Numerical Solution of the Heat Conductance Equation

1 Introduction

1.1 Motivation

Numerical solutions are approximate answers obtained through computational methods for problems that are difficult or impossible to solve analytically.

This report analyses different numerical methods for solving the heat conductance equation, given by the parabolic PDE in equation 1.

$$\frac{\partial^2 T(x,t)}{\partial x^2} = \frac{1}{k}\frac{\partial T(x,t)}{\partial t} \tag{1}$$

1.2 Problem statement

This equation describes evolution in time of the temperature distribution across a one-dimensional homogeneous bar of length $L = 100$ cm and thermal conductivity $k = 0.875\ \mathrm{cm^2 s^{-1}}$. The boundary conditions are $T(x,0) = 500\ °\mathrm{C}$ and $T(0,t) = T(0,L) = 0\ °\mathrm{C}$.

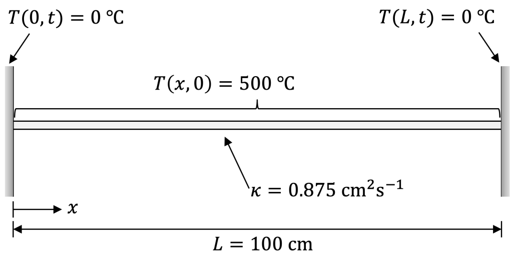

Figure 1: Problem description

Since we will be using finite difference approximations , it is useful to consider the problem as a two-dimensional grid representing space-time.

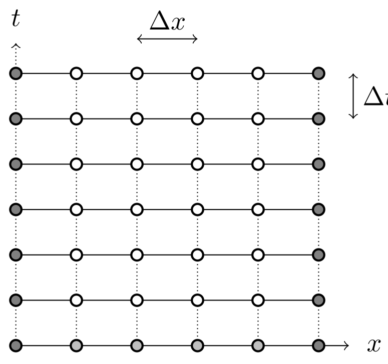

Figure 2: grid with $\Delta x = 20$ cm and $\Delta t = 100$ s

We explore two types of numerical solution to this problem: **explicit** and **implicit** solutions. To evaluate our numerical solutions, the analytical solution is given for Parts I and II is given by equation 2.

$$T(x,t) = \frac{2000}{\pi}\sum_{\substack{n=1\\ n\text{ odd}}}^{\infty}\frac{1}{n}\sin\frac{n\pi x}{L}e^{-\frac{n^2\pi^2 k}{L^2}t} \tag{2}$$

1.3 Software setup

For the software implementation **Python3** will be used since it has nice linear algebra and calculus tooling (e.g. **NumPy**, **SciPy**), and because these tools are very well integrated with **Matplotlib** which will be use to plot our results.

The code is structured between the `HeatSolver` module and class which solves the heat equation in various ways, and a Notebook to run and visualize the experiments. The snippet in figure 3 gives some boilerplate for defining the class and implementing the benchmark analytic solution.

```python
@attrs.define
class HeatSolver:
  dx: np.float32
  dt: np.float32
  L: np.float32 = np.float32(100.0)
  k: np.float32 = np.float32(0.875)
  t_total: np.float32 = np.float32(600.0)
  T_x0: Callable = lambda x: 500.0 * np.ones_like(x)
  T_0t: np.float32 = np.float32(0.0)
  T_Lt: np.float32 = np.float32(0.0)

  ...

  def _solve_analytic_x_t(self, x , t , N = 100_001):
    x = np.atleast_1d(x)
    t = np.atleast_1d(t)

    ns = np.arange(1, N + 1, 2)
    sums = (1 / ns) * sin(ns[,:] * pi * x[:,] / self.L) @ \
    exp(- ns[:,]**2 * pi**2 * self.k * t[,:] / self.L**2)

    return 2000.0 / np.pi * sums.T
```

Figure 3: Definition & Analytic solution code

The function `_solve_analytic_x_t` takes advantage of Numpy's **broadcasting** features to compute solutions given vectors $\boldsymbol{x_i}$ and $\boldsymbol{t_j}$ all in one operation. This optimisation allows using a high precision factor $\boldsymbol{N}$ while keeping run-time low. It follows that for any Δx and Δt we can deduce the vectors defining the discrete grid and sample all the analytical solutions needed.

Within this class we can now define numerical solutions for a general version of the heat equation, and easily compare those solutions against an analytic implementation.

2 Part I: Explicit Solution: Forward Time Centered Space

2.1 Deriving the update equations

To derive the FTCS update equations we will use the central difference approximation for the spatial derivative (3), and the forward difference approximation for the time derivative (4).

$$\frac{\partial^2 T_{i,j}}{\partial x^2} = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} \tag{3}$$

$$\frac{\partial T_{i,j}}{\partial t} = \frac{T_{i,j+1} - T_{i,j}}{\Delta t} \tag{4}$$

Substituting these into (1) and rearranging[A] gives the update equation for node $T_{i,j+1}$

$$T_{i,j+1} = \lambda T_{i+1,j} + (1 - 2\lambda)T_{i,j} + \lambda T_{i-1,j} \tag{5}$$

Where

$$\lambda = \frac{k\Delta t}{\Delta x^2}$$

This equation gives the following **computational molecule**.

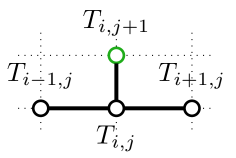

Figure 4: Explicit computational molecule

Given or boundary conditions and the parabolic nature of the heat equation, we can simply slide the computational molecule left to right bottom to top to compute our approximation.

2.2 Software Implementation

```
def solve_finite_distance(self) -> np.ndarray:
  A = self.k * self.dt / self.dx**2
  B = 1 - 2 * A

  N_x = int(self.L / self.dx + 1)
  N_t = int(self.t_total / self.dt + 1)

  T = np.zeros((N_t, N_x))
  T[0, :] = self.T_x0(np.arange(0, self.L + self.dx, self.dx))
  T[:, 0] = self.T_0t
  T[:, -1] = self.T_Lt

  for j in range(0, N_t - 1):
    for i in range(1, N_x - 1):
        T[j+1, i] = A * (T[j, i+1] + T[j, i-1]) + B * T[j, i]

  return T
```

Figure 5: Explicit solution

The algorithm has the consists of the following steps:

1. Pre-compute the coefficients $\mathtt{A} = \lambda$ and $\mathtt{B} = 1 - 2\lambda$
2. Derive the shape of the solution grid from Δx, Δt
3. Initialize an empty matrix and apply boundaries
   (a) Set the row $T_{i,0}$ to $T(\boldsymbol{x_i}, 0)$
   (b) Set the edge $T_{0,j}$ to $T(0, j)$
   (c) Set the edge $T_{L,j}$ to $T(L, j)$
4. Apply the computational molecule row by row, from the bottom up.

