Thermodynamics Coding Project

Shiraz Wasim

April 22, 2016

Contents

1 Introduction

1.1 Introduction

Computer-based modelling has become an integral part of mathematical modelling for various natural systems, particularly within engineering. It becomes particularly useful when estimating the performance of a system that is too complicated for an analytical solution. The purpose of this project is to calculate, using a relaxation method, the steady-state temperature distribution across a rectangular plate which is heated and cooled on opposite corners. Edges that aren't being heated or cooled are described as insulating; this means that there is no heat flux through this part of the surface. Furthermore, the plate contains a hole of radius 0.05m that can be placed at any location on the plate. A highly conductive fluid passes through this hole; consequently, the temperature at any point inside the hole is equal to the average of the points at its edge.

Other than the ability to solve the problem including a default case, there are several other requirements for the written code. These include user friendliness, the efficient use of logic (brevity), tolerances to incorrect user input and speed of execution. Although all important, particular emphasis is required on the optimisation of the code to achieve the best possible execution times.

However, before the code can be written, a fundamental understanding of the physical concepts behind the calculation is required.

1.2 Physical Context

In order to calculate the temperature distribution, the plate is first modelled as a rectangular mesh. Δx and Δy describe the distances between the nodes along the length and width of the plate respectively. Through implemention of Fourier's law of Thermal conduction, Fick's 2nd law of diffusion and a bit of manipulation, we can derive Laplace's equation. This equation is satisfied by the steady-state (in which time derivatives become negligible) temperature distribution, T(x, y), and is shown below:

$$\nabla^2 T = \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) = 0 \tag{1}$$

An assumption made here to apply this formula is that the plate does not generate any internal heat.

In order to code this in matlab, we require a numerical solution to Laplace's equation. The relaxation method mentioned above uses an initial guess to begin the process that converges this value towards the correct result. The equation that does this is:

$$T_{i,j}^{n+1} \approx \frac{T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} + T_{i,j-1}^{n}}{4} \tag{2}$$

This equation is applied on all the nodes within the mesh. However, $T_{i,j}^{n+1}$ is not calculated for the boundary conditions. These sections of the boundary are heated and cooled at specific values: assigned the next inner-most values within the mesh as they are on the insulated boundary layer: and are on the circle and thus are given values by averaging the temperatures on its edge.

2 Methodology

The code is split up into three m files, 'Main','UserInput' and 'HeatTransferProblem'. The 'Main' file allows the user to specify values: runs several grid spacings starting with low grid resolutions up until the final required grid resolution :and uses outputed values from the 'HeatTransferProblem' function to plot the results. The 'HeatTransferProblem' executes the numerical simulation to define the steady-state temperature distribution on the plate for specified input values including: matrices for the initial guesses for T_old, the X and the Y values: the location of the centre of the circle: the temperature convergence criterion: the heating and cooling conditions at the boundary: and the dx/dy values for the current loop. The function then returns an output of 'Tconv' which is the matrix that describes the temperature distribution for the current loop along with its X and Y values. The following sections explain the methodology in further detail through a series of flowcharts that break up the code into several different sections.

2.1 Main Function

This is the m file on which the program is run. Figure 1 describes this process as a flow chart. 'UserInput' is the function that allows the user to choose whether or not to start with the default values or to input their own. This section of the code is done through a series of if statements. One of these inputs includes 'Optsteps.' This variable defines the amount of iterations for which a coarser mesh will be refined through interpolation and rerun through the loop before reaching the required result. The next few lines of the code define dxstep and dystep. These are vectors of length 'optsteps' that contain the values of dx and dy that will be used for each loop. As dy is required to equal dx, dxstep and dystep can also be equated. 'T_init', 'Xinit' and 'Yinit' are inputed into the 'HeatTransferProblem' function and the simulation is run. 'T_init', 'Xinit' and 'Yinit' are then equated to 'Tconv', X and Y (the outputs of the simulation) respectively and then inputed back into the function again. This is repeated for 'Optsteps' iterations and then the final 'Tconv' outputed as well as the 'errsave' vector are plotted. The plots include a surface contour plot of X and Y position along with temperature in Kelvin: an above view of the same surface plot mentioned before: and an iteration count verses temperature change plot showing the convergence.

