ME 3291 Homework Assignment

Q1a) To solve for the temperature distribution equation with the given Dirichlet boundary conditions in Fig 1 in the question paper, I first discretised the spatial (i, j) and time (k) domain of the plot by using finite difference method; namely **central differencing scheme**, using an **iterative approach**.

Central differencing scheme:

Where $u(x, y, t) = U^{k+1}_{x, y}$

Given equation: $\partial T/\partial t = \partial^2 T/\partial x^2 + \partial^2 T/\partial y^2$

Rewriting the given equation using finite difference method:

$$\frac{u_{i,j}^{k+1} - u_{i,j}^{k}}{\Delta t} = \frac{\left(u_{i+1,j}^{k} - 2u_{i,j}^{k} + u_{i-1,j}^{k}\right)}{\Delta x^2} + \frac{(u_{i,j+1}^{k} - 2u_{i,j}^{k} + u_{i,j-1}^{k})}{\Delta y^2}$$

Rearranging, by taking $\Delta x = \Delta y$:

$$u_{i,j}^{k+1} = \Upsilon(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - 4u_{i,j}^{k}) + u_{i,j}^{k}$$

$$where, \Upsilon = \frac{\Delta t}{\Delta x^2}$$

Coding the above equations into the Python Script, the Dirichlet boundary conditions to be included, where the length of x and y of the plate in the code = 10 units, as python script can only take in integers to represent the individual index positions of each element in each matrix.

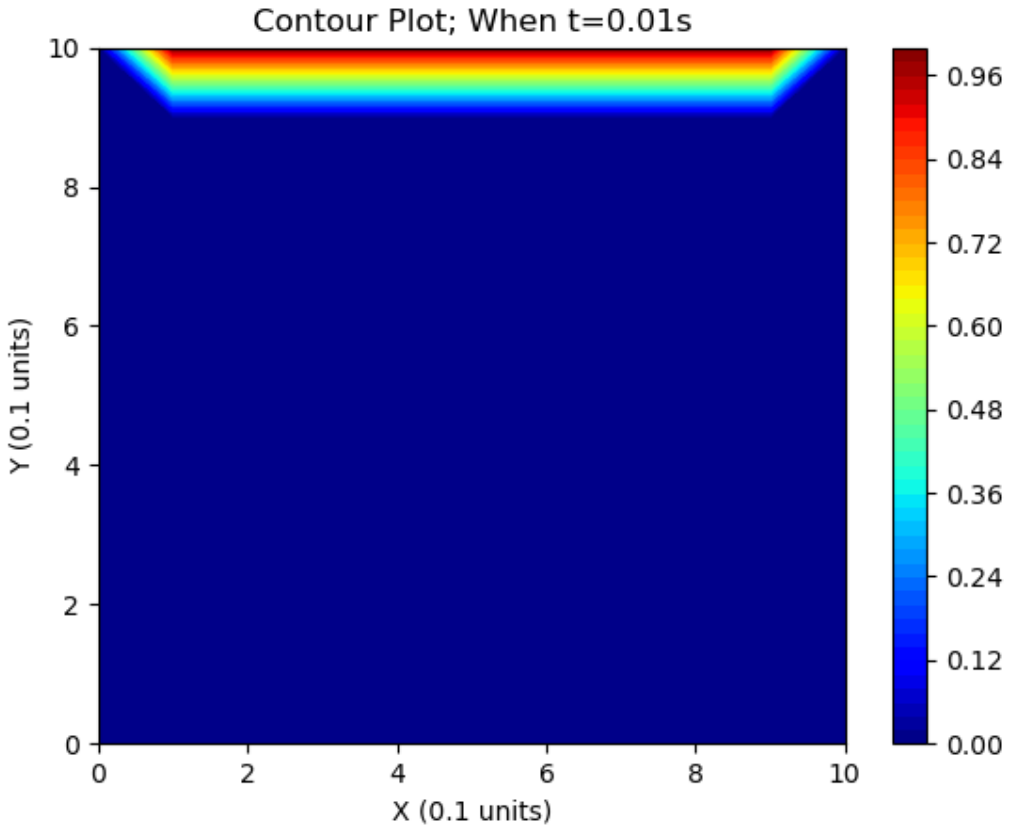
Thus, the delta x in the code = 1, hence the axial units for x and y are 0.1 accordingly to fit the physical condition of having a plate of 1 by 1 units

