretisation is given by the formula

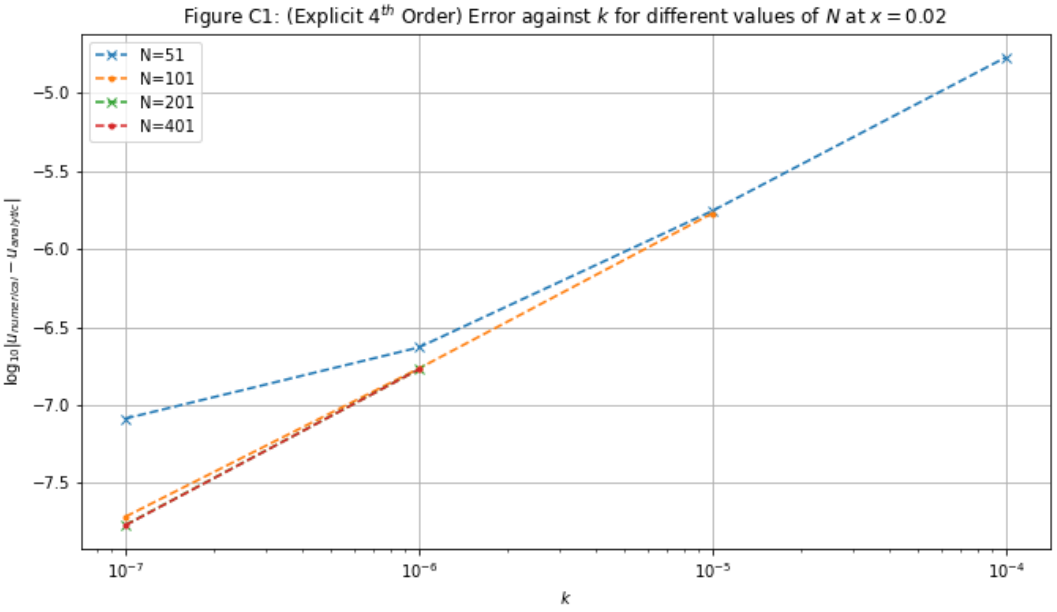
$$U_2^{j+1} - U_2^j = r\left[\frac{11}{12}U_1^j - \frac{5}{3}U_2^j + \frac{1}{2}U_3^j + \frac{1}{3}U_4^j - \frac{1}{12}U_5^j\right]. \tag{12}$$

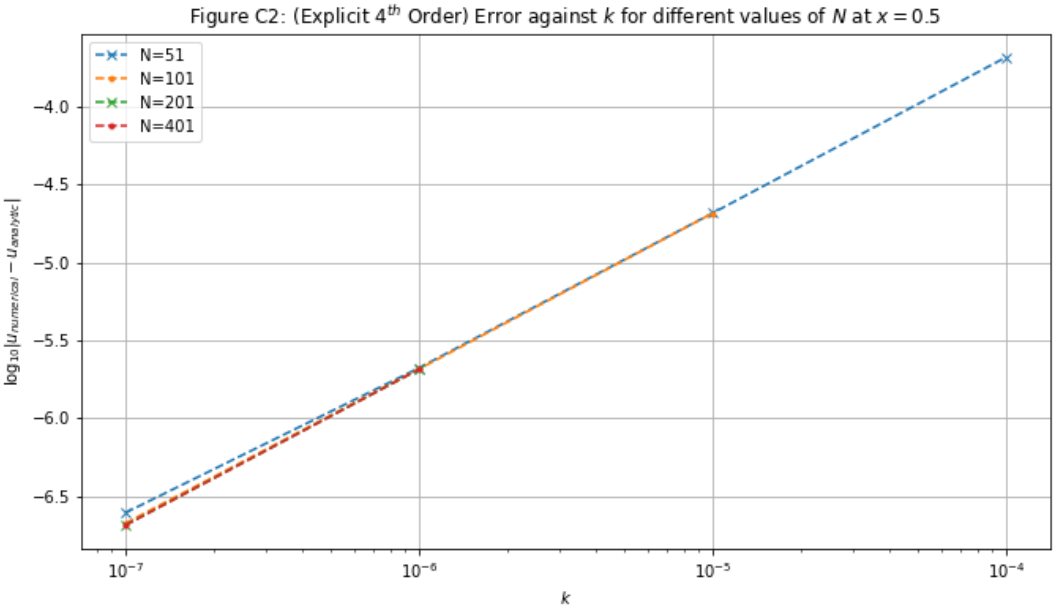
The 4th-order backward difference discretisation is given by the formula

$$U_{N-1}^{j+1} - U_{N-1}^j = r\left[-\frac{1}{12}U_{N-4}^j + \frac{1}{3}U_{N-3}^j + \frac{1}{2}U_{N-2}^j - \frac{5}{3}U_{N-1}^j + \frac{11}{12}U_N^j\right]. \tag{13}$$

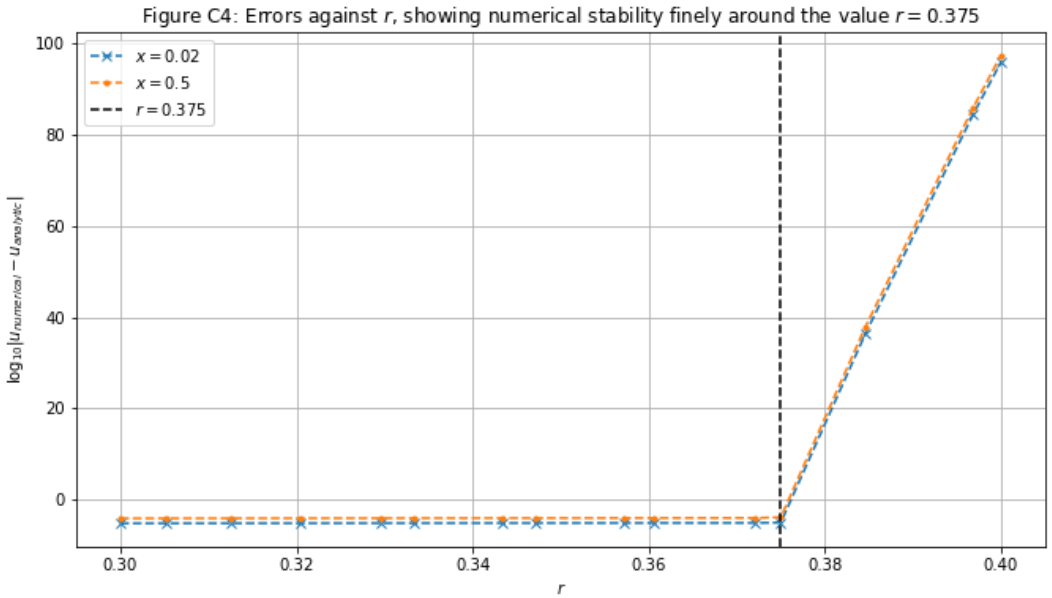
Numerical Stability and Accuracy of Scheme:

We expect this scheme to be more accurate than the second-order scheme for low enough k. However, this comes at the expense of stability, as the region of stability will be smaller, i.e. $r_{2nd\ order} > r_{4th\ order}$. The other drawback of this method is that at each iteration we have to multiply by a pentadiagonal matrix.





We see that the error in figures C1 and C2 is now mostly due to k. This is especially the case when computing the error at $x = 0.5$ since all the errors for all the lines increase linearly with k. This is to be expected, as the method is still $O(k)$ accurate but it is $O(h^4)$ accurate. This allows us to use a larger discretisation step h, reducing the amount of operations that need to be computed at each iteration. We only see that the blue line ($N = 51$) is flattening in Figure C1, and this corresponds to the limit of accuracy $[O(k) + O(h^4)]$ the scheme, since at that point, $h = 0.02 \implies h^4 = 1.6 * 10^{-7}$, and $k = 10^{-7}$, hence we expect the total error in the solution to be in the order of 10^{-7} magnitude, which is seen in the figures.



We can empirically that cut-off point of the stability region of this scheme is $r \approx 0.375$, i.e. we expect the scheme to be stable for values of $r < 0.375$.

3.3 Comparison of Explicit Schemes 2nd vs 4th Order

We begin by plotting the numerical solutions for u at $t = 0.005$ for $N = 21, k = 10^{-6}$ as well as the analytical solution superimposed onto one graph.

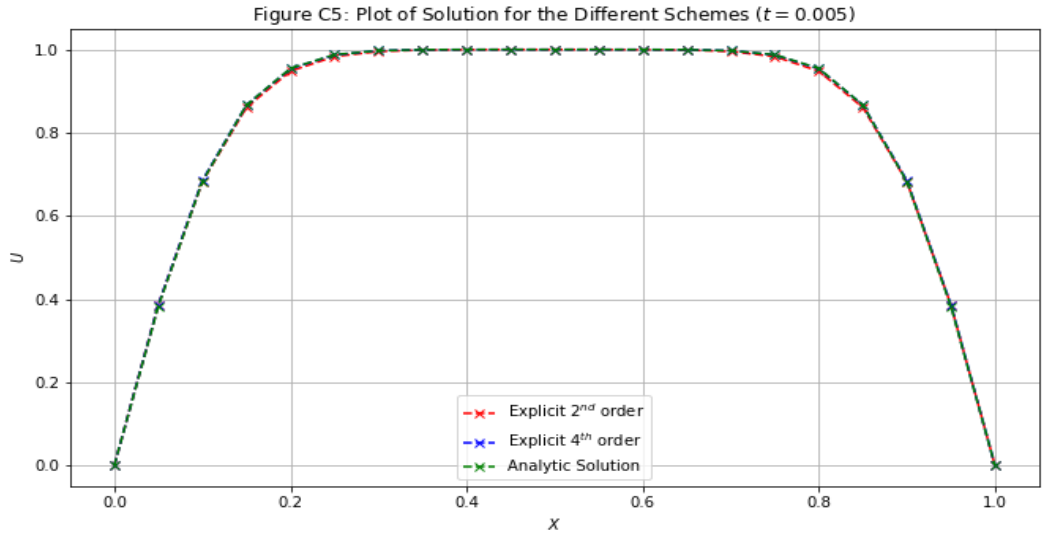


Figure C5 shows that the errors cannot be seen with the eye and must be computed and plotted on a log scale. We now move on to plot the plot of errors for the same parameters.

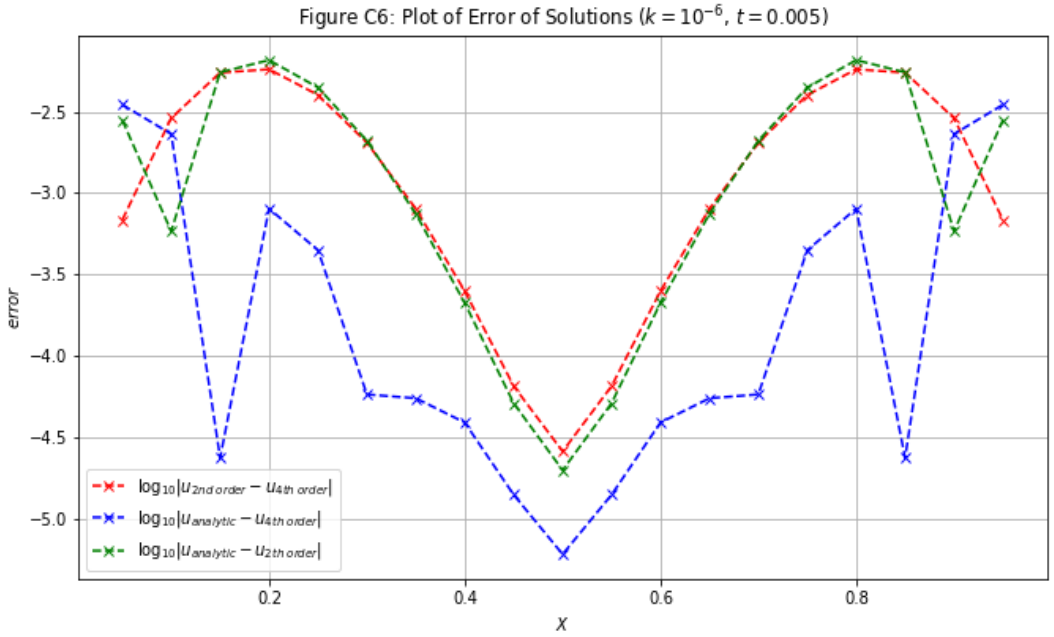


Figure C6 shows how the errors behave across the interval (errors at points U_1, U_N are omitted). We see that overall, the errors for the fourth-order are much lower than the errors for the second-order, however they oscillate a lot more, which is due to picking a small step-size h. Interestingly enough, if we take the error between the second-order and fourth-order differences, the line becomes less oscillatory and smoother, implying that some of the oscillatory behaviour may get cancelled out.

Another observation is that the errors in the center appear to be much smaller than the errors at the edges. This is to be expected, as the points near the middle remain mostly unchanged for small t. This can be seen clearly in Figure C5, as the function is mostly flat near for $0.3 \leq x \leq 0.7$ at $t = 0.005$. Physically, this corresponds to the heat in the middle of the function having not dissipated or diffused yet. This causes the errors at the middle to be smaller than at the edges, at least for a short period of time during the solution span.

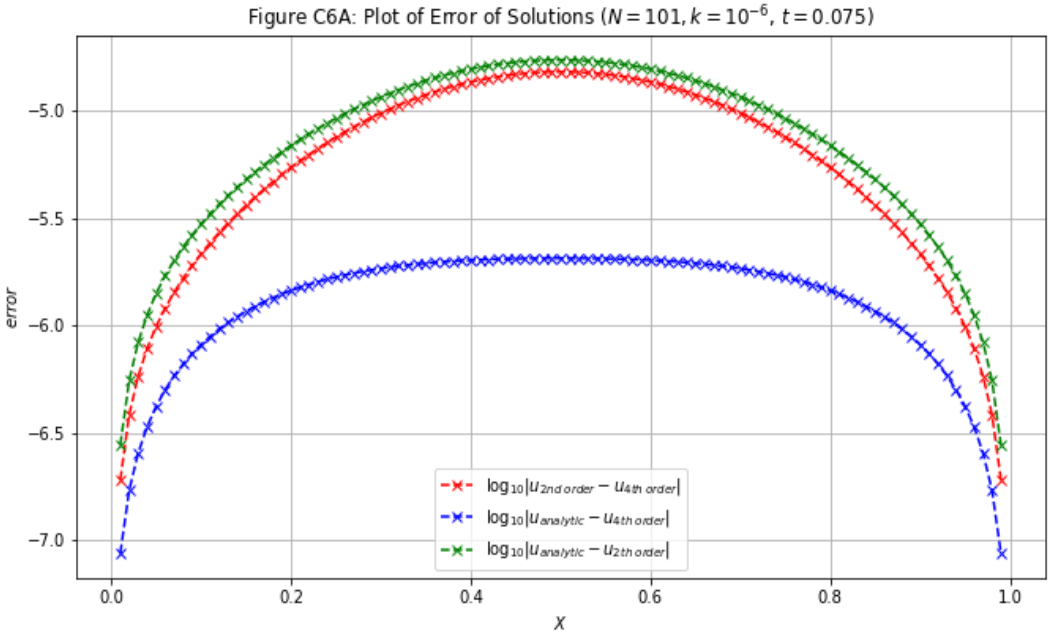


Figure C6A illustrates the errors for a much larger time i.e. $t = 0.075$. At this time, our errors at the boundaries are much smaller than the errors furthest away from the boundaries, which is expected. Physically this means that by $t = 0.075$, enough time has passed for the diffusion process to have reached the region around $x = 0.5$.

3.4 Fully-Implicit Scheme Fourth Order

The 4th-order central difference fully-implicit discretisation is given by the formula

$$U_n^{j+1} - U_n^j = r\left[-\frac{1}{12}(U_{n-2}^{j+1} + U_{n+2}^{j+1}) + \frac{4}{3}(U_{n-1}^{j+1} + U_{n+1}^{j+1}) - \frac{5}{2}U_n^{j+1}\right],$$

$$\frac{r}{12}(U_{n-2}^{j+1} + U_{n+2}^{j+1}) - \frac{4r}{3}(U_{n-1}^{j+1} + U_{n+1}^{j+1}) + \left(1 + \frac{5r}{2}\right)U_n^{j+1} = U_n^j. \tag{14}$$

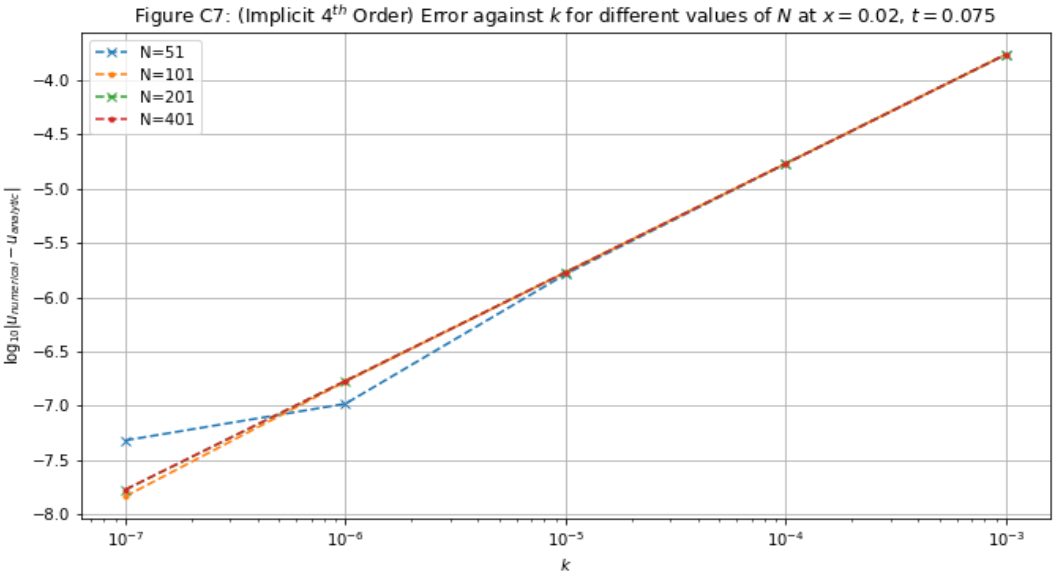
The 4th-order forward difference fully-implicit discretisation is given by the formula

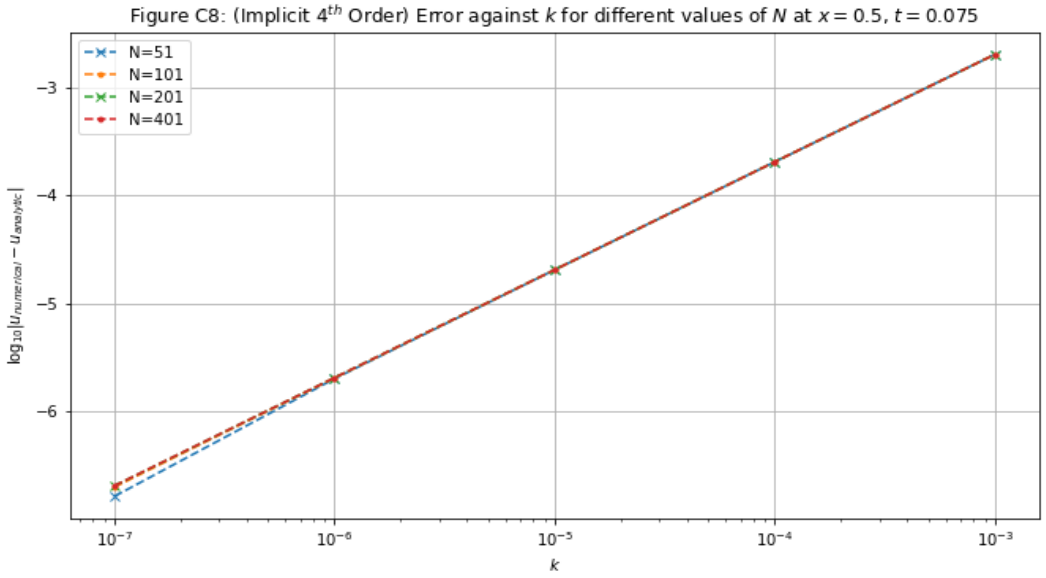
$$U_2^{j+1} - U_2^j = r\left[\frac{11}{12}U_1^{j+1} - \frac{5}{3}U_2^{j+1} + \frac{1}{2}U_3^{j+1} + \frac{1}{3}U_4^{j+1} - \frac{1}{12}U_5^{j+1}\right]. \tag{15}$$

The 4th-order backward difference fully-implicit discretisation is given by the formula

$$U_{N-1}^{j+1} - U_{N-1}^j = r\left[-\frac{1}{12}U_{N-4}^{j+1} + \frac{1}{3}U_{N-3}^{j+1} + \frac{1}{2}U_{N-2}^{j+1} - \frac{5}{3}U_{N-1}^{j+1} + \frac{11}{12}U_N^{j+1}\right]. \tag{16}$$

Numerical Stability and Accuracy of Scheme:





Figures C7, C8 show that the errors are now all due to the accuracy in terms of k. Overall, the errors have decreased when compared to the 2nd-order fully-implicit scheme. This is to be expected, as we are using more data at each time-step and it is mathematically of a higher order of accuracy.

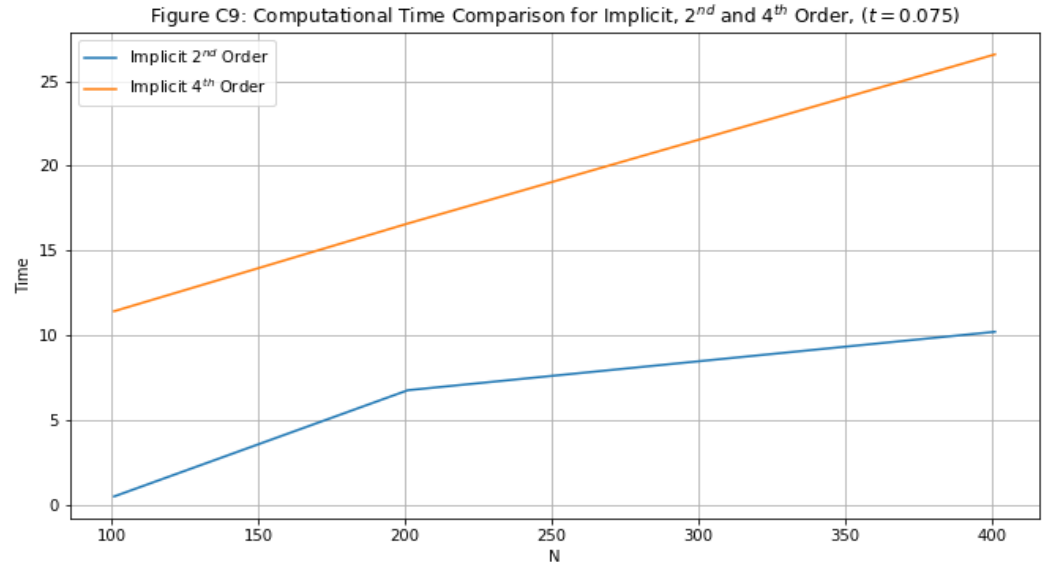


Figure C9 illustrates the times (in s) taken to achieve a similar error for the second-order implicit scheme compared to the fourth-order implicit scheme. It is clear from the graph that the fourth-order implicit scheme is much slower, therefore less numerically efficient. This is due to the fact that we need to solve a pentadiagonal matrix system. This is much more costly than solving a tridiagonal Toeplitz matrix, and this explains the differences in times.