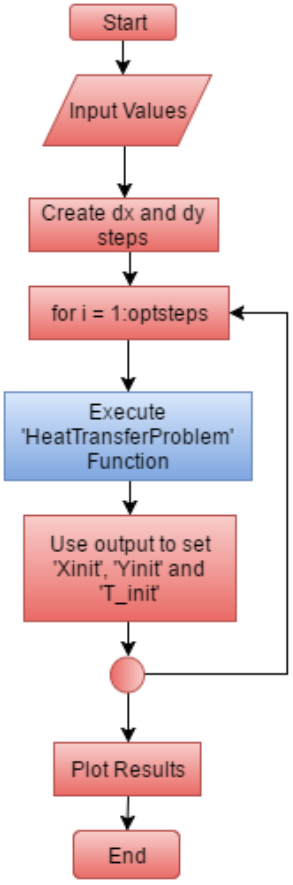

Figure 1: Coding Processes executed by Main.m File

2.2 Heat Transfer Function

The 'HeatTransferProblem' function is described in Figure 2. The code begins by calculating values for 'nx' and 'ny' by dividing the length and width by dx and dy respectively. As this value needs to be an integer the 'ceil' function is used. Consequently, in the following section an if statement is coded to determine whether or not the dx value is altered because of this. If this is the case a message will be displayed to the user to inform them of this. Another error check performed is for the placement of the circle. For this code the circle has to be within the edge of the boundary otherwise it will not work. If this occurs a message will be displayed to the user to tell them that their coordinates for the centre of the circle were not within the specified range. The meshgrid section calculates the coordinates of all grid points and identifies which gridpoints belong to the circle and its edge. The storage vectors section

predefines the size of the matrix for T_old only if its the first loop. The vector where the error is saved is also predefined in this section. The benefit of doing this is that the vector length does not need to increase after each loop. The initial boundary conditions section enforces boundary conditions on the initial T_old that has been inputed. This includes the dirichlet, insulated and circle boundary conditions. T_new is set equal to T_old before the while loop starts. The while loop then calculates the T_new values, resets the insulated and circle boundary conditions, finds the total error and then sets T_old to T_new. Finally, at the end of the code 'Tconv' is saved as the final value of 'T_new' and is outputed to the 'Main' file so that it can be used to plot the results or be used in the 'HeatTransferProblem' function again for the next dx/dy step . The 'Define Mesh', 'Set Initial Boundary Conditions and 'Execute while loop' sections are described in more detail below.

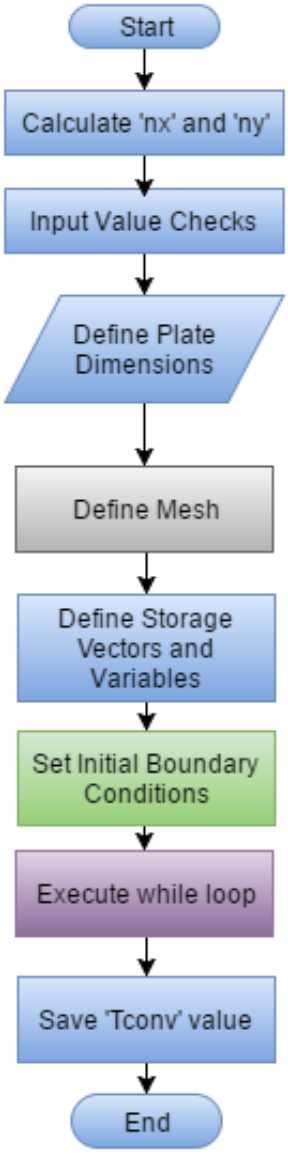

Figure 2: Coding Processes executed by HeatTransferProblem.m File

2.2.1 Define Meshgrid

The following text describes in detail the process shown in the flowchart in Figure 3. This section of the code sets up the meshgrid and defines the location of the circle on it. The 'meshgrid' function is used to produce a rectangular grid of 'X' and 'Y' coordinates using grid vectors obtained by inputting the length and width along with nx and ny values respectively to the 'linspace' function. Using the 'griddata' function, provided 'Xinit' is not currently empty, we can linearly interpolate the 'Xinit','Yinit' and 'T_init' values and evaluate these new points within the finer mesh. This is then outputed as T_old. To define