It is important to note that the boundary conditions must be set in that exact order, to maintain the information at the horizontal edges of the rod through time. You might also ask, why does the $t = 0$ s value take a vector $\boldsymbol{x_i}$? This is to prepare for when the function $T(x, 0)$ is no longer uniform in Part III. For now, the bottom row is a uniform vector of value $500°\mathrm{C}$.

2.3 Experiments and analysis

Three experiments where conducted:

1. Plotting the temperature distribution over the length of the bar at 100s interval with step sizes $\Delta x = 20\mathrm{cm}, \Delta \mathrm{t} = 100\mathrm{s}$
2. Plotting the temperature distribution and error against the analytical solution overtime with step sizes $\boldsymbol{\Delta x} = [20, 20, 10]\mathrm{cm}$, $\boldsymbol{\Delta t} = [100, 50, 100]\mathrm{s}$ at $x = 20\mathrm{cm}$, and recording the values at the final time-step
3. Plotting the temperature distribution in space overtime for $\Delta x = 10\mathrm{cm}, \Delta \mathrm{t} = 30\mathrm{s}$

It can be seen in figure 6 that the explicit numerical solution with that step size gives more accurate results towards the center of the bar, and they get more accurate as time progresses.

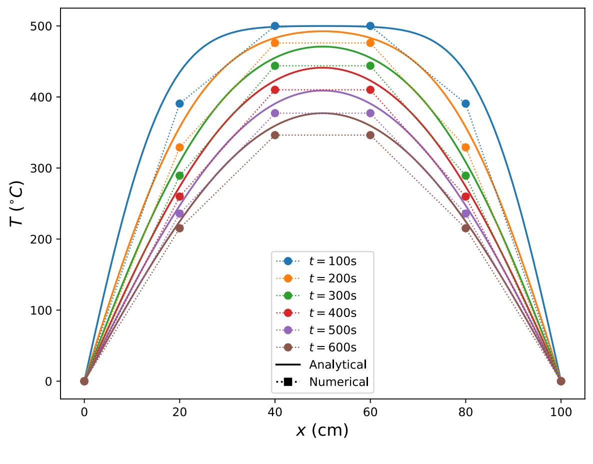

Figure 6: T vs. x, $\Delta \mathrm{x} = 20\mathrm{cm}, \Delta \mathrm{t} = 100\mathrm{s}$

The same trend can also be confirmed in figure 7 for the first two step size pairs, seeing that the error trends to 0 *asymptotically*. This indicates that for these step size values this method is stable.

Moreover, in the same figure it is apparent that the approximation fails to converge on the solution when the step size is set to $\Delta x = 10\mathrm{cm}$ and $\Delta t = 100\mathrm{s}$, it is *unstable*.

This can be verified theoretically as well. The stability condition for equation (5) as a solution to the heat equation is given by the following:

$$(1-2\lambda) \geq 0$$
$$\implies \frac{k\Delta t}{\Delta x^2} \leq \frac{1}{2} \quad (6)$$

Substituting Δx and Δt in equation (6) gives

$$\frac{k\Delta t}{\Delta x^2} = \frac{0.875 \cdot 100}{10^2} > \frac{1}{2} \quad (7)$$

which clearly breaks the stability condition. Hence, it can be concluded that the implementation behaves as expected, as this oscillation and increasing magnitude of the error is a feature of the method, not a bug in the implementation *(e.g. precision issue)*.

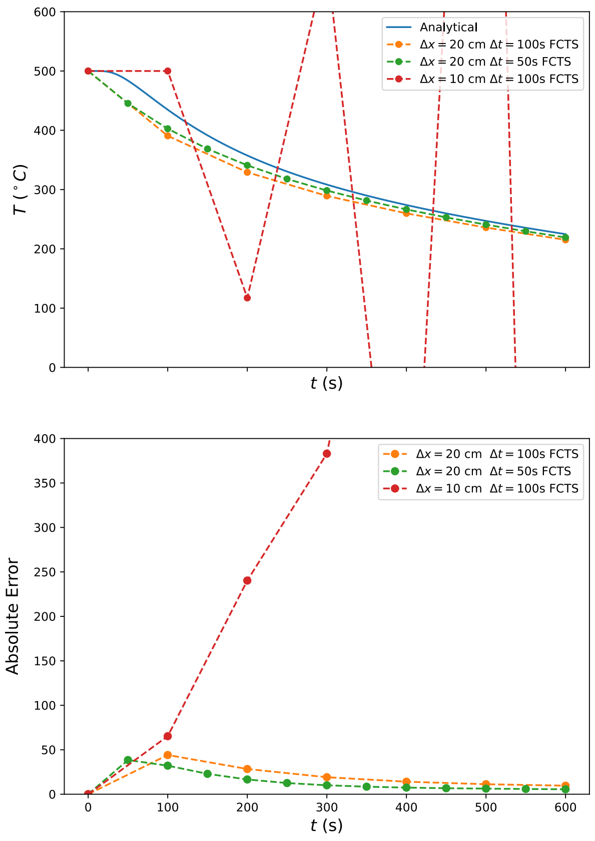

Figure 7: T vs. t at x $= 20$cm explicit

A observation on figure 7 is the fact that, as expected, out of the two stable configurations tested, the one with a finer grid is also the one with the better approximation. This observation enforces the idea that as long as the stability threshold is not crossed lowering either the step-size yields better accuracy.

$\Delta x/\Delta t$	20 / 100	20 / 50	10 / 100	analytic
$T(°C)$	215	219	-3161	225
abs. err.	10	6	3386	

Table 1: $T(°C)$ and error at $t = 600$s

Finally, figures 8 and 9 are 3-dimensional visualisations of the Temperature distribution and Absolute Error of the approximations made for step sizes $\Delta x = 10$cm and $\Delta t = 30$s, chosen to be granular, but without becoming unstable.

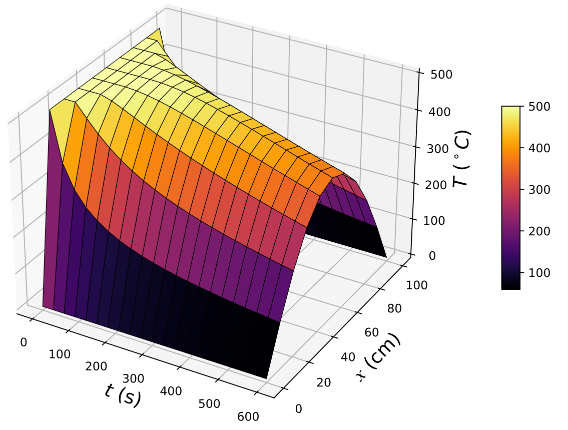

Figure 8: $T(x,t)$ with $\Delta x = 10\text{cm}, \Delta\text{t} = 30\text{s}$

In figure 9 it can be seen that the assumption made above regarding the error converging towards 0 overtime holds. The nodes with the largest Absolute Error ($\sim 40°C$) occur at one lengthwise extremities at the first time-step ($T_{1,1}, T_{L-1,1}$), this corresponds to the region in the temperature distribution in figure 8 with the steepest slope; as that slope smooths out overtime, so does the error.

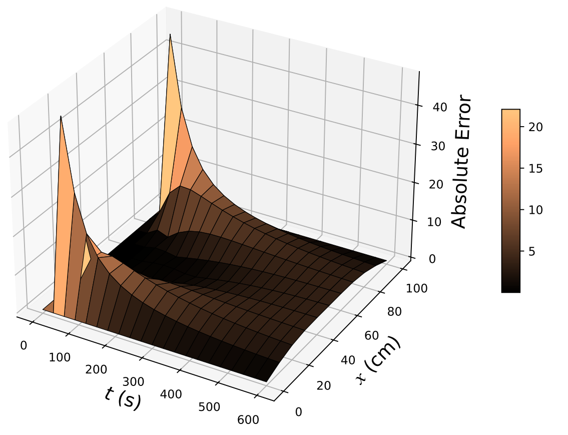

Figure 9: AE with $\Delta x = 10\text{cm}, \Delta\text{t} = 30\text{s}$

Therfore, the conclusion can be drawn that an accurate implementation of the explicit numeric solution has been achieved. It computes better results overtime and towards the center of the bar, but it is unstable for step sizes that break the inequality in 6.

3 Part II: Implicit Solution: Crank-Nicolson Method

3.1 Deriving the update equations

To derive the update equations with the Crank-Nicolson Method, the derivatives at the time midpoint between two time steps are considered, given by the time derivative (8) and the space derivative (9)

$$\left.\frac{\partial T}{\partial t}\right|_{i,j+\frac{1}{2}} = \frac{1}{\Delta t}(T_{i,j+1} - T_{i,j}) \tag{8}$$

$$\left.\frac{\partial^2 T}{\partial x^2}\right|_{i,j+\frac{1}{2}} = \frac{1}{2\Delta x^2}(T_{i-1,j} - 2T_{i,j} + T_{i+1,j} + T_{i-1,j+1} - 2T_{i,j+1} + T_{i+1,j+1}) : \tag{9}$$

Substituting these into (1) and rearranging$^{[B]}$ gives the update scheme for the Crank-Nicolson method, in which the left hand side contains unknown values and the right hand side contains known values.

$$\lambda T_{i-1,j+1} + \overbrace{(-2 - 2\lambda)}^{\alpha} T_{i,j+1} + \lambda T_{i+1,j+1} = -\lambda T_{i-1,j} + \overbrace{(2\lambda - 2)}^{\beta} T_{i,j} - \lambda T_{i+1,j} \tag{10}$$

This equation gives the computational molecule in figure 10.

We can now construct a system of equations from all the interior nodes at a particular time step to compute all the interior nodes at the next time step. To that end, the following two matrices and bias vector are constructed to encode this system of equations:

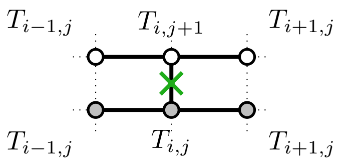

Figure 10: Implicit computational molecule

$$\overbrace{\begin{bmatrix} \alpha & \lambda & & & \\ \lambda & \alpha & \lambda & & \\ & \lambda & \ddots & \ddots & \\ & & \ddots & \ddots & \lambda \\ & & & \lambda & \alpha \end{bmatrix}}^{A} \times \begin{bmatrix} T_{1,j+1} \\ T_{2,j+1} \\ T_{3,j+1} \\ \vdots \\ T_{L-1,j+1} \end{bmatrix} = \overbrace{\begin{bmatrix} \beta & -\lambda & & & \\ -\lambda & \beta & -\lambda & & \\ & -\lambda & \ddots & \ddots & \\ & & \ddots & \ddots & -\lambda \\ & & & -\lambda & \beta \end{bmatrix}}^{B} \times \begin{bmatrix} T_{1,j} \\ T_{2,j} \\ T_{3,j} \\ \vdots \\ T_{L-1,j} \end{bmatrix} + \overbrace{\begin{bmatrix} -\lambda T_{0,j+1} - \lambda T_{0,j} \\ \\ \\ \\ -\lambda T_{L,j+1} - \lambda T_{L,j} \end{bmatrix}}^{\boldsymbol{b}}$$

Which can be written in the form

$$A\boldsymbol{T_{j+1}} = B\boldsymbol{T_j} + \boldsymbol{b}\,. \tag{11}$$

Equation (11) gives a row-by-row update scheme for the Crank-Nicolson model. To solve this equation for the vector $\boldsymbol{T_{j+1}}$ we can use the Thomas Algorithm[4], since the matrix A is a tridiagonal matrix.

3.2 Software Implementation

```
def solve_crank_nicolson(self) -> np.ndarray:
  lmbd = self.k * self.dt / self.dx**2
  alph, beta = - 2 * lmbd - 2, - 2 * lmbd + 2
  N_x, N_t = int(self.L/self.dx)-1, int(self.t_total/self.dt)+1

  Ab = np.zeros((3, N_x))
  Ab[0, 1:] = [lmbd] * (N_x - 1)
  Ab[1, :] = [alph] * N_x
  Ab[2, :-1] = [lmbd] * (N_x-1)
  B = diag([beta] * N_x) + diag([-lmbd] * (N_x - 1), k=1) + \
    diag([-lmbd] * (N_x - 1), k=-1)
  b = np.zeros(N_x)
  b[0] = - 2 * lambda *  self.T_0t
  b[-1] = - 2 * lambda *  self.T_Lt

  T = np.zeros((N_t, N_x + 2))
  T[0, :] = self.T_x0(self.xs)
  T[:, 0] = self.T_0t
  T[:, -1] = self.T_Lt
  for j in range(N_t - 1):
    T[j+1, 1:-1] = solve_banded((1,1), Ab, B @ T[j, 1:-1] + b)
  return T
```

Figure 11: Crank Nicolson solution

The software implementation differs slightly from equation (11) since it uses the Scipy function `scipy.linalg.solve_banded`[2], which does a version of the Thomas algorithm for tridiagonal matrices, but it takes as input the banded[5] form of the matrix. Next, it simply computes the matrix B and the vector $\boldsymbol{b}$, then initialises the temperature distribution matrix with the boundary conditions, same as Part I, and then applies the computational molecule row by row.

3.3 Experiments and analysis

In this section, the same 3 experiments in Part I are run to compare between the implicit and the explicit methods.

Figure 12 shows the Crank-Nicolson model gives a closer approximation than the explicit method at the outer inner-nodes and it gets considerably closer to the analytical solution overtime.

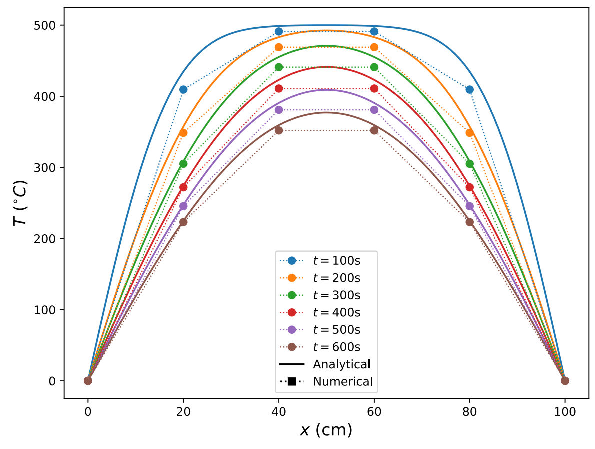

Figure 12: T vs. x, Δx $= 20$cm, Δt $= 100$s CN

This can also be seen at $x = 20\text{cm}$ in direct comparison with the explicit model in figure 13. For every step size, the explicit method yields better results then the other model.

Additionally, at $\Delta x = 10\text{cm}, \Delta t = 100\text{s}$, the implicit method is stable, and gives the best results among all other steps, particularly at the first time-step. This is a feature of the Crank-Nicolson method, since it is unconditionally stable for linear problems.

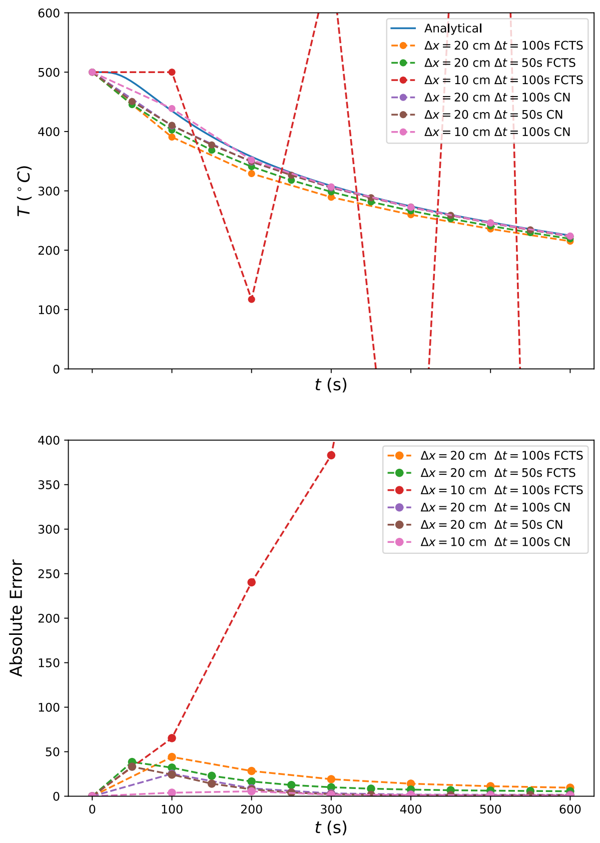

Figure 13: T vs. t at x $= 20$cm

$\Delta x/\Delta t$	20 / 100	20 / 50	10 / 100	analytic
$T(^\circ C)$	223.11	223.48	223.91	224.78
abs. err.	1.66	1.30	0.86	

Table 2: $T(^\circ C)$ and error at $t = 600$s CN

The Temperature Distribution over time and space doesn't look very different, so instead the Absolute Error and the difference between the errors of the explicit and implicit methods are shown in figures 14 and 15.

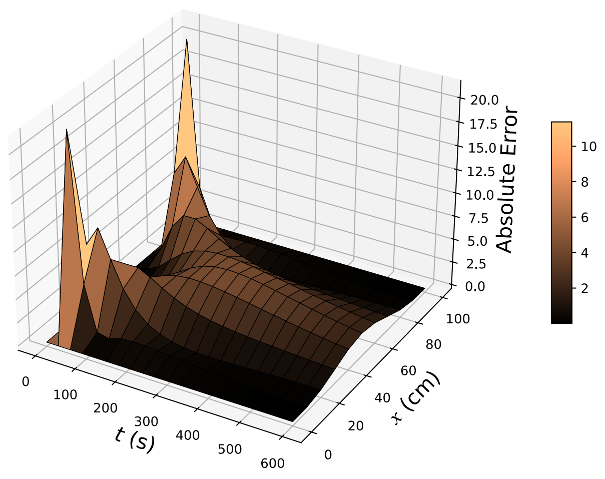

Figure 14: AE at $\Delta x = 10\text{cm}, \Delta\text{t} = 30\text{s}$ CN

In figure 15, blue essentially represents where the implicit method gives a lower error than the explicit method. The mean and median error and the standard deviation were recorded for both models in table 3.

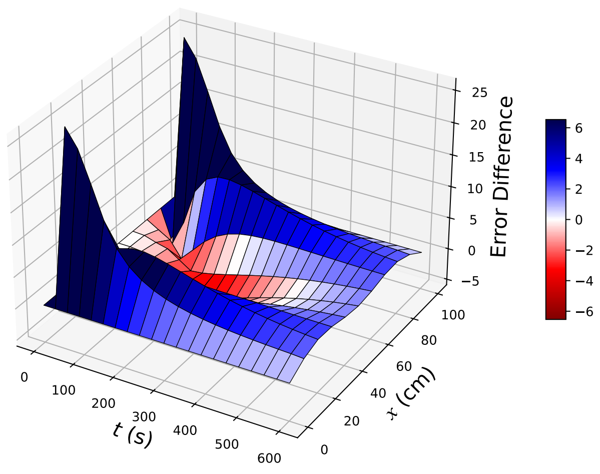

Figure 15: ED at $\Delta x = 10\text{cm}, \Delta\text{t} = 30\text{s}$ CN

model	mean	std	median
explicit	3.73	5.77	2.99
implicit (CN)	1.83	2.67	0.96

Table 3: mean, median and std comparison

Across the board, the Crank-Nicolson method shows great improvement over the explicit model thanks to the unconditional stability and better overall accuracy. This is due to the method's formulation as a weighted average of the *(explicit)* Forward Time Centered Space and the *(implicit)* Backward Time Centered Space methods, effectively balancing between the stability of the implicit scheme and the accuracy of the explicit scheme.

4 Part III: Modelling the Initial Temperature Change

4.1 Problem setup

In this section, the bar's temperature distribution changes (12), and we are interested in the first 60s of cooling.

$$T(x,0) = -xg(x-L)+c \quad g=0.1,\ c=400 \qquad (12)$$

The challenge comes from the lack of an analytical solution to the new model so there isn't a straightforward way to validate the implementation against.

4.2 Software Implementation

The way boundary conditions are applied in software makes it straightforward to accommodate this change. So far, `HeatSolver.T_x0` has been a function that returns 500 at all inputs. It then can be replaced with the one in equation (12)

```
# In the numerical solution method
T[0, :] = self.T_x0(self.xs)

            ...

# When using the solver
hs = HeatSolver(dx=dx, dt=dt, t_total=60)
hs.T_x0 = lambda x: -x * g * (g - hs.L) + c
```

Figure 16: Non-uniform initial state

This means the the rest of the implementation can remain unchanged.

Additionally, to further analyse and compare numerical solutions , a Backward Time Centered Space solution has also been implemented$^{[D]}$, which uses a backward difference approximation for the time derivative. This produces[C] the update scheme in equation (13) and the computational molecule in figure 17.

$$\lambda T_{i+1,j} - (1+2\lambda)T_{i,j} + \lambda T_{i+1,j} = -T_{i,j-1} \qquad (13)$$

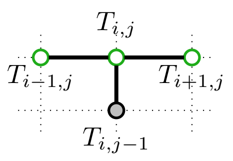

Figure 17: BTCS computational molecule

4.3 Experiments and analysis

Since there is no analytical solution to benchmark against, 3 different experiments are conducted. Firstly, the solution is approximated with CN at different step-sizes (halving Δx and Δt at each iteration) and the difference between the results at the current iteration and the result in the previous iteration (i.e. the approximation with double Δx and Δt) is computed. From this data, the difference across time and space is plotted in figure 18, and the mean and standard deviation of each difference is recorded in table 4.

$\Delta x/\Delta t$	10 / 5	5 / 2.5	2.5/1.25	1.25/0.625
mean	4.27	2.64	0.84	0.22
std.	6.35	4.30	2.07	0.82

Table 4: mean and std. of difference between $2\Delta x$, $2\Delta t$ CN and Δx, Δt CN

The mean over the decreasing step-sizes tends towards 0. This indicates that the approximation is converging on an answer. The standard deviation also tends towards 0; its *larger* values stem from the extremities at the first time steps, where the rate of change of the temperature of the bar is largest, similar to what was observed in parts I and II.

Next, the solutions are approximated at at different step-size resolutions with the CN, FCTS and BTCS models, The CN models' temperature distributions are show in figure 19, together with the difference between the CN and the FTCS and BTCS distributions.

It can be observed that the surface of the difference tends to flatten out overall as the resolution increases, with the exception of the spatial extremities at the first time-steps where the difference tends to grow. This is in accordance with what was seen in Part II, and can be theoretically justified by the idea that both FTCS and BTCS models fail to capture the regions with more abrupt changes in the temperature distribution.

However, the fact that, in general, the difference surface tends to flatten out at 0, in both experiments, indicates that the CNN algorithm is converging on an accurate solution.

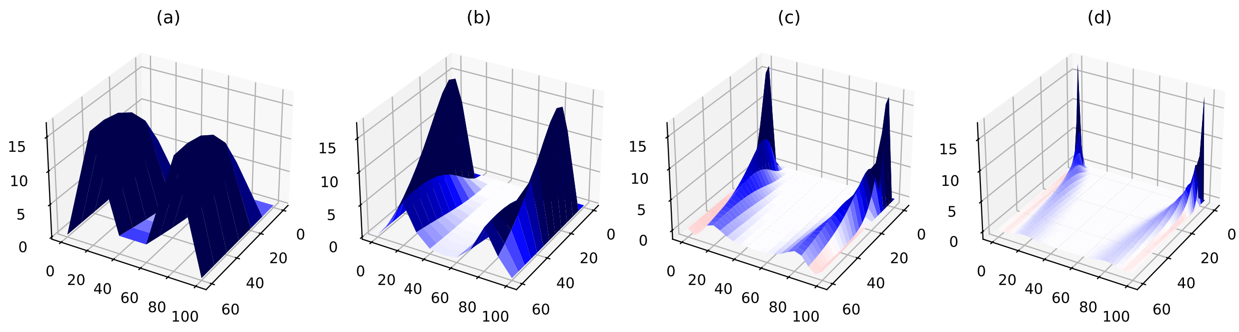

Figure 18: CN Methods difference of previous approximation. Sampled at (a) $\Delta x = 5\text{cm}$, $\Delta t = 10\text{s}$, (b) $\Delta x = 2.5\text{cm}, \Delta\text{t} = 5\text{s}$, (c) $\Delta x = 1.25\text{cm}, \Delta\text{t} = 2.5\text{s}$, (d) $\Delta x = 0.625\text{cm}, \Delta\text{t} = 1.25\text{s}$

The reason behind the differences in figure 19 (.2) and (.3) being inverted in sign can be theoretically justified by the idea mentioned in part II, that the CN model is a weighted average of the FCTS and BCTS models.

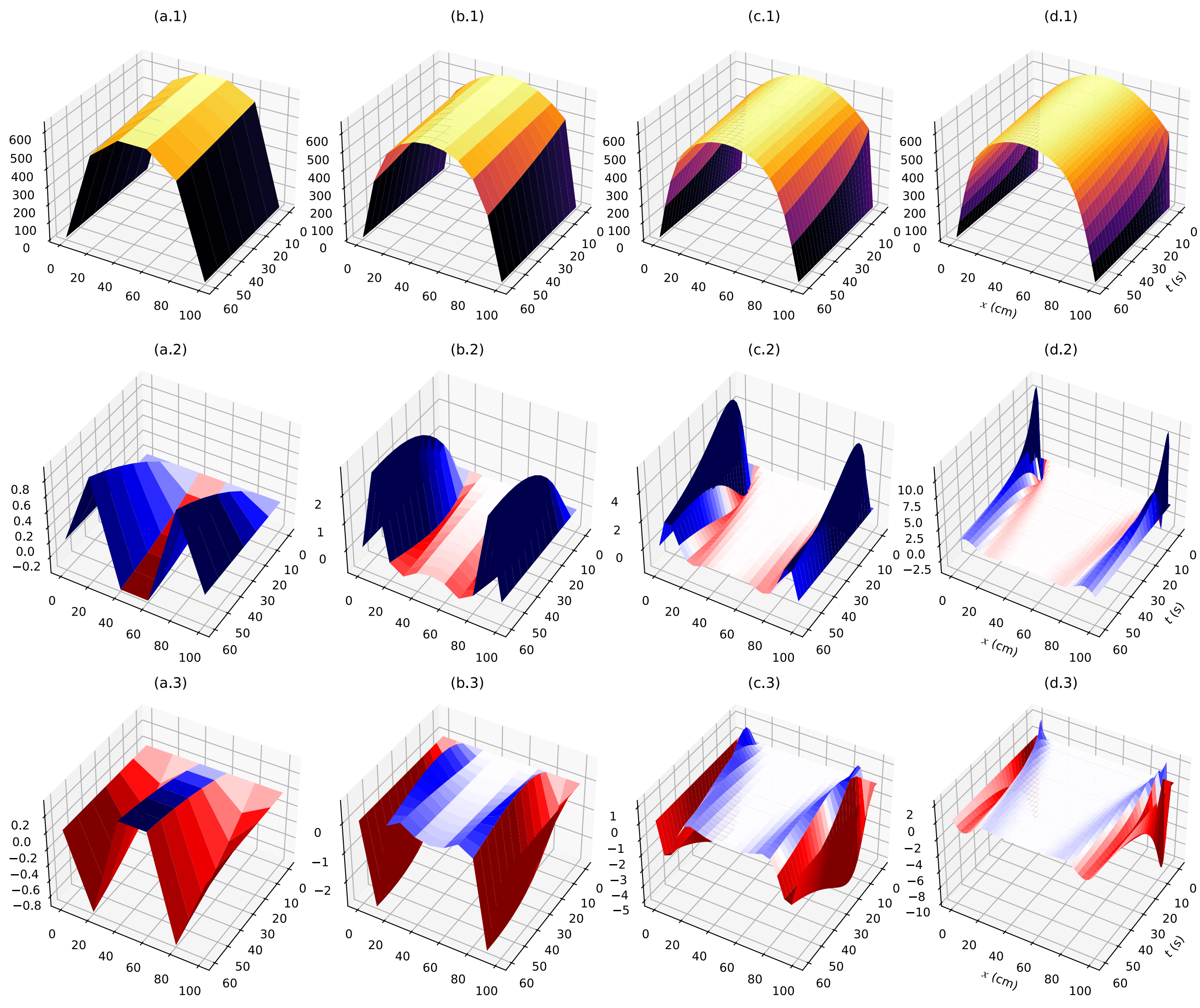

Figure 19: (.1) Temperature Distribution (.2) FTCS vs. CN Difference (.3) BTCS vs. CN Difference. Sampled (a) $\Delta x = 20\text{cm}, \Delta\text{t} = 10\text{s}$, (b) $\Delta x = 10\text{cm}, \Delta\text{t} = 5\text{s}$, (c) $\Delta x = 5\text{cm}, \Delta\text{t} = 2.4\text{s}$, (d) $\Delta x = 2.5\text{cm}, \Delta\text{t} = 1.2\text{s}$

When considering computational effort, the implicit FTCS method is the most lightweight, allowing nodes at the same time-step to be computed independently, which opens the door for parallelisation. Both explicit methods, require solving a linear system of equations, which is more computationally intensive than the FTCS scheme. The difference between BTCS and CN is that the latter takes two more terms into account (i.e. $T_{i-1,j}$ and $T_{i+1,j}$). This justifies why CN tends to give results with lower temperatures than BTCS: the $0°C$ ends of the bar propagate their *coldness* slower overtime, since the BTCS model only takes into account one the *center* node from the previous time-step.

Finally, a **defect** analysis, inspired by *"Check 3"* in this article[3] is performed. The defect is calculated using the numerical approximation of the PDE by subtracting the right-hand side of the equitation from the left-hand side and determine how close to 0 the result is.

$$D(x,t) = \frac{\partial^2 T_{\text{nm}}(x,t)}{\partial x^2} - \frac{1}{k}\frac{\partial T_{\text{nm}}(x,t)}{\partial t} \tag{14}$$

This defect is calculated at every discrete node using (3) and (4) by substituting the numerical solution, giving a metric of the accuracy with which it respects the PDE it was generated from, plotted in figure 20.

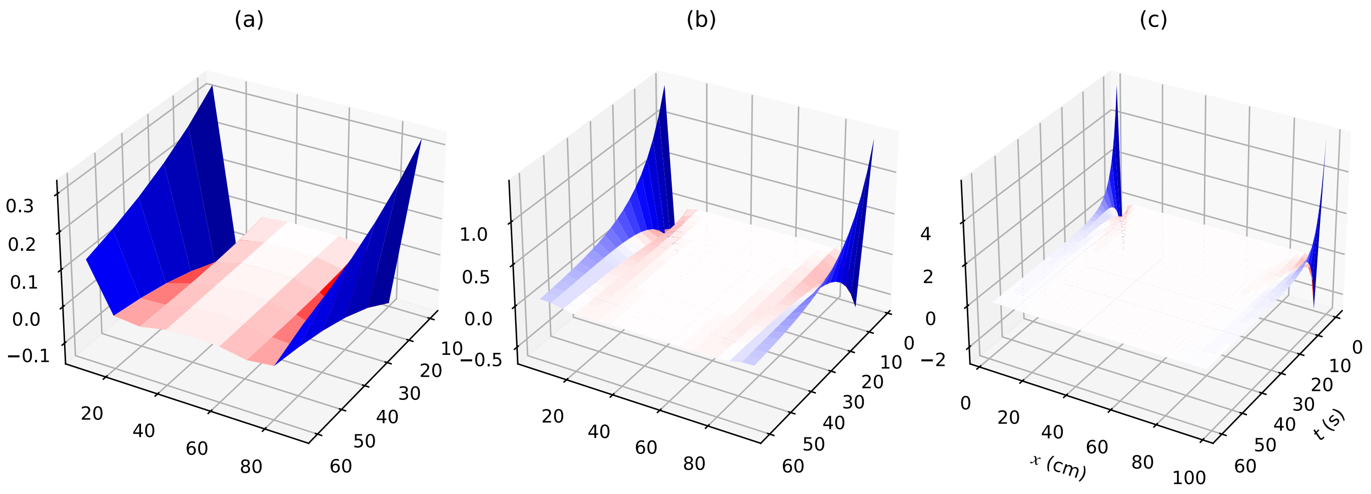

Figure 20: t vs. x vs. Defect. Sampled at (a) $\Delta x = 10\text{cm}$, $\Delta t = 10\text{s}$, (b) $\Delta x = 5\text{cm}, \Delta \text{t} = 2.85\text{s}$, (c) $\Delta x = 2.5\text{cm}, \Delta \text{t} = 0.71\text{s}$

In order to minimise the defect, Δx and Δt where chosen such that $\lambda^{(5)}$ was less then 0.1. This works because if Δx and Δt were to be sampled as they were previously, by being halved, λ, which has an impact in the accuracy of the model, would change more dramatically, giving a large defect at the two extremities in the first time-step. With the ratio preserved, the results obtained are closer to optimal, with the mean defect and standard deviation are recorded in table 5.

$\Delta x/\Delta t$	10 / 10	5 / 2.85	2.5/0.71
mean	0.0265	0.0255	0.0159
std.	0.106	0.177	0.249

Table 5: mean and std. of defect for CN model

This final experiment confirms that the solution on which the CN model converges is accurate. The defect is very close to 0 for all values, except the points with the highest rate of change in the temperature, at the ends of the rod start of the simulation.

4.4 Conclusion

In this section, 3 experiments were conducted: temperature difference between CN models of different sample rates, temperature difference of CN, FTCS and BTCS models, and a defect analysis for the CN. From the results of these experiments, it can be concluded that a robust implementation of the FTCS, CN and BTCS method has been reached. CN gives the most accurate results and FTCS provides an approximation with the least computations per node.

5 Final remarks

This report analysed numerical solutions for the heat conductance equation with parabolic boundary conditions. It was proved that the Crank-Nicolson (CN), Forward Time Centered Space (FTCS), and Backward Time Centered Space (BTCS) methods are effective in solving the heat equation. The comparative analysis highlighted CN's superior accuracy due to its implicit nature and average approximation between time steps. FTCS, being explicit and simpler, offers faster computations but with limited stability and accuracy, particularly for larger time steps. BTCS, similar to CN, provides stability, but with slightly less accuracy.

The experiments conducted underscored the importance of selecting an appropriate numerical method based on the specific requirements of the problem at hand. For problems where accuracy is critical and computational resources are available, the CN method is preferred. However, for quick approximations or when resources are limited, FTCS might be a suitable choice. It's also crucial to consider the stability conditions of each method; for example, FTCS requires adherence to the Courant-Friedrichs-Lewy (CFL) condition to ensure stability.

In conclusion, this report has demonstrated the utility and comparative merits of the CN, FTCS, and BTCS methods for solving the heat equation. By leveraging these insights, engineers and scientists can make informed decisions when modeling heat conduction and other diffusion-based phenomena, leading to more accurate and efficient computational solutions.

References

[1] NumPy Docs. URL: https://numpy.org/doc/stable/reference/generated/numpy.linalg.solve.html.

[2] Scipy Docs. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.solve_banded.html#scipy.linalg.solve_banded.

[3] W.H. Enright. "Verifying approximate solutions to differential equations". In: *Journal of Computational and Applied Mathematics* 185.2 (2006). Special Issue: International Workshop on the Technological Aspects of Mathematics, pp. 203–211. ISSN: 0377-0427. DOI: https://doi.org/10.1016/j.cam.2005.03.006. URL: https://www.sciencedirect.com/science/article/pii/S037704270500110X.

[4] Computational Fluid Dynamics Online. URL: https://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm_-_TDMA_(Thomas_algorithm).

[5] Wikipedia contributors. *Band matrix — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-April-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Band_matrix&oldid=1199530668.

Appendices

A Calculating the update equations for FTCS

From (1), (3) and (4) we get

$$\frac{T_{i+1,j}-2T_{i,j}+T_{i-1,j}}{\Delta x^2}=\frac{T_{i,j+1}-T_{i,j}}{k\Delta t} \qquad |\cdot k\Delta t \tag{A1}$$

$$\overbrace{\frac{k\Delta t}{\Delta x^2}}^{\lambda}(T_{i+1,j}-2T_{i,j}+T_{i-1,j})=T_{i,j+1}-T_{i,j} \tag{A2}$$

$$\implies \lambda T_{i+1,j}-2\lambda T_{i,j}+\lambda T_{i-1,j}=T_{i,j+1}-T_{i,j} \qquad |-T_{i,j} \tag{A3}$$

$$T_{i,j+1}=\lambda T_{i+1,j}+(1-2\lambda)T_{i,j}+\lambda T_{i-1,j} \tag{5}$$

B Calculating the update equations for CN

$$\frac{1}{2\Delta x^2}(T_{i-1,j}-2T_{i,j}+T_{i+1,j}+T_{i-1,j+1}-2T_{i,j+1}+T_{i+1,j+1})=\frac{1}{k\Delta t}(T_{i,j+1}-T_{i,j}) \qquad |\cdot k\Delta t \tag{B1}$$

$$\overbrace{\frac{k\Delta t}{2\Delta x^2}}^{\lambda}(T_{i-1,j}-2T_{i,j}+T_{i+1,j}+T_{i-1,j+1}-2T_{i,j+1}+T_{i+1,j+1})=T_{i,j+1}-T_{i,j}t \tag{B2}$$

$$\implies \lambda T_{i-1,j+1}+(-2-2\lambda)T_{i,j+1}+\lambda T_{i+1,j+1}=-\lambda T_{i-1,j}+(2\lambda-2)T_{i,j}-\lambda T_{i+1,j} \tag{10}$$

C Calculating the update equations for BTCS

Equation (15) gives the backward difference approximation for the time derivative.

$$\frac{\partial T_{i,j}}{\partial t}=\frac{T_{i,j}-T_{i,j-1}}{\Delta t} \tag{15}$$

From (1), (3) and (15) we get

$$\frac{T_{i+1,j}-2T_{i,j}+T_{i-1,j}}{\Delta x^2}=\frac{T_{i,j}-T_{i,j-1}}{k\Delta t} \qquad |\cdot k\Delta t \tag{C1}$$

$$\overbrace{\frac{k\Delta t}{\Delta x^2}}^{\lambda}(T_{i+1,j}-2T_{i,j}+T_{i-1,j})=T_{i,j}-T_{i,j-1} \tag{C2}$$

$$\implies \lambda T_{i+1,j}-2\lambda T_{i,j}+\lambda T_{i-1,j}=T_{i,j}-T_{i,j-1} \qquad |-T_{i,j} \tag{C3}$$

$$\lambda T_{i+1,j}-(2\lambda+1)T_{i,j}+\lambda T_{i-1,j}=-T_{i,j-1} \tag{13}$$

D Coding the BTCS solver for Part III

Unlike the CN solver here, it was chosen to put all the equations (across time-steps) in a single system and solve for all the solutions, this is done for computational efficiency. The update scheme computed in (13) gives the following matrix form, where $\alpha = -(2\lambda + 1)$:

$$\overbrace{\begin{bmatrix} \alpha & \lambda & & & & & & & & \\ \lambda & \alpha & \lambda & & & & & & & \\ & \lambda & \ddots & \ddots & & & & & & \\ & & \ddots & \ddots & \lambda & & & & & \\ & & & \lambda & \alpha & & & & & \\ 1 & & & & & \alpha & \lambda & & & \\ & 1 & & & & \lambda & \alpha & \lambda & & \\ & & 1 & & & & \lambda & \ddots & \ddots & \\ & & & \ddots & & & & \ddots & \ddots & \lambda \\ & & & & 1 & & & & \lambda & \alpha \end{bmatrix}}^{A} \times \begin{bmatrix} T_{1,1} \\ T_{2,1} \\ T_{3,1} \\ \vdots \\ T_{I-1,1} \\ T_{1,2} \\ T_{2,2} \\ T_{3,2} \\ \vdots \\ \vdots \\ T_{I-1,J-1} \end{bmatrix} = \overbrace{\begin{bmatrix} -T_{1,0} - \lambda T_{0,1} \\ -T_{2,0} \\ -T_{3,0} \\ \vdots \\ -T_{I-1,0} - \lambda T_{I,1} \\ -\lambda T_{0,2} \\ 0 \\ 0 \\ \vdots \\ \vdots \\ -\lambda T_{I,J-1} \end{bmatrix}}^{B} \tag{D1}$$

Which can be solved as a equation of the form $A\boldsymbol{x} = B$, with the `numpy.linalg.solve`[1] function.

```
def solve_backward_distance(self) -> np.ndarray:
  lmbd = self.k * self.dt / (self.dx**2)
  alph = -(2 * lmbd + 1)
  N_x = int(self.L / self.dx) - 1
  N_t = int(self.t_total / self.dt)
  N = N_x*N_t

  A = np.diag([alph] * N) + np.diag([1] * (N-N_x), k=-N_x)
  A_T_im1_j = np.tile([lmbd] * (N_x - 1) + [0], N_t)
  A_T_ip1_j = np.tile([0] + [lmbd] * (N_x - 1), N_t)
  A += np.diag(A_T_im1_j[:-1], k=1) + np.diag(A_T_im1_j[:-1], k=-1)

  B = np.zeros(N) + (A_T_im1_j - lmbd) * \
    self.T_0t + (A_T_ip1_j - lmbd) * self.T_Lt
  B[:N_x] -= self.T_x0(np.arange(self.dx, self.L, self.dx))

  T = np.linalg.solve(A, B).reshape((N_t, N_x))
  T = np.insert(T, 0, self.T_x0(np.arange(self.dx, self.L, self.dx)), axis=0)
  T = np.insert(T, 0, self.T_0t, axis=1)
  T = np.insert(T, N_x + 1, self.T_Lt, axis=1)

  return T
```

Figure 21: BTCS solver

E Source Archive

The figures used, along with the Jupyter Notebook, Python script and the TeX source file of this report are hosted on GitHub, at https://github.com/NemoInfo/Mathatical-Systems/tree/main/cw2