National University of Singapore

ME3291 Numerical Methods in Engineering

Numerical Methods: Temperature Distribution (Plate)

Author:

CHUA PING CHAN

A0126623L

April 6, 2017

Contents

1 Introduction

This report details the process of numerically solving the temperature distribution across a 2D square plate over time and discusses important aspects of the numerical methods used as well as the results obtained. Discussions on the approach to the problem and the methods of verifying and testing of programs will also be included.

1.1 Modelling the Physical Problem

The governing equation that describes the temperature distribution on a 2D square plate measuring 1 unit^2 is the heat equation (Brown and Marco, 1958)

$$\frac{\partial T}{\partial t} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right) \tag{1}$$

where T is the temperature of points on a plate, t represents time, α is the thermal diffusivity of the plate's material and x and y are the spacial coordinates in the horizontal and vertical directions respectively. All quantities are in SI units. In this assignment, the case when $\alpha = 1$ is considered. Note that this is not entirely realistic because even the most (thermally) conductive materials on Earth such as gold and pure silver have thermal diffusivity values in the order of magnitude of 10^{-4} m^2/s (Brown and Marco, 1958). However, because the plate material is not specified, it is sufficient to consider the case of $\alpha = 1$ for the mathematical treatment of the heat equation (Equation 1)

$$\frac{\partial T}{\partial t} = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \tag{2}$$

Equation 2 is also referred to as the *Poisson equation*. At steady state, the left side of the equation is 0, resulting in the *Laplace equation*

$$0 = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \tag{3}$$

The above equations can be solved numerically using the finite difference approach. For spacial discretisation, the length of the plate in the x and y directions can be divided into M and N equally spaced segments respectively, resulting in a total of $(M + 1)(N + 1)$ number of nodes across the entire plate.

$$\Delta x = \frac{1}{M} \qquad \text{and} \qquad \Delta y = \frac{1}{N}$$

For simplicity, the square plate is partitioned such that $\Delta x = \Delta y$. For the purpose of standardisation to compare different numerical methods, the values of M and N are both fixed to be 25 wherever possible. The values of M and N will only be changed in discussions related to the effect of partition size on numerical solutions. With this, we have $\Delta x = \Delta y = 0.04$ units.

By letting the origin to be at the bottom-left corner of the plate and i and j to be the indices in the x and y directions respectively, the distance x and y from the origin can be represented as $x = i\Delta x$ and $y = j\Delta y$ respectively. Let k be the index representing time starting from $k = 0$, by employing the second order 5-point central differencing scheme, we have

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{(\Delta x)^2} \qquad \text{and} \qquad \frac{\partial^2 T}{\partial y^2} = \frac{T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k}}{(\Delta y)^2} \tag{4}$$

For temporal discretisation, by employing the first order forward differencing for the partial time derivative of T

$$\frac{\partial T}{\partial t} = \frac{T_{i,j,k+1} - T_{i,j,k}}{\Delta t} \tag{5}$$

On the other hand, employing the first order backward differencing to the partial time derivative of T would give

$$\frac{\partial T}{\partial t} = \frac{T_{i,j,k} - T_{i,j,k-1}}{\Delta t} \tag{6}$$

The temperature distribution across the plate can then be solved numerically by substituting these expressions into the heat equation (Equation 2) and Laplace equation (Equation 3) and applying appropriate boundary conditions. This will be shown clearly later.

1.2 Methodology

Python is used by the author in implementing and programming the necessary functions and numerical methods learned in the first part of ME3291 to solve for the temperature distribution across the plate over time. Snippets of Python codes will be shown where necessary. The complete source code is given in Appendix A.

2 Question 1(a) – Dirichlet Boundary Conditions

In question 1(a), the plate is subjected to Dirichlet boundary conditions, whereby the edges of the plate are held at constant temperatures, with the top edge of the plate maintained at 1 °C and the remaining edges at 0 °C as shown in Figure 1. The initial temperature of all the nodes within the boundary of the plate is 0 °C.

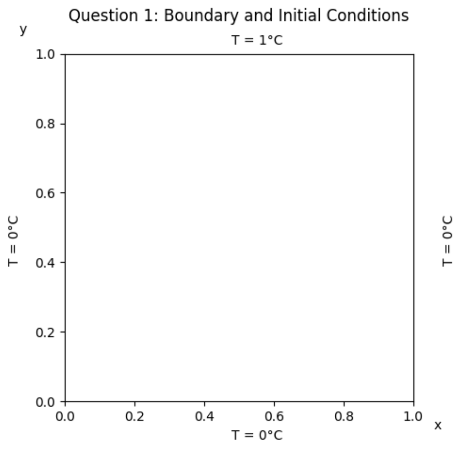


Figure 1: Boundary and initial conditions of question 1.

There are numerous ways of tackling this problem numerically. In this homework assignment, the author attempted both the explicit and implicit methods. These methods have trade-offs which one should take into account when deciding which to employ. These trade-offs will be further discussed in various sections throughout the report, as well as in the *discussion* section.

2.1 Explicit Method

Explicit methods involve directly computing the solution for the present state of a system from the (known) previous states. This is done by first employing the second order 5-point central differencing scheme for spacial discretisation and the first order forward differencing scheme for temporal discretisation to obtain Equation 4 and Equation 5. These expressions are then substituted into Equation 2 to give

$$\frac{T_{i,j,k} - T_{i,j,k-1}}{\Delta t} = \frac{T_{i+1,j,k-1} - 2T_{i,j,k-1} + T_{i-1,j,k-1}}{(\Delta x)^2} + \frac{T_{i,j+1,k-1} - 2T_{i,j,k-1} + T_{i,j-1,k-1}}{(\Delta y)^2} \tag{7}$$

Since $\Delta x = \Delta y$, the equation can be simplified to

$$T_{i,j,k} = \lambda(T_{i-1,j,k-1} + T_{i+1,j,k-1} + T_{i,j-1,k-1} + T_{i,j+1,k-1} - 4 * T_{i,j,k-1}) + T_{i,j,k-1}$$

where $\lambda = \dfrac{\Delta t}{(\Delta x)^2}$. The temperature of each node within the boundary edges can thus be calculated directly from the temperature of nodes from the previous time step. The process can be repeated for every nodes on the plate for every time step increment until the temperature of the plate reaches steady state.

The relative error in temperature, ε_{temp}, of each node at a given time step is calculated as

$$\varepsilon_{temp} = \frac{T_{i,j,k} - T_{i,j,k-1}}{T_{i,j,k}} \times 100\% \tag{8}$$

The temperature distribution across the plate is deemed to have reached steady state when the magnitude of the largest relative error of all the nodes across the plate is less than a prescribed percentage tolerance, ε_s

$$|\varepsilon_{temp}| < \varepsilon_s$$

Scarborough (1966) had shown that if the following criterion is met, we can be assured that the result obtained is correct to at least n significant figures.

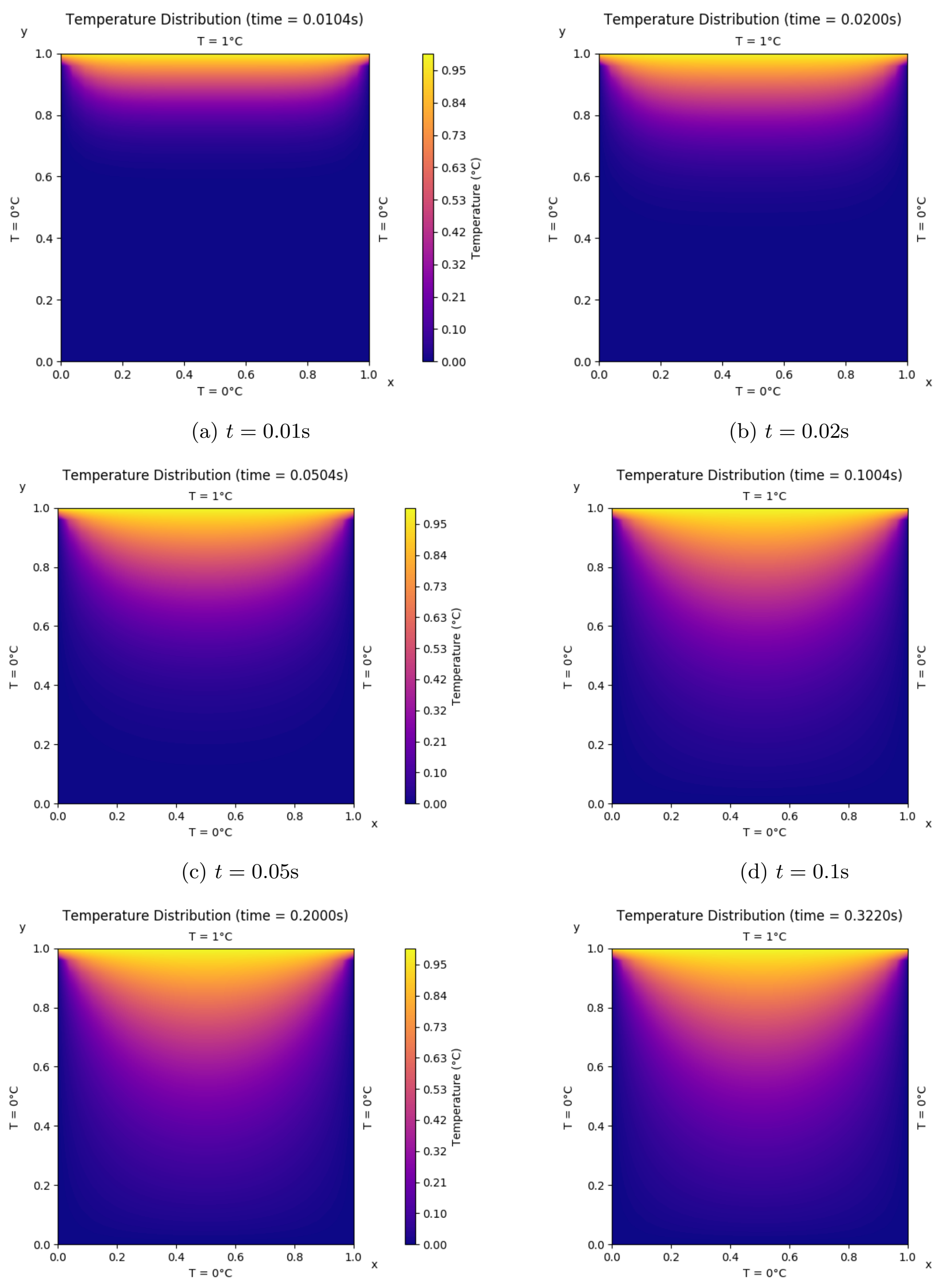
$$\varepsilon_s = (0.5 \times 10^{2-n})\% \tag{9}$$

For the purpose of this assignment, n will be taken to be 4, which gives us $\varepsilon = 0.005\%$. Note that ε_s will be referenced numerous times across this report as it is also used in the determination of the terminating condition of iterative methods later.

As mentioned earlier, different numerical methods have its upsides and downsides. One downside of the explicit method is that the size of the time step, Δt, is limited by a stringent criterion (Chapra and Canale, 2010) in order for the system to be both convergent and stable. That criterion for this particular question is

$$\Delta t \leq \frac{(\Delta x)^2 + (\Delta y)^2}{8} \tag{10}$$

Since $\Delta x = \Delta y = 0.04$ units, the value of Δt used will be 0.0004s. By using all the necessary parameters determined, the temperature distribution across the plate at various time (t = 0.01s, 0.02s, 0.05s, 0.1s, ...) is solved numerically and shown in Figure 2. Six different contour plots at various time are shown to depict the convergence of the result. Using the explicit method, the temperature distribution was found to reach steady state at $t = 0.322$s (Figure 2f).



(a) $t = 0.01$s (b) $t = 0.02$s

(c) $t = 0.05$s (d) $t = 0.1$s

(e) $t = 0.2$s (f) $t = 0.322s$ (steady state)

Figure 2: Numerical solutions to question 1(a) using explicit method.

The advantages of using explicit methods are that they are relatively easy to implement and they usually require less computation. Common sense would tell us that if the plate is divided into smaller partitions, we will be able to have a more 'continuous' model of the plate and simulate the real situation better by reducing *differencing error*. However, due to the criterion specified by Equation 10, as the partition size decreases, the maximum allowable time step also decreases. If the time step used does not satisfy this criterion, truncation errors will magnify and the system becomes unstable.

Problems that involve differential equations that require extremely small step size for explicit methods to be numerically stable are said to be *stiff*. For such problems, implicit methods are used instead, this brings us to the next section.

2.2 Implicit Method

In this section, **question 1(a) is be solved again** using the implicit method. Implicit methods obtain solutions by solving equations involving both the current (unknown) state of the system as well as previous states. Implicit methods which often involve solving systems of equations (simultaneous equation) usually represented in the form of matrices. Due to this, implicit methods require higher computation and are generally harder to program or implement. However, implicit methods are much more stable. For stiff problems that would require impractically small time steps Δt for explicit methods, implicit methods are often be able to produce solutions up to the same accuracy but with lower computational time by using larger time steps. The inherent stability of implicit methods makes them applicable to a much broader range of engineering problems.

By employing the second order 5-point central differencing scheme for spacial discretisation and the first order backward differencing scheme for temporal discretisation, Equation 4 and Equation 6 are obtained, which can be substituted into the heat equation (Equation 2) to give

$$\frac{T_{i,j,k} - T_{i,j,k-1}}{\Delta t} = \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{(\Delta x)^2} + \frac{T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k}}{(\Delta y)^2} \tag{11}$$

Since $\Delta x = \Delta y$, the equation can be rearranged to give

$$(1+4\lambda)T_{i,j,k} - \lambda(T_{i+i,j,k} + T_{i-1,j,k} + T_{i,j+1,k} + T_{i,j-1,k}) = T_{i,j,k-1} \tag{12}$$

where $\lambda = \dfrac{\Delta t}{(\Delta x)^2}$. This equation is for a particular node within the boundary edges of the plate. With $(M-1)(N-1)$ nodes within the edges of the plate, there will be $(M-1)(N-1)$ such equations and $(M-1)(N-1)$ unknowns to solve. This is typically done by constructing a linear matrix equation of the form $Ax = b$ and solving it, where x is a matrix containing the temperatures of nodes across the plate, A is a matrix of their coefficients and b the constants. For the nodes next to the edges of the plate, some of the values of the terms on the left side of Equation 12 are known from the given boundary conditions. These known values are thus shifted to the right side of the equation to be incorporated into matrix b. Matrix b also contains the terms $T_{i,j,k-1}$ on the right side of Equation 12.

There are various ways to solve the given problem implicitly. One way is by using *direct methods*, for example the Gaussian elimination method. A more versatile and efficient way of solving larger scale problem is by using the *iterative methods* (e.g. Jacobi method, Gauss-Seidel method) because the computation time taken is shorter for large matrices. For this homework assignment, the *Jacobi's iterative method* is used. The author implemented the Jacobi's method in Python to solve the plate's temperature distribution numerically.

```python
def jacobi(A, b, x=None):
    """Solves the equation Ax=b via the Jacobi iterative method."""
    # Create an initial guess if needed
    if x is None:
        x = np.zeros(len(A[0]))

    # Create a vector of the diagonal elements of A
    # and subtract them from A
    D = np.diagflat(np.diag(A))
    D_inv = np.diagflat(1/np.diag(A))
    C = D - A

    # Iterate
    x_history = x[np.newaxis,:]
    while not stabalised(x_history, machine_epsilon):
        x = np.diag((b + np.dot(C,x)) * D_inv)
        x_history = np.concatenate( (x_history,x[np.newaxis,:]) )
    return x
```

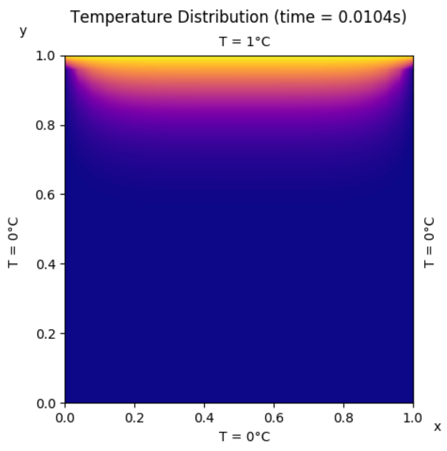
In order to test for the correct functioning of this implementation, the Jacobi's method implemented is tested by comparing its output solutions to simple linear matrix equations with solutions calculated by hand. The computed results for larger linear matrix equations were also tested by comparing them to that of SciPy's linear equation matrix solver.

In order to make a comparison in the time taken to reach steady state with that obtained from the explicit method, the time step used is set to $\Delta t = 0.0004$s. The relative error, ε_a, of each node on the plate at each iteration is calculated as

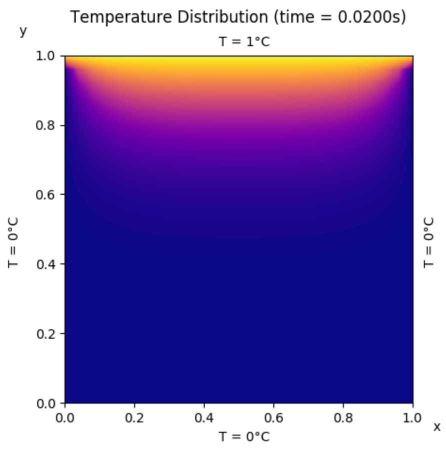
$$\varepsilon_a = \frac{x_n - x_{n-1}}{x_n} \times 100\% \tag{13}$$

where n is the number of iterations.

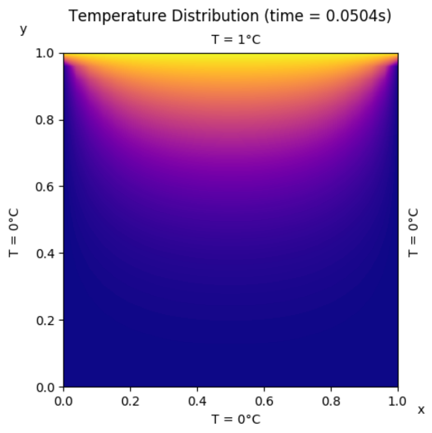
For the iterations at each time step increment, the computed result is deemed accurate enough for iteration to stop when the magnitudes of the relative errors of all nodes on the plate is less than the prescribed percentage tolerance, $\varepsilon_s = 0.005\%$ as calculated from Equation 9. The condition used to determine steady state is the same as that in the previous section, whereby the relative error in temperature is calculated using Equation 8 and ensuring all the nodes on the plate satisfy the condition of $|\varepsilon_{temp}| < \varepsilon_s$. The numerical solution using the iterative method is shown in Figure 3. Using the Jacobi's iteration method, the temperature distribution was found to reach steady state at $t = 0.324$s (Figure 3f), which is very close to that computed using the explicit method (Figure 2f).



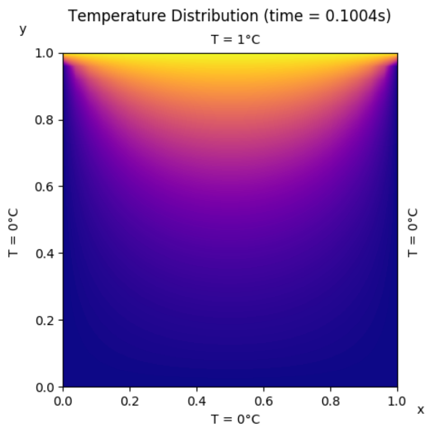
(a) $t = 0.01$s



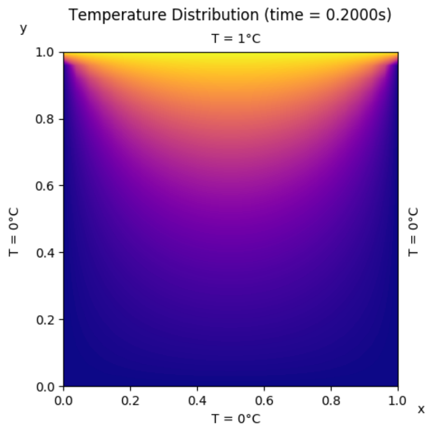
(b) $t = 0.02$s



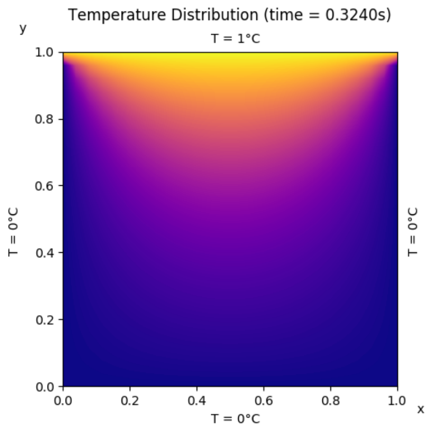
(c) $t = 0.05$s



(d) $t = 0.1$s



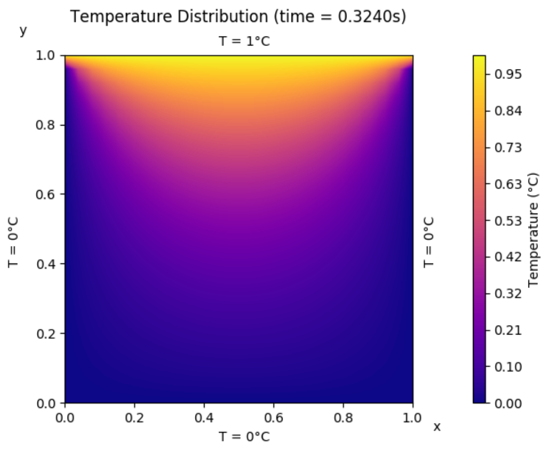
(e) $t = 0.2$s



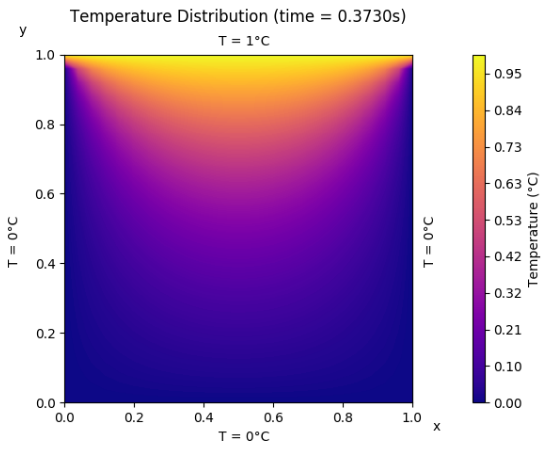
(f) $t = 0.324s$ (steady state)

Figure 3: Numerical solutions to question 1(a) using Jacobi's iterative method.

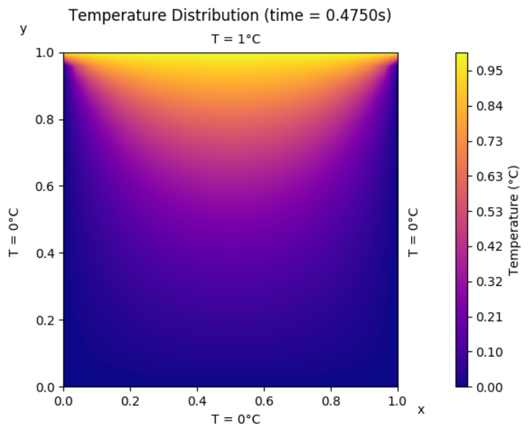
The author also obtained numerical solutions by using time steps Δt of different sizes to observe the effects on the numerical solutions. Some of the solutions obtained at selected time step sizes are shown in Figure 4. It was discovered that the implicit iterative method remains stable regardless of the size of time step, producing identical contour plots. The rate of convergence is also found to increases as the size of time step increases. However, at larger time steps, the accuracy of the calculated total time to reach steady state decreases. This may be attributed to the increase in the accumulation of truncation error in the increased amount of computation. It was also found that as the number of partitions increases, the computation time taken increases. This makes sense as more computation is required for operation such as matrix manipulation and inversion as the size of the matrices increases.



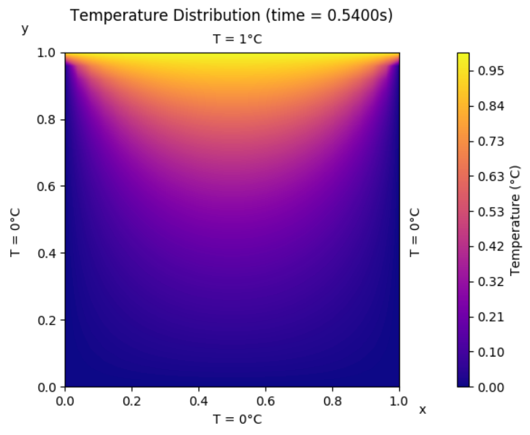
(a) $\Delta t = 0.0004$s



(b) $\Delta t = 0.001$s



(c) $\Delta t = 0.005$s



(d) $\Delta t = 0.01$s

Figure 4: Effect of varying Δt on numerical solutions using implicit method.

3 Question 1(b) – Dirichlet Boundary Conditions (Steady State)

In question 1(b), the boundary conditions are identical to that of part(a). This section requires that, instead of solving for the steady state by marching through time, the steady state can be solved directly by letting $\dfrac{\partial T}{\partial t} = 0$. Thus, from the heat equation (Equation 2) we obtain the Laplace equation

$$0 = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}$$

By employing the second order 5-point central differencing scheme for spacial discretisation to obtain Equation 4 and substituting it into the Laplace equation, we now have

$$0 = \frac{T_{i+1,j,k} - 2T_{i,j,k} + T_{i-1,j,k}}{(\Delta x)^2} + \frac{T_{i,j+1,k} - 2T_{i,j,k} + T_{i,j-1,k}}{(\Delta y)^2} \tag{14}$$

Since $\Delta x = \Delta y$, the Laplace equation can be simplified to

$$T_{i+1,j,k} + T_{i-1,j,k} + T_{i,j+1,k} + T_{i,j-1,k} - 4T_{i,j,k} = 0$$

Similar to how the linear matrix equation is constructed in part(a), the system of $(M-1)(N-1)$ number of equations for each node in the form of Equation 3 can be represented by a linear matrix equation of the form $Ax = b$, where x is a matrix containing the temperatures of all the nodes to solve, A is a matrix of their coefficients and matrix b incorporates the known values of some the terms on the left side of Equation 3 from the boundary condition.

The implementation of the Jacobi's iterative method for part(a) can be reused for this section. To be consistent, the same iteration termination condition as the previous section is used, whereby the relative error of each iteration is calculated using Equation 13 and compared with $|\varepsilon_s| < 0.005\%$, stopping the iteration when $|\varepsilon_a| < \varepsilon_s$.

The steady state condition used for the temperature distribution is also kept consistent with that of part(a), whereby the relative error in temperature is calculated using Equation 8 and steady state is reached when all nodes on the plate satisfy the condition of $|\varepsilon_{temp}| < \varepsilon_s$. With this, the linear matrix equation is solved and the result is shown in Figure 5.

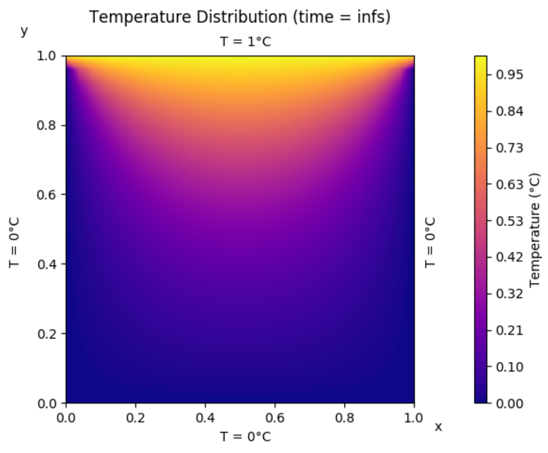


Figure 5: Steady state solution to question 1(b) using Jacobi's iterative method.

The contour plot obtained for steady state is very identical to that in part(a). This is a good sign that the solution is likely to be correct. Compared to the methods in part(a), if the goal is to obtain the temperature distribution contour without needing to know the time elapsed at steady state, then this method (without marching through time) is more efficient as it takes much less iteration and thus computation time.

4 Question 2 – Neumann Boundary Condition

Question 2 imposes the Neumann boundary condition along the right edge of the plate. In this case, the right edge of the plate is insulated. This means that the heat flux is 0 across the right edge of the plate

$$\frac{\partial T}{\partial x} = 0$$

The same Dirichlet boundary conditions apply to the remaining three edges of the plate, with the top edge maintained at 1 °C and the remaining two edges at 0 °C as shown below.

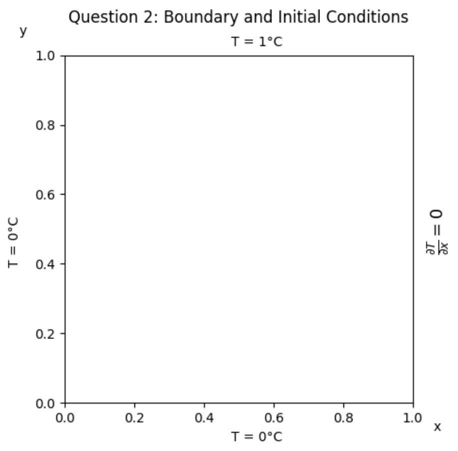

Question 2: Boundary and Initial Conditions
T = 1°C
T = 0°C
T = 0°C
$\frac{\partial T}{\partial x} = 0$
y
x
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Figure 6: Boundary and initial conditions of question 2.

Due to the Neumann boundary condition, the temperatures of the nodes along the right edge of the plate are now unknowns. There are different ways to tackle this problem. In order to reuse the functions and methods implemented in solving question 1, the method used for this homework assignment is to express the temperature of the nodes along the right edge of the plate in terms of the boundary condition derivative.

For each nodes along the right edge of the plate (M, j), by using the first order central differencing scheme, the flux in the x-direction can be represented as

$$\frac{\partial T}{\partial x} \cong \frac{T_{M,j} - T_{M+1,j}}{2\Delta x}$$

where $(M+1, j)$ represents imaginary points outside the right edge of the plate. We can then solve for

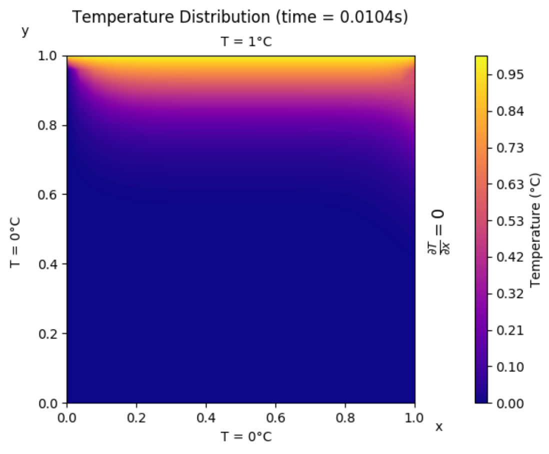
$$T_{M+1,j} = T_{M,j} + 2\Delta x \frac{\partial T}{\partial x}$$

Substituting this expression into Equation 11 and rearranging gives

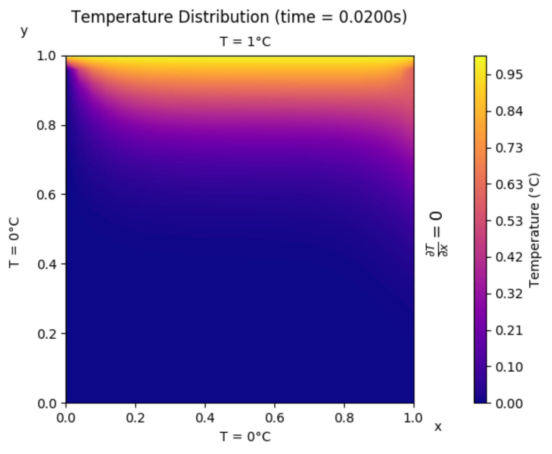
$$(1+4\lambda)T_{i,j,k} - \lambda(T_{i-1,j,k} + T_{i,j-1,k} + 2T_{i,j+1,k}) = T_{i,j,k-1} \tag{15}$$

where $\lambda = \frac{\Delta t}{(\Delta x)^2}$. Unlike question 1, there are now $(M)(N-1)$ number of unknowns and $(M)(N-1)$ number of equations in the form of Equation 15 to solve. Using this system of simultaneous equations, we can form a linear matrix equation in the form of $Ax = b$, where x is a matrix containing the temperatures of the unknowns to solve, A is a matrix of their coefficients, and matrix b contains the temperatures of $T_{i,j,k-1}$ as well as the known values on the left side of Equation 15 from the boundary conditions.

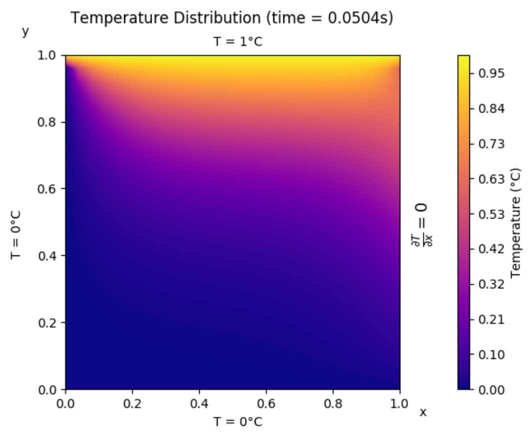
To be consistent with the previous sections, the size of the time step Δt used is 0.0004s. The condition used to terminate the iteration process at each time step increment and the steady state condition used are also the same as that described in question 1, whereby the relative errors of each iterations and the relative errors in temperature distribution at each time steps are calculated and compared to the prescribed percentage tolerance $\varepsilon_s = 0.005\%$, stopping when $|\varepsilon_{temp}| < \varepsilon_s$ and $|\varepsilon_a| < \varepsilon_s$ respectively. With the same initial conditions as before ($T = 0\,°$C at $t = 0$s for the whole domain), the numerical solution obtained is presented in Figure 7.



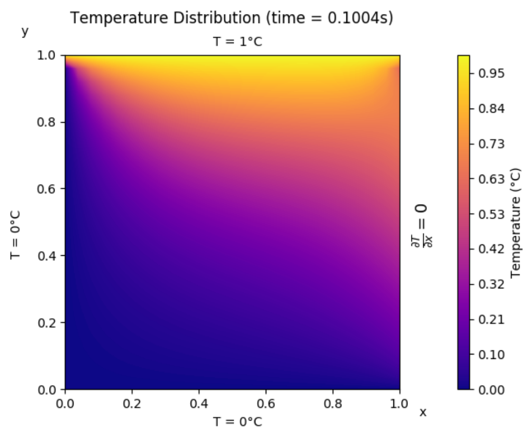
(a) $t = 0.01$s



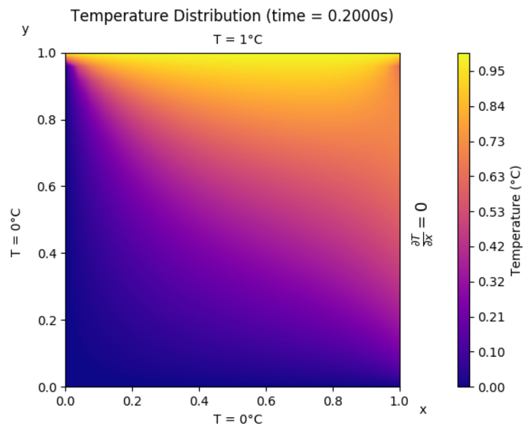
(b) $t = 0.02$s



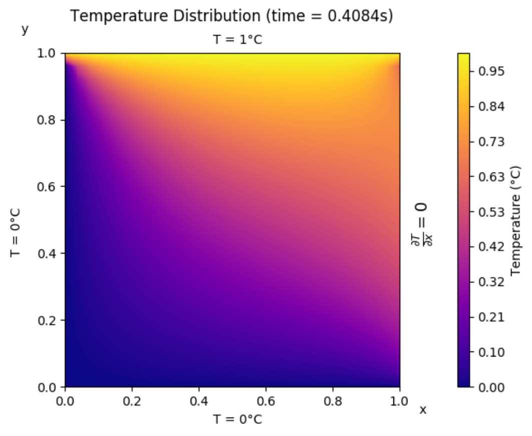
(c) $t = 0.05$s



(d) $t = 0.1$s



(e) $t = 0.2$s



(f) $t = 0.4084s$ (steady state)

Figure 7: Numerical solutions to question 2 using implicit iterative method.

The numerical solutions obtained are consistent with the physics of heat transfer. Compared to solutions of question 1 (Figure 1), the part of the plate along the right edge is at higher temperature. This makes sense because heat cannot 'escape' via the right edge of the plate and thus accumulates more in that region, and the heat flux is deflected downwards and leftwards along the right edge. Note that the darker spot at the top-right corner of the plate is not a bug, it is due to the limitation of finite partitioning of the plate ($M = N = 25$).

5 Discussions

This section will further discuss the process of testing for correctness of the numerical solutions obtained and the trade-offs between explicit and implicit methods. Limitations of the methods used in this homework assignment will also be discussed and possible improvements recommended.

5.1 Correctness of Results

In numerical methods, it can sometimes be tricky to determine whether a solution is correct or not. While working on the homework assignment, there were numerous occurrences when solutions look fine at first sight but was found to be 'buggy' or erroneous upon further scrutiny.

Plotting and visualisation is one of the most useful ways of checking numerical solutions. Contour plots are used for the purpose of this assignment. The contour plot is also animated to observe the transient process for detection of errors in the transient phase.

Programming style also affects the ease of testing programs. By using appropriate comments in the source code, encapsulating frequently used code into modular and generalised functions, implementations may become less error-prone and testing can be made much easier.

Testing with small samples is another very useful way to check the correct functionality of programs. An example that is appropriate for this assignment problem would be to set $M = N = 4$, whereby the systems of equation is small enough to be calculated by hand which can then be used to compare to the output of the program to ensure correct functionality. This method had helped the author to locate and 'catch' numerous 'bugs' while working on the homework assignment.

Comparison between solutions of different methods will also indicate the correctness of programs or methods. This is done in question 1(a) by comparing the results obtained with the explicit method and the implicit iterative method. The steady state solution obtained in question 1(a) is also compared to that of question 1(b).

Understanding the problem well is also very important for an engineer to formulate expectations which can be used as the basis of comparison with the numerical solutions obtained. This can be seen in the discussion of question 2 regarding the consistency of the numerical solution with the physics of heat transfer. The numerical methods are also employed with varying parameters (e.g. size of time step, number of partitions) and the results are observed and compared to expectations based on the physics of heat transfer.

5.2 Trade-Offs: Explicit vs Implicit Method

This section summarises the trade-offs between different numerical methods used in this homework assignment, namely the explicit and implicit methods.

Explicit methods require less computation and are easier to implement and program. However, depending on the problem, it can sometimes be difficult to compute the state of a system solely based on its previous state. Also, due to the limiting criterion for the maximum allowable time step Δt as described by Equation 10, the use of explicit methods to solve *stiff* equations usually requires impractically small time steps to remain convergent and stable.

It is for these reasons that **implicit methods** are more commonly used in real world engineering problems. To achieve a given accuracy in cases where problem are *stiff*, implicit methods are able to produce results in a shorter amount of time as they are stable even when larger time steps are used. The stability of implicit methods allows it to be applied to a much broader range of problem compared to explicit methods. The downsides of implicit methods are that they are usually harder to program and require higher computation.

5.3 Limitations and Possible Improvements

As ME3291 is an introductory course to numerical methods, the author merely learned about the basics of numerical methods which limits the author's ability to solve the assignment problem in the most efficient way possible. This section will outline the various improvements that can be made to improve the efficiency or accuracy of the numerical methods.

First of all, more advanced numerical methods such as the *Alternating Direction Implicit (ADI) method* can be used to solve the problem more efficiently. Furthermore, methods such as the successive over-relaxation method can be used with the Gauss-Seidel method to hasten convergence of iterative methods. Solving the system more efficiently means smaller time steps or larger matrices can be used, which improves the simulation or numerical solution.

Finally, readers should be reminded that the very first assumption made in this homework assignment in letting the thermal diffusivity α in the heat equation (Equation 1) to be 1 m^2/s had already posed limitations to the simulation results in simulating real world situations (even though it is a necessary assumption because the plate's material is not specified). Even the most (thermally) conductive material such as gold and pure silver have thermal diffusivity values in the order of magnitude of 10^{-4} m^2/s (Brown and Marco, 1958). When the thermal diffusivity of pure silver ($\alpha = 1.6563 \times 10^{-4}$ m^2/s) is used by the author with time steps of size $\Delta t = 0.1$s, the total time taken for the temperature distribution to reach steady state was a whopping 16.8 minutes (rather than mere seconds) as shown in Figure 8.

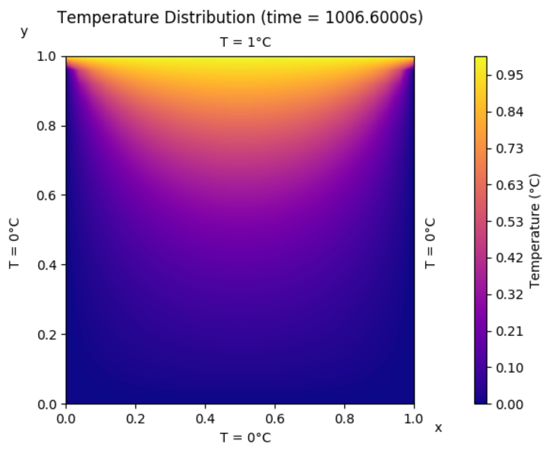


Figure 8: Steady state temperature distribution of pure silver plate.

6 Conclusion

This assignment had helped the author greatly in learning and appreciating the basic concepts of numerical methods. Extensive experimentation was done in programming and testing the numerical methods. Considering the agreement of the numerical solutions with the physics of heat transfer as well as the agreement between the results produced by different numerical methods, the author is confident that the numerical solutions obtained are satisfactory in simulating heat transfer across a square plate.

Bibliography

A. I. Brown and S. M. Marco. Introduction to Heat Transfer, 1958.

S. C. Chapra and R. P. Canale. Numerical Methods for Engineers, 2010.

J. B. Scarborough. Numerical Mathematical Analysis, 1966.

A Program Code

```
# ME3291 Homework Assignment: Temperature Distribution over Plate
# Name: Chua Ping Chan
# Matric no.: A0126623L

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

def calc_machine_epsilon(n):
    """
    n(int): Number of significant figures.
    return(float): Maximum tolerable relative error(%).
    """
    return 0.5 * 10**(2-n)

def status(idx=-1, disp=False):
    print('count =', count)
    print('time = {:1.4f}s'.format(time))
    if disp: print(plate[idx])
    return

def stabalised(plate, machine_epsilon, track=False, count_int=100):
    """Checks for steady state of plate's temperature distribution
    plate[np.ndarray]: Array with number of dimensions >= 2
    machine_epsilon[float]: Maximum tolerable relative error(%)
    track[bool]: Prints max_error every [count_int] num of iteration
    """
    if len(plate) < 2:
        return False
    x_new = np.copy(plate[-1])
    x_old = np.copy(plate[-2])
    error_matrix = abs(x_new - x_old)/x_new * 100
    max_error = np.nanmax(error_matrix)
    ##############################
    # Track max_error: Print status after every 100 time steps(dt)
    if track and (count%count_int == 0):
        status()
        print('max_error =', max_error)
        print()
    ##############################
    return max_error <= machine_epsilon

def solve_1a_explicit(plate, stop_time=np.inf):
    """Solves question 1(a) explicitly.
    stop_time[float]: Elapsed time to stop iteration
    """
    global time, count, dt
    # Overwrite dt - Criteria for convergence
    dt_new = (dx**2 + dy**2) / 8
    if dt_new < dt:
        dt = dt_new
        dt_changed = True
    else:
        dt_changed = False

    plate = apply_boundary_conditions_1(plate)
    while not stabalised(plate, machine_epsilon, track=True) and time < stop_time:
        time += dt
        count += 1
        new = np.copy(plate[-1])
        for j in range(1,N):
            for i in range(1,M):
                new[j,i] = ((plate[-1,j,i+1] - 2 * plate[-1,j,i] + plate[-1,j,i-1])/dx
**2
                          + (plate[-1,j+1,i] - 2 * plate[-1,j,i] + plate[-1,j-1,i])/dy
**2) * k * dt + plate[-1,j,i]
        plate = np.concatenate((plate, new[np.newaxis,:,:]))
    if dt_changed:
        print('NOTE!\ndt changed to {:1.7f}s'.format(dt))
    return plate

```

```python
def apply_boundary_conditions_1(plate):
    """Apply boundary conditions of question 1"""
    plate[:, -1, :] = 1.0    # top
    return plate

def apply_boundary_conditions_2(plate):
    """Apply boundary conditions of question 2"""
    plate[:, -1, :] = 1.0
    return plate

def update_plate_1(plate, lin_arr):
    """
    Updates plate according to question 1.
    Uses apply_boundary_condition functions.
    """
    lin_arr = np.reshape(lin_arr,(N-1,M-1))
    lin_arr = np.pad(lin_arr, 1, 'constant', constant_values=0)
    plate = np.concatenate((plate, lin_arr[np.newaxis,:,:]))
    plate = apply_boundary_conditions_1(plate)
    return plate

def update_plate_2(plate, lin_arr):
    """
    Updates plate according to question 2.
    Uses apply_boundary_condition functions.
    """
    lin_arr = np.reshape(lin_arr,(N-1,M))
    lin_arr = np.pad(lin_arr, 1, 'constant', constant_values=0)
    lin_arr = lin_arr[:,:-1]
    plate = np.concatenate((plate, lin_arr[np.newaxis,:,:]))
    plate = apply_boundary_conditions_2(plate)
    return plate

def numpy_solver(A, b):
    """Solves Ax = b"""
    return np.linalg.solve(A,b)

def solve_1a_implicit(plate, dt, stop_time=np.inf, method=None):
    """Solves question 1(a) implicitly using either the
    Jacobi's iterative method or np.linalg.solve().
    stop_time[float]: Elapsed time to stop iteration
    """
    global time, count
    plate = apply_boundary_conditions_1(plate)
    mlen = (M-1)*(N-1)

    # Construct matrix A of Ax=b.
    # Note that A is dependent on differencing scheme and equation used.
    A = ( (4*lam + 1) * np.eye(mlen)
         - lam * np.eye(mlen,k=1)
         - lam * np.eye(mlen,k=M-1)
         - lam * np.eye(mlen,k=-1)
         - lam * np.eye(mlen,k=1-M) )
    for i in range(M-1, mlen, M-1):
        A[i-1, i] = 0
        A[i, i-1] = 0

    while time < stop_time and not stabalised(plate, machine_epsilon, track=True):
        time += dt
        count += 1

        # Construct matrix b
        # Note that b is dependent on boundary conditions and equation used.
        b = np.zeros((M-1, N-1))
        b += plate[-1, 1:-1, 1:-1]  # Update from previous iteration
        # Apply boundary conditions
        b[-1,:] += lam * plate[0, -1, 1:-1]
        b = b.flatten()

        # Solve system of equations
        if method == 'jacobi':
            x = jacobi(A, b)    # jacobi method
        else:
            x = numpy_solver(A, b)   # np linear matrix eq solver
        # Update plate with solution
        plate = update_plate_1(plate, x)

    return plate

def jacobi(A, b, x=None):
    """Solves the equation Ax=b using Jacobi's method."""
    # Create an initial guess if needed
    if x is None:
        x = np.zeros(len(A[0]))

    # Create a vector of the diagonal elements of A
    # and subtract them from A
    D = np.diagflat(np.diag(A))
    D_inv = np.diagflat(1/np.diag(A))
    C = D - A

    # Iterate
    x_history = x[np.newaxis,:]
    while not stabalised(x_history, machine_epsilon):
        x = np.diag((b + np.dot(C,x)) * D_inv)
        x_history = np.concatenate( (x_history,x[np.newaxis,:]) )
    return x

def solve_1b_implicit(plate, method=None):
    """Solves question 1(b) implicitly using either the
    Jacobi's iterative method or np.linalg.solve()."""
    global time, count
    time = np.inf

    plate = apply_boundary_conditions_1(plate)
    mlen = (M-1)*(N-1)
    # Construct matrix A of Ax=b.
    # Note that A is dependent on differencing scheme and equation used.
    A = (4 * np.eye(mlen)
         - 1 * np.eye(mlen,k=1)
         - 1 * np.eye(mlen,k=M-1)
         - 1 * np.eye(mlen,k=-1)
         - 1 * np.eye(mlen,k=1-M))
    for i in range(M-1, mlen, M-1):
        A[i-1, i] = 0
        A[i, i-1] = 0

    # Construct matrix u
    # u = plate[-1, 1:-1, 1:-1].flatten()

    # Construct matrix b
    # Note that b is dependent on boundary conditions
    b = np.zeros((M-1, N-1))
    b[-1,:] = plate[0, -1, 1:-1]   # top
    b = np.reshape(b, mlen) # Alternative: Use ndarray.flatten()

    # Solve system of equations
    if method == 'jacobi':
        x = jacobi(A, b)    # jacobi method
    else:
        x = numpy_solver(A, b)   # np linear matrix eq solver

    # Update plate
    plate = update_plate_1(plate, x)

    return plate

def solve_2_implicit(plate, dt, stop_time=np.inf, method=None):
    """Solves question 2 implicitly using either the
    Jacobi's iterative method or np.linalg.solve().
    stop_time[float]: Elapsed time to stop iteration
    """
    global time, count
    plate = apply_boundary_conditions_2(plate)
    mlen = (M-1)*(N)
    shape = (N-1,M)

    # Construct matrix A of Ax=b.
    # Note that A is dependent on differencing scheme, boundaryconditions
    # and the governing equation used.
    A = ( (4*lam + 1) * np.eye(mlen)
         - lam * np.eye(mlen,k=1)
         - lam * np.eye(mlen,k=M)
         - lam * np.eye(mlen,k=-1)
         - lam * np.eye(mlen,k=-M) )
    for i in range(M, mlen, M):
        A[i-1, i] = 0
        A[i, i-1] = 0
    for i in range(M-1, mlen, M):
        if (i+M) < mlen:
            A[i,i+M] -= lam

    while time < stop_time and not stabalised(plate, machine_epsilon, track=True):
        time += dt
        count += 1

        # Construct matrix b
        # Note that b is dependent on boundary conditions and equation used.
        b = np.zeros(shape)
        b += plate[-1, 1:-1, 1:]    # Update from previous iteration
        # Apply boundary conditions of question 2
        b[-1,:] += lam * plate[0, -1, 1:]
        b = b.flatten()

        # Solve system of equations
        if method == 'jacobi':
            x = jacobi(A, b)    # jacobi method
        else:
            x = numpy_solver(A, b)   # np linear matrix eq solver
        plate = update_plate_2(plate, x)

    return plate

def update_contour(idx=-1, levels=100):
    """Function to update contour for animation."""
    cont = ax.contourf(X, Y, plate[idx], levels, cmap=plt.cm.plasma)
    return cont

def label_axes(ax, show_title=True):
    """Labels the x- and y- axis and title of contour plot."""
    ax.set_xlabel('x')
    ax.xaxis.set_label_coords(1.07, -0.05)
    ylabel = ax.set_ylabel('y')
    ylabel.set_rotation(0)
    ax.yaxis.set_label_coords(-0.12, 1.05)

    # Label boundary conditions
    if ques == 1:
        fig.text(0.4, 0.9, u'T␣=␣1\u00B0C')   # top
        fig.text(0.05, 0.5, u'T␣=␣0\u00B0C', rotation='vertical')   # left
        fig.text(0.74, 0.5, u'T␣=␣0\u00B0C', rotation='vertical')   # right
        fig.text(0.4, 0.025, u'T␣=␣0\u00B0C')   # bottom
    elif ques == 2:
        fig.text(0.4, 0.9, u'T␣=␣1\u00B0C')   # top
        fig.text(0.05, 0.5, u'T␣=␣0\u00B0C', rotation='vertical')   # left
        fig.text(0.74, 0.5, r'$\frac{\partial␣T}{\partial␣t}␣=␣0$',
                 size=13, rotation='vertical')   # right
        fig.text(0.4, 0.025, u'T␣=␣0\u00B0C')   # bottom

    # Set title
    if show_title:
        ax.set_title('Temperature␣Distribution␣(time␣=␣{:1.4f}s)'.format(time),
                     position=(0.5,1.07))

    return ax

###########################################################
###########################################################
###########################################################
```

```python
# MAIN

# Initialisation
machine_epsilon = calc_machine_epsilon(4)
k = 1           # thermal diffusivity of plate's material
L = 1
M = 25          # No. of divisions per row
N = M           # No. of divisions per column
nrows = N + 1   # No. of nodes per row
ncols = M + 1   # No. of nodes per column
dx = L/M
dy = L/N
dt = 0.0004      # May be changed by explicit method
stop_time = np.inf
if dx == dy:
    dh = dx
    lam = k * dt/(dx**2)
plate = np.zeros((1, nrows, ncols))
time = 0.0
count = 0

###########################################
# Solvers - Uncomment the relevant line for each question.

# Question 1(a)
plate = solve_1a_explicit(plate, stop_time=stop_time); ques=1
##plate = solve_1a_implicit(plate, dt, stop_time=stop_time, method=None); ques=1

# Question 1(b)
##plate = solve_1b_implicit(plate, method=None); ques=1

# Question 2
##plate = solve_2_implicit(plate, dt, stop_time=stop_time, method=None); ques=2

###########################################
# Plotting & Visualisation

# Set plotting parameters
plot_enable = True
animate_enable = False
showplot_enable = True
num_frames = 10
max_frame_idx = count

# Plot/Animate
if plot_enable:
    fig = plt.figure()
    ax = fig.add_subplot(111)

    x = np.linspace(0, 1, ncols)
    y = np.linspace(0, 1, nrows)

    # Label axes
    ax = label_axes(ax)

    X, Y = np.meshgrid(x, y)
    cont = update_contour(levels=200)
    ax.set_aspect('equal')

    # Animate with Func
    if animate_enable:
        frames = range(0,max_frame_idx,int(np.ceil(max_frame_idx/num_frames)))
        ani = animation.FuncAnimation(fig, update_contour, frames=frames,
                                      interval=5, repeat=False)

    cbar = plt.colorbar(cont, ax=ax, aspect=30, format='%.2f', fraction=0.1, pad=0.13)
    cbar.ax.set_ylabel(u'Temperature (\u00B0C)')

    if showplot_enable:
        plt.show()
###########################################
```