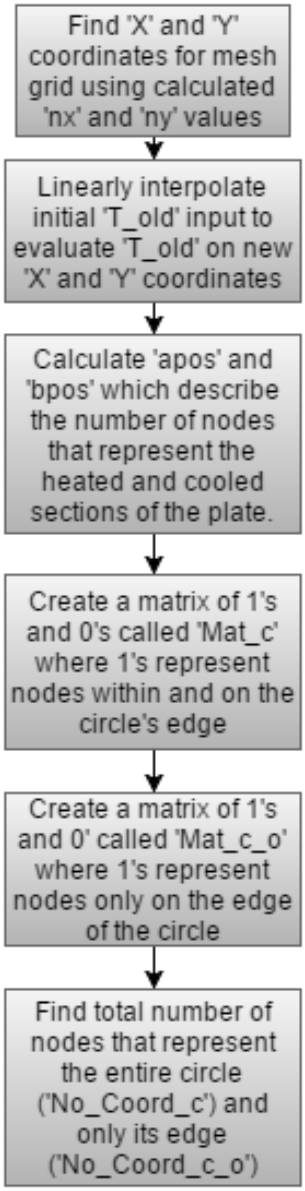

Figure 3: Calculation of Mesh Coordinates and Circle Location Points

the circle, it is first discritised and the coordinates of the edge are found. The 'inpolygon' function using these coordinates creates a polygon to represent the circle, and evaluates whether the coordinates 'X','Y' are in the polygon. It then returns a matrix, 'Mat_c', of 1's and 0's where 1's represent nodes within or on the edge of the circle. To locate nodes on the edge of the circle, a combination of for loops and if statements are used. If any point within 'Mat_c' is surrounded by a 0 either horizontally or vertically, this corresponding point in the 'Mat_c_o' matrix will be assigned a 1. All other points within 'Mat_c_o' are assigned 0's. Finally 'xrange' and 'yrange' are the indices used for the 'T_new' calculation within the while loop later on in the code.

2.2.2 Define Boundary Conditions

The defining boundary conditions section supported by Figure 4 forces the values on the edge of the plate and within the circle. The dirichlet boundary are the areas of the plate that are being heated and cooled. Using 'apos' and 'bpos' within the matrix indexing, we can define the nodes representing these sections of the plate. All other nodes on the edge of the boundary are insulated. This means the values for these will be identical to the next-inner-most node. These are set by indexing those locations for each side. Furthermore, by using matrix operations logically, we can set nodes on the plate within and on the edge of the circle to the average value of the nodes on its edge. These lines of codes are shown

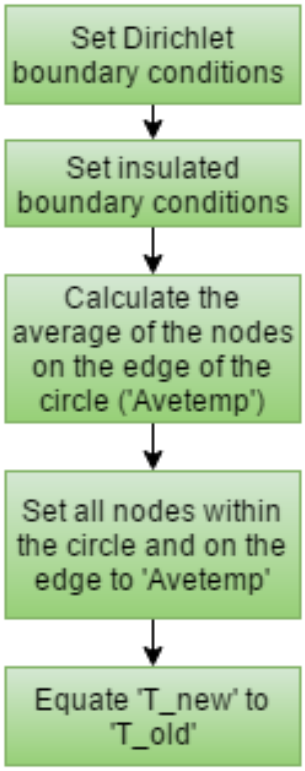

Figure 4: Setting the different Boundary Conditions

below:

```
Avetemp=(sum(sum((T_new).*(Mat_c_o))))/No_Coord_c_o;
T_new=(T_new.*(~Mat_c))+(Avetemp*Mat_c);
```

Both 'T_old' and 'T_new'are set to be equal with dirichlet boundary conditions enforced: the dirichlet boundary conditions are not set within the loop and it is ensured that these index locations are not changed within the loop. This improves efficiency as these conditions do not need to be set on each loop.