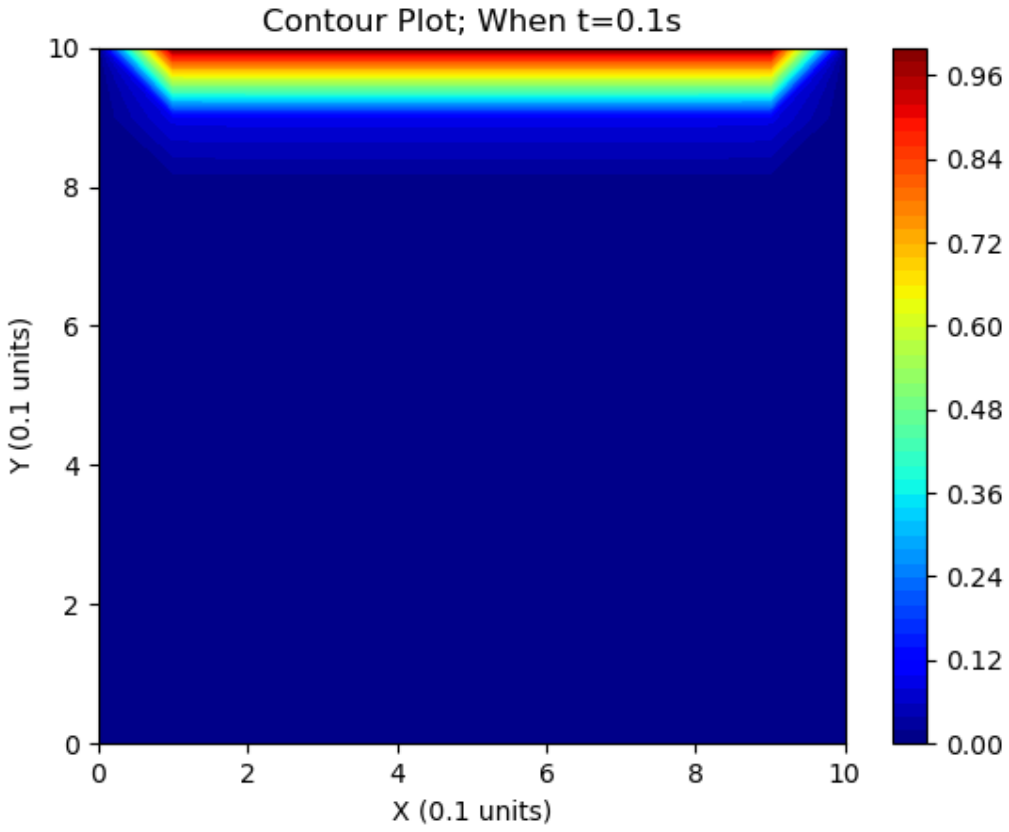
Running the code with a total of 15000 iterations to achieve the convergence plot where each iteration represents 1 unit of Δt, where $\Delta t = 0.01$.

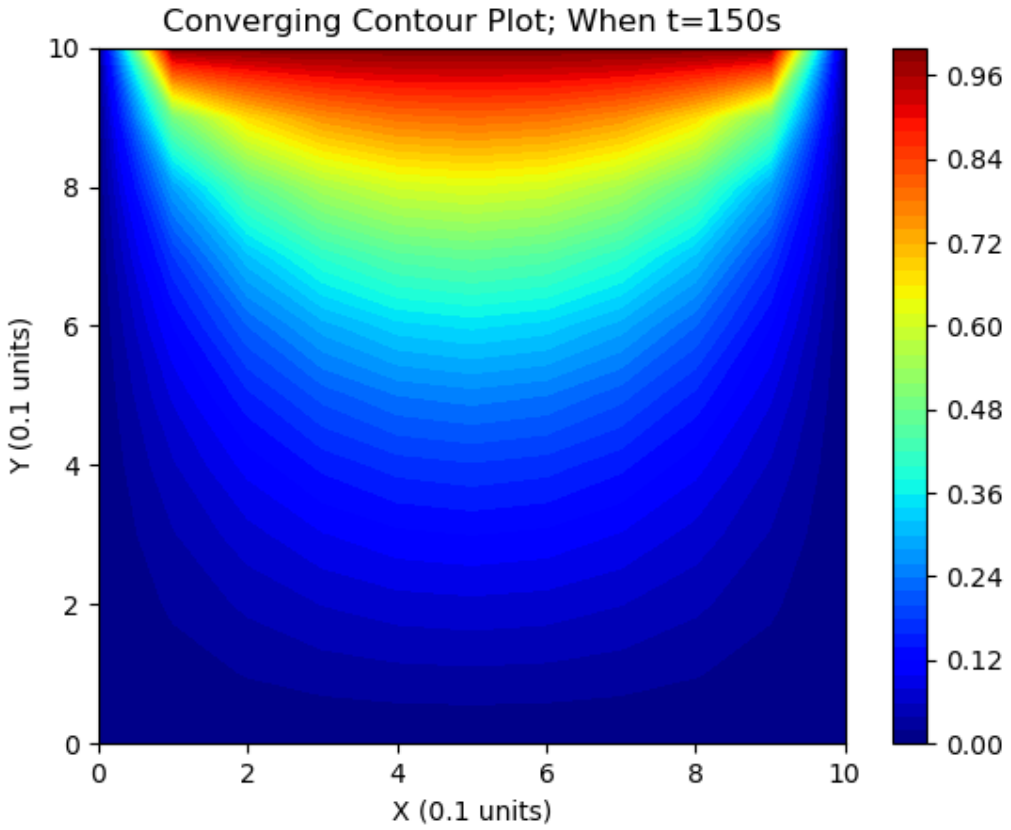
Python code used for solving Q1a)

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

# Design Parameters
# 1 unit of square plot length = 10 units on calculated plot
plate_length = 11 # Compensation for indexing counting from 0 to 11 (total 10)
delta_x = 1 # Each discretized grid space with 0.1 units delta

# Each iteration represent each dt that passes
delta_t = 0.01
iterations = 15000

# After manipulation of given equation: let gamma be dt/(dx^2)
gamma = (delta_t) / (delta_x ** 2)

# Initialize solution: Creating the grid of u(k, j, i)
u = np.empty((iterations, plate_length, plate_length))

# Initial condition everywhere inside the grid
u_initial = 0

# Dirichlet Boundary conditions
u_top = 1.0
u_left = 0.0
u_bottom = 0.0
u_right = 0.0

# Set the initial condition
u.fill(u_initial)

# Set the boundary conditions
u[:, (plate_length-1):, :] = u_top
u[:, :, :1] = u_left
u[:, :1, 1:] = u_bottom
u[:, :, (plate_length-1):] = u_right

# Formula attained from finite difference method
def calculate(u):
    for k in range(0, iterations-1, 1):
        for j in range(1, plate_length-1, delta_x):
            for i in range(1, plate_length-1, delta_x):
                u[k+1, i, j] = gamma * (u[k][i+1][j] + u[k][i-1][j] + u[k][i][j+1] + u[k][i][j-1] - 4*u[k][i][j]) + u[k][i][j]
    return u

u = calculate(u)

# Setting color plot
colorinterpolation = 50
colormap = plt.cm.jet

# Contour plot when t = 0.01
# plt.contourf(u[1],colorinterpolation,cmap=colormap)
# plt.title("Contour Plot; When t=0.01s")

# Contour plot when t = 0.1s
# plt.contourf(u[10],colorinterpolation,cmap=colormap)
# plt.title("Contour Plot; When t=0.1s")

# Converging contour plot when t = 150s
plt.contourf(u[14999],colorinterpolation,cmap=colormap)
plt.title("Converging Contour Plot; When t=150s")

# Show color axis, axis and title
plt.colorbar()
plt.xlabel("X (0.1 units)")
plt.ylabel("Y (0.1 units)")

plt.show()
```

Resulting temperature contour plots of Q1a):






Converging Contour Plot; When t=150s
Y (0.1 units)
X (0.1 units)
0
2
4
6
8
10
0.96
0.84
0.72
0.60
0.48
0.36
0.24
0.12
0.00

Q1b) The Laplace equation $\partial^2T/\partial x^2 + \partial^2T/\partial y^2 = 0$, can be solved using **central differencing scheme** and **direct Gaussian elimination approach**.

Given equation: $\partial^2T/\partial x^2 + \partial^2T/\partial y^2 = 0$

Rewriting the given equation using finite difference method:

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

Rearranging, by taking $\Delta x = \Delta y$:

$$u_{i,j} = \frac{1}{4}(u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1})$$

From the above equation, we can form a numerical stencil to create a matrix in which it contains the coefficients to the solutions that we are required to evaluate to solve for the Laplace equation for the interior nodal points.

Matrix equation obtained for Q1b)

$$\begin{pmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Where a,b,c,d,e,f,g,h,i represents individual nodal points to be found in the solution plot

Using direct Gaussian Elimination:

Running the Python code, we will then be able to solve for the individual nodal points using Gaussian Elimination method. With the solution obtained, we will then also substitute in the given Dirichlet boundary conditions to the top, right left and bottom parts of the solution matrix forming a 5x5 solution matrix; noting that the solution matrix has an "inverted view" of the actual solution plot.

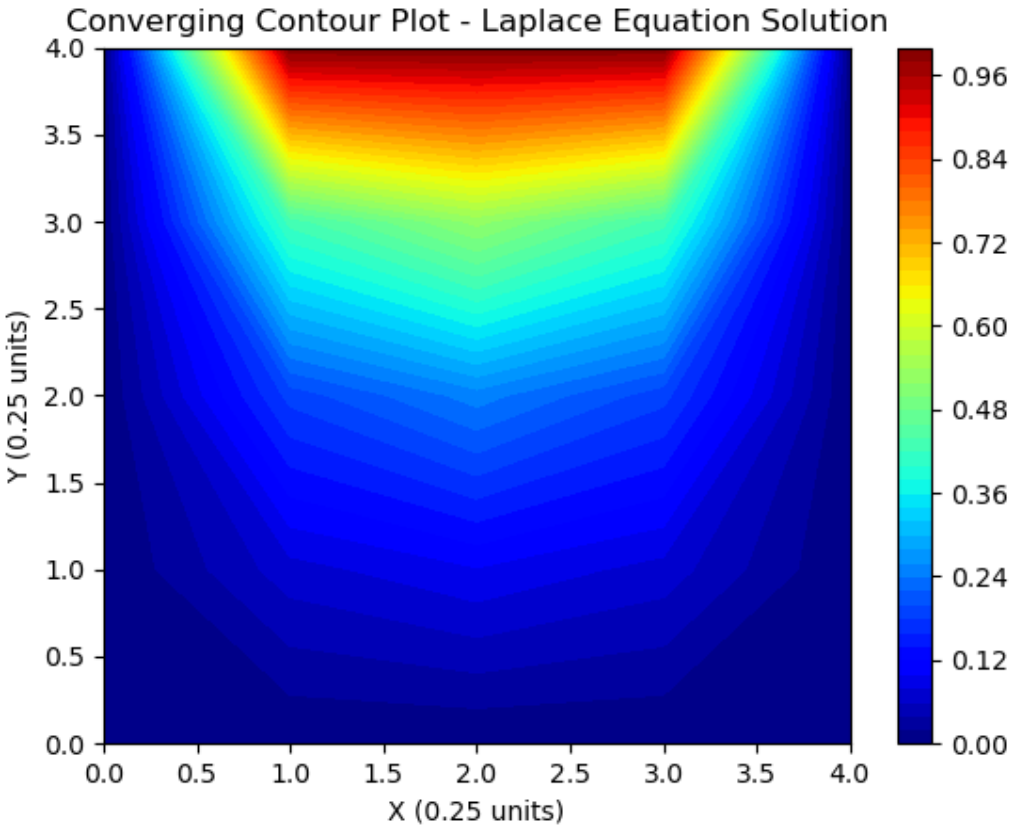
Solution matrix obtained from Python code for Q1b)

```
[[0.         0.         0.         0.         0.        ]
 [0.         0.07142857 0.09821429 0.07142857 0.        ]
 [0.         0.1875     0.25       0.1875     0.        ]
 [0.         0.42857143 0.52678571 0.42857143 0.        ]
 [0.         1.         1.         1.         0.        ]]
```

Python code used for Q1b)

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import inv

# Design Parameters
# 1 unit of square plot length = 25 units on calculated plot
plate_length = 11
delta_x = 1 # Each discretized grid space with 0.25 units delta

# Boundary/ RHS matrix
b = np.empty((plate_length-2,1))
for i in range(len(b)):
    if (i < (plate_length-5)):
        b[i,0] = 0
    else:
        b[i,0] = 1

# Force matrix for triangular matrix
f = np.empty((plate_length-2, plate_length-2))

for i in range((plate_length-2)):
    for j in range((plate_length-2)):
        f[i,j] = 0

# Creating diagonal coefficients matrix
j = 0
for i in range((plate_length-2)):
    f[i, j] = 4
    j+=1

j = 1
for i in range(plate_length-3):
    if i == 2 or i == 5:
        f[i,j] = 0
        j += 1
    else:
        f[i,j] = -1
        j += 1

j = 0
for i in range(plate_length-3):
    if i == 2 or i ==5:
        f[i+1, j] = 0
        j+=1
    else:
        f[i+1, j] = -1
        j+=1

j = 0
for i in range(plate_length-5):
    f[i+3, j] = -1
    j+=1

j = 3
for i in range(plate_length-5):
    f[i,j] = -1
    j+=1

# Matrix inversion calculation and results evaluation
r = inv(np.matrix(f)) * np.matrix(b)

# Computation of final matrix to be processed
row = [0]
results = []

## Applying boundary conditions
first_row = [0,0,0,0,0]
final_row = [0,1.0,1.0,1.0,0]

results.append(first_row)

for i in range(3):
    row.append(float(r[i]))
row.append(0)
results.append(row)

row = [0]

for i in range(3,6):
    row.append(float(r[i]))
row.append(0)
results.append(row)

row = [0]

for i in range(6,9):
    row.append(float(r[i]))
row.append(0)
results.append(row)
results.append(final_row)

results = np.matrix(results)

# Setting color plot
colorinterpolation = 50
colormap = plt.cm.jet

plt.contourf(results ,colorinterpolation,cmap=colormap)
plt.title("Converging Contour Plot - Laplace Equation Solution")

# # Show color axis, axis and title
plt.colorbar()
plt.xlabel("X (0.25 units)")
plt.ylabel("Y (0.25 units)")

plt.show()
```

Solution plot for Q1b):



Comparison of convergence plot between Q1a) and Q1b)

Comparing the two convergence plot of Q1a) and Q1b), due to the smaller discretised delta x and delta y in Q1a), the plot in Q1a) is finer and show more details which thus mean that it has a smoother curvature around turning points as compared to the convergence solution plot in Q1b). But noting that numerically, both the convergence solution plots of both Q1a) and Q1b) should be similar; even though they were obtained using different approaches.

Q2) To solve for the same governing equation for temperature distribution with time, we will again employ the same finite difference method to the solution, **central differencing scheme**, using an **iterative approach**.

Central differencing scheme:

Where u(x, y, t) = $U^{k+1}_{x, y}$

Given equation: $\partial T/\partial t = \partial^2 T/\partial x^2 + \partial^2 T/\partial y^2$

Rewriting the given equation using finite difference method:

$$\frac{u_{i,j}^{k+1} - u_{i,j}^{k}}{\Delta t} = \frac{\left(u_{i+1,j}^{k} - 2u_{i,j}^{k} + u_{i-1,j}^{k}\right)}{\Delta x^2} + \frac{(u_{i,j+1}^{k} - 2u_{i,j}^{k} + u_{i,j-1}^{k})}{\Delta y^2}$$

Rearranging, by taking $\Delta x = \Delta y$:

$$u_{i,j}^{k+1} = \Upsilon(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - 4u_{i,j}^{k}) + u_{i,j}^{k}$$
$$where, \Upsilon = \frac{\Delta t}{\Delta x^2}$$

The main difference from Q2) to Q1a) is the employment of a Neumann boundary condition rather than the Dirichlet boundary condition used in Q1a) on the right-hand side of the plate. To account for the Neumann boundary condition of, $0 \leq y \leq 1.0$, x = 1.0, given as $\partial T/\partial x$ = 0.0, we will proceed with backward difference method for the boundary in which the Neumann boundary condition is applied to.

Neumann boundary condition using backward difference:

$$\frac{\partial u_{i,j}^{k}}{\partial x} = \frac{3u_{i,j}^{k} - 4u_{i-1,j}^{k} + u_{i-2,j}^{k}}{2\Delta x}$$

Since the given condition is $\partial T/\partial x = 0.0$, thus $\frac{\partial u_{i,j}^{k}}{\partial x} = 0$

Rearranging the equation:

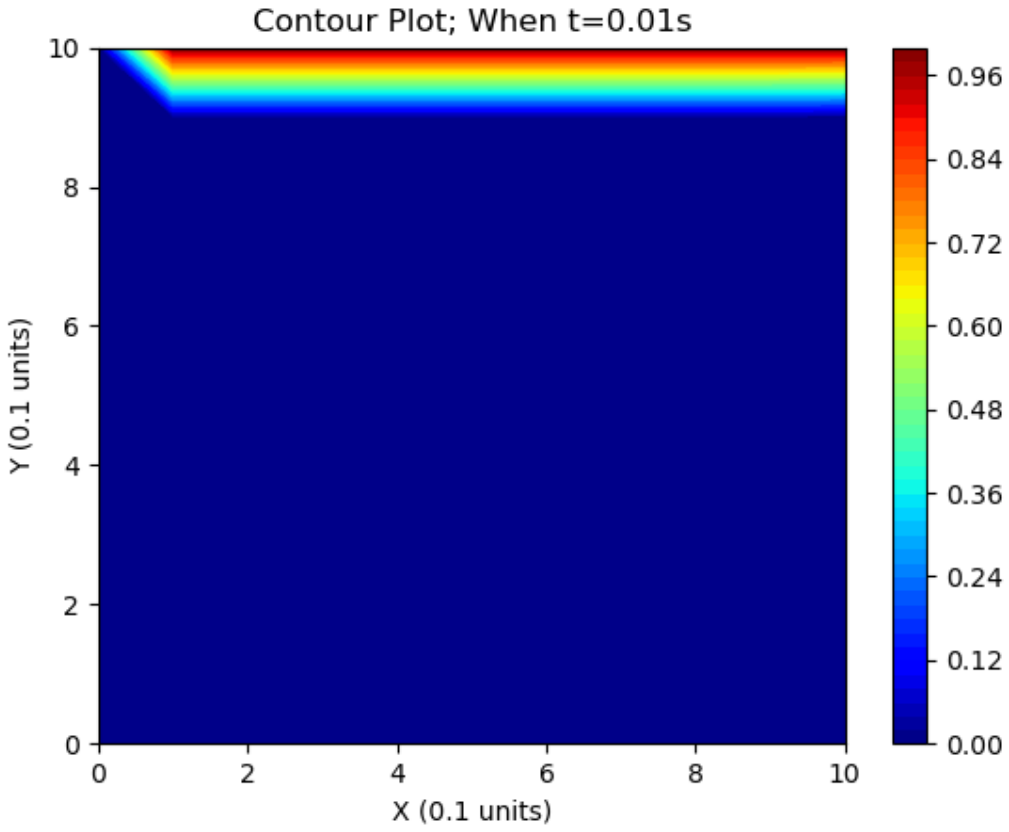
$$u_{i,j}^{k} = \frac{4u_{i-1,j}^{k} - u_{i-2,j}^{k}}{3}$$

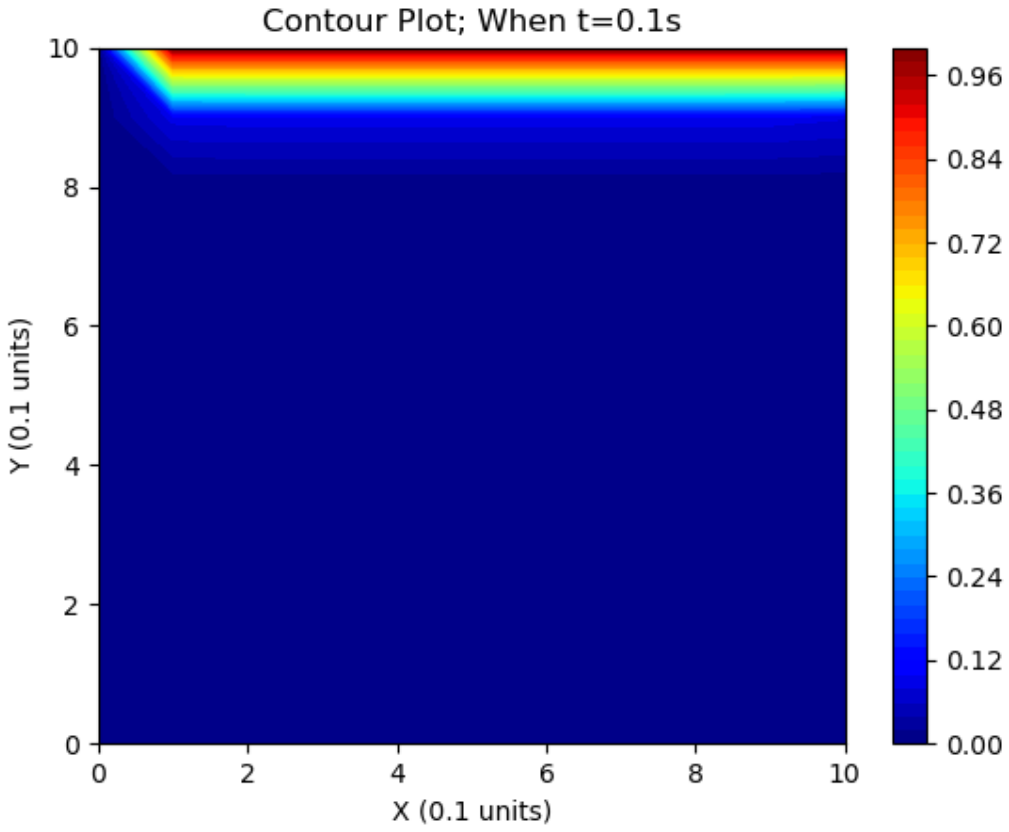
Where the above equation will be used to determine the Neumann boundary conditions on the right hand right of the 2D plate, where the rest of the sides (top, left and bottom) are calculated using the same structure of code and methodology to account for the Dirichlet boundary condition for the forementioned sides of the 2D plate.

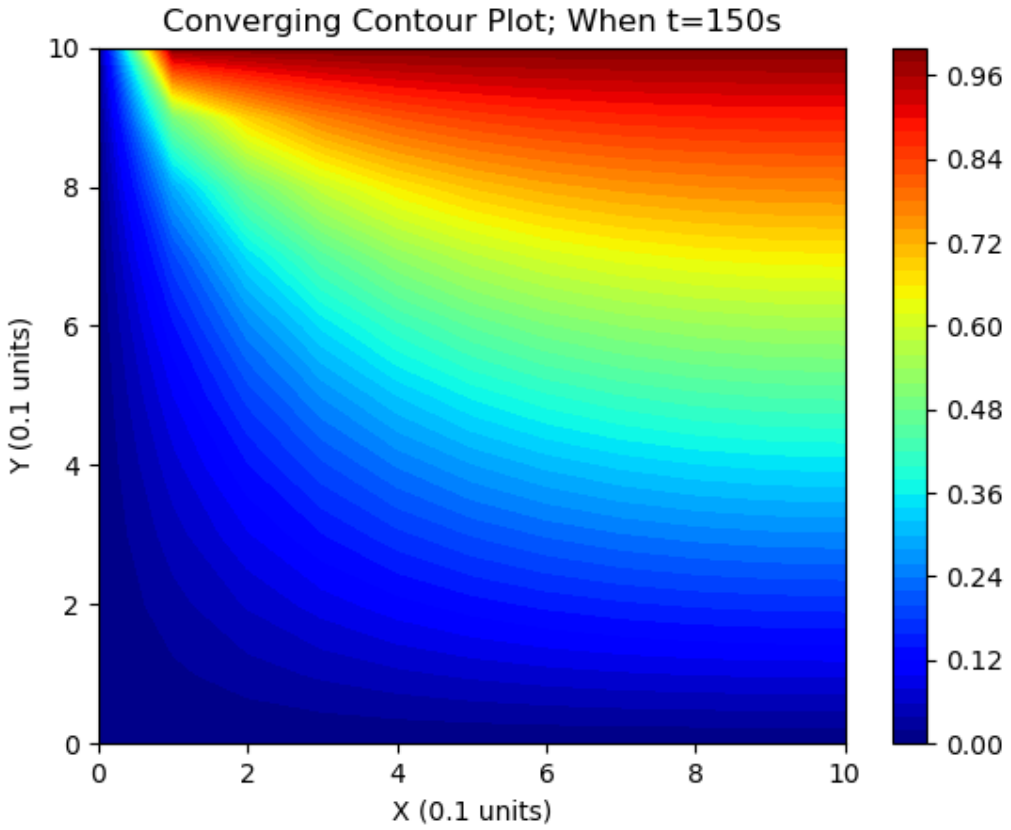
Python code used for Q2)

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

# Design Parameters
# 1 unit of square plot length = 10 units on calculated plot
plate_length = 11 # Compensation for indexing counting from 0 to 11 (total 10)
delta_x = 1 # Each discretized grid space with 0.1 units delta

# Each iteration represent each dt that passes
delta_t = 0.01
iterations = 15000

# After manipulation of given equation: let gamma be dt/(dx^2)
gamma = (delta_t) / (delta_x ** 2)

# Initialize solution: Creating the grid of u(k, j, i)
u = np.empty((iterations, plate_length, plate_length))

# Initial condition everywhere inside the grid
u_initial = 0

# Dirichlet Boundary conditions
u_top = 1.0
u_left = 0.0
u_bottom = 0.0

# Neumann Boundary Condition
## Using backward differencing: u[i,j] = u[i-1,j]
### Implementation further in the code

# Set the initial condition
u.fill(u_initial)

# Set the boundary conditions
u[:, (plate_length-1):, :] = u_top
u[:, :, :1] = u_left
u[:, :1, 1:] = u_bottom

# Formula attained from finite difference method
def calculate(u):
    for k in range(0, iterations-1, 1):
        for j in range(1, plate_length-1, delta_x):
            for i in range(1, plate_length-1, delta_x):

                # Calculation for Neumann boundary condition of right side of plate
                if i == plate_length-2:
                    # u[k+1, j, i+1] = u[k+1, j, i]
                    u[k+1, j, i+1] = (4*u[k, j, i] - u[k, j, i-1])/3
                else:
                    pass
                u[k+1, j, i] = gamma * (u[k][j+1][i] + u[k][j-1][i] + u[k][j][i+1] + u[k][j][i-1] - 4*u[k][j][i]) + u[k][j][i]
    return u

u = calculate(u)

# Setting color plot
colorinterpolation = 50
colormap = plt.cm.jet

# Contour plot when t = 0.01
# plt.contourf(u[1],colorinterpolation,cmap=colormap)
# plt.title("Contour Plot; When t=0.01s")

# Contour plot when t = 0.1s
# plt.contourf(u[10],colorinterpolation,cmap=colormap)
# plt.title("Contour Plot; When t=0.1s")

# Converging contour plot when t = 150s
plt.contourf(u[14999],colorinterpolation,cmap=colormap)
plt.title("Converging Contour Plot; When t=150s")

# Show color axis, axis and title
plt.colorbar()
plt.xlabel("X (0.1 units)")
plt.ylabel("Y (0.1 units)")

plt.show()
```

Resultant temperature contour plots for Q2):






Converging Contour Plot; When t=150s
Y (0.1 units)
X (0.1 units)
0
2
4
6
8
10
0.96
0.84
0.72
0.60
0.48
0.36
0.24
0.12
0.00