2.2.3 While loop

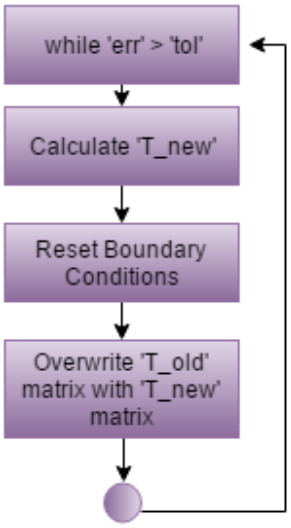

Figure 5: Coding Processes executed within While loop

Figure 5 describes the process within the while loop. The while loop is only set to function for when the error is greater than the tolerance. 'T_new' is calculated using a matrix approach instead of using a for loop. This improves the speed of execution and is described in more detail in the optimisation section. The boundary conditions are then coded as a separate function: these only reset the insulated boundary and circle nodes. Subsequently, the error is calculated also using a matrix approach by summing the difference between 'T_new' and 'T_old' for set indices. Lastly 'T_old' is overwritten by the 'T_new' matrix.

3 Results and Discussion

This section displays the final results for the default case provided in the specification. It provides proof of convergence as well as proof of the boundary conditions being set.

3.1 Simulation Output

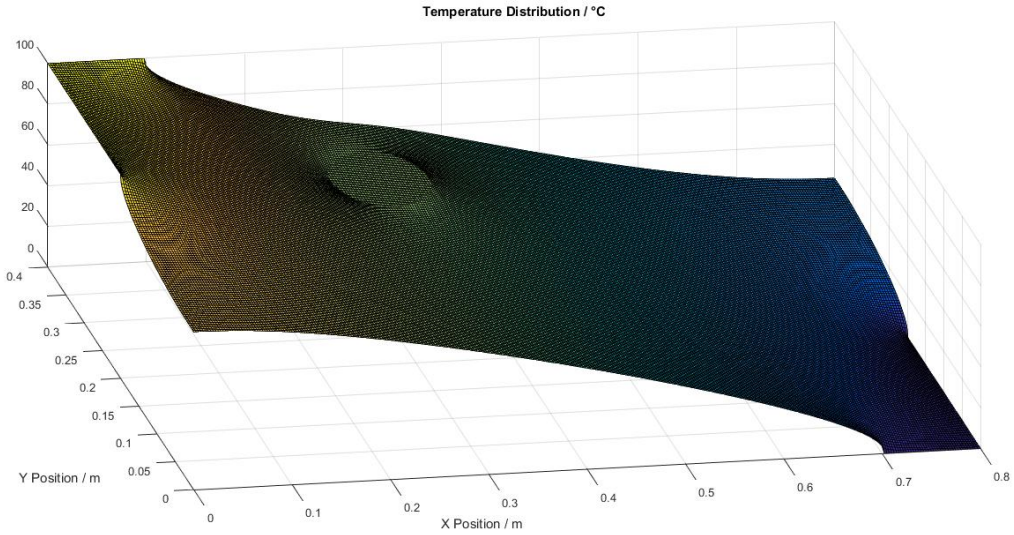

Figure 6: Graph showing the converged temperature distribution T(x, y)

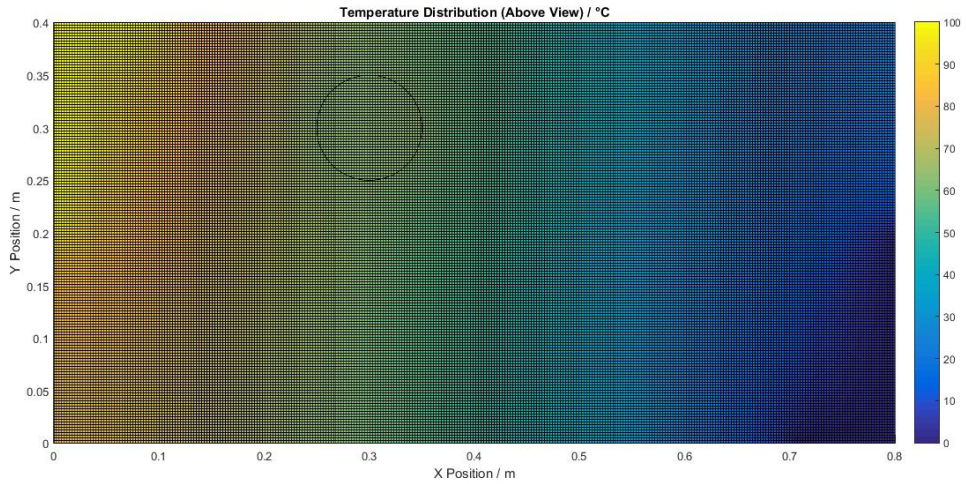

Figure 7: Graph showing the converged temperature distribution T(x, y) (Above view)

Figures 6 and 7 show the steady-state temperature distribution across the plate. Figure 6 shows this in the form of a contour. As we can see the top left section of the plate and the bottom right are the areas of the plate that are being heated and cooled respectively. The diagram, through the axes, shows proof that these sections are being maintained at 100K (Top-Left) and 0K (Bottom-Right). Furthermore, we can see the flat surface that forms the circular hole. The significance of a flat surface is that it represents an area in which which all the temperatures are equal.

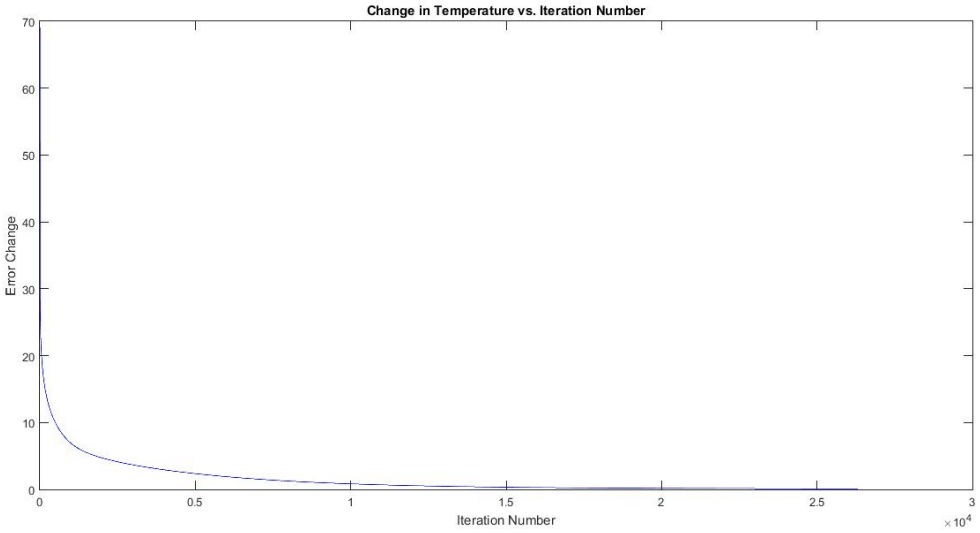

Figure 8: Graph showing the convergence plot of T vs. iteration count

3.2 Optimisation

Optimising the performance of a code is a vital skill within programming. For this particular code, there were several areas within which improvements could be made. The table below describes different aspects of the code that were altered within the optimisation phase. The most important column showing the improvement in times for each of these sections.

Table 1: Nonlinear Model Results

Section Description	Section Line No(s).	Previous Time (s)	Current Time (s)	Improvement in Time (s)
Overwrite T_old with T_new	107	224.09	0.15	223.94
Error Calculation	103-104	1000+	6.36	1000+
T_new calculation	97	1000+	11.73	1000+
Introducing interpolation	35-37	392.32	35.65	356.67

The following text explains the rationale behind the modifications described in the table. It is written in chronological order of which aspect of the code was considered first.

Removing for loops and replacing them with matrix operations: Using loops in any code can dramatically reduce its performance. Hence, this was the first aspect of the code that was attempted to be reduced. Within the code, there were initially four for loops. The first of which that was replaced is shown below.

```
for ix=1:nx
        for jy=1:ny
            T_old(ix,jy) = T_new(ix,jy);
        end
end
```

This part of the code selects each element of the T_old matrix individually and replaces it with the corresponding value within the T_new matrix. As these are both matrices, a more efficient way to do this process would be to replace one with the other completely without selecting individual elements. The line of code to do this is shown below.

```
T_old = T_new;
```

The next section of code that was condensed was the calculation of the error. The original is shown below:

```
for ix=2:nx-1
        for jy = 2:ny-1
            err=err+abs(T_new(ix,jy)-T_old(ix,jy));
        end
end
```

This for loop is used to calculate the absolute error of the plates temperature values. The difference between T_new and T_old for each corresponding node is calculated and these are all added up . A faster way to do this is to find the difference between the T_old and T_new matrices and sum the value of all the elements within. This line of code is shown below:

```
err = sum(sum(abs(T_new(2:end-1, 2:end-1)-T_old(2:end-1, 2:end-1))));
```

Similarly for the T_new calculation, the same format of for loop was initially used.

```
for ix=2:nx-1
        for jy=2:ny-1
T_new(ix,jy)=0.25*(T_old(ix+1,jy)+T_old(ix-1,jy)+T_old(ix,jy+1)+T_old(ix,jy-1));
        end
end
```

This was also converted to a matrix array operation described below. This section of code takes the longest to run through

```
T_new(xrange, yrange) = 0.25*(T_old(xrange+1, yrange) + T_old(xrange-1, yrange) + ...
 T_old(xrange, yrange+1) + T_old(xrange, yrange-1));
```

Interpolation: The modification that had the most significant effect on the speed of execution was the introduction of the interpolation of the meshgrid. By solving for lower resolutions and then interpolating, we were able to reduce the gap between the initial guess and the final answer at each step.

the gap between your starting guess and the final answer plays a big role in the time taken, so how would you improve your starting guess? You could start by solving a simpler relevant problem, and using the answer to this simpler problem as a starting guess. Instead of starting with a 80 40 grid, you could start with a 10 5 grid, a much larger x, y, and a much shorter time to converge on this approximate problem. Then copy the solution onto a finer mesh, maybe 20 10, solve this, then repeat on 4020, then finally on an 8040. This method is a form of multigrid, so-named for obvious reasons, and it one of the fastest and most flexible ways of solving Poisson-equation problems. The copying process from a coarse grid to a finer one takes a little care to get the indexing correct, and only every second value in each direction will have a directly supplied value. You will need to do a simple linear interpolation to fill in the gaps.

The final run time of the code is: 34.23 seconds

3.3 Aesthetics and User Friendliness

As per the requirement, parts of the code have been modified to make them more user friendly and have better aesthetics. These changes will be described in the following:

The software interacts with the user through a series of questions. It initally asks whether to run the simulation with default values, prompting a "Y/N" answer. The input is not caps sensitive. If 'n' is input, it prompts the user to enter each of the main function's inputs; if the user wants to use the default value for any given input, he/she may just press enter, i.e. not provide any input. Furthermore, numerical checks have been placed to allow the code to run. For example, if the value of 'dx' (the mesh discretization length) is not exactly divisible by the length or width of the plate, the next closest values is chosen to ensure 'nx' is an integer. A message is output to notify the user of this change. Another example is that if the centre of the circle is not within the required range, the code will not run and a message is displayed telling the user the cause of the error.

To improve the aesthetics, the code has been divided into three separate m files. This is to avoid cluttering of code on one file. The method used to reduce the number of lines in the code is by defining new local functions. Using local functions, large areas of the code that are repeated, such as the setting boundary conditions sections, can be reduced to a single line. Although this makes the code more aesthetically pleasing, it does decrease the speed of execution. However, the improvement in how legibile the code is worth the small reduction in performance. A final minor improvement is to include sections within the code (using the double percentage sign) to make the code easier to follow.

3.4 Improvements

Although the code provides the user with the required answer fairly efficiently, there are several improvements that still could be made. From the optimisation section we can see that the majority of the time is taken within the while loop whilst completing the matrix calculations. Ways to improve these could be to implement the 'sparse' function which reduces matrices by removing any zero elements it may contain. This reduces the time it takes to multiply matrices together. The 'full' function could then be used at the end to bring the matrix back to its original dimensions.

4 Conclusion

In conclusion, we can see that the final output of the code provides the correct answer for the problem specified. Through the plots provided we can see the steady-state temperature distribution across the plate as well as the evidence of convergence occurring.

5 Bibliography

References

6 Appendix

6.1 Main.m

```
clear all
close all

%% Define User Input

UserInput %Run User Input Code


%% Run Simulation

%Create dx and dy steps
dxstep = linspace(dx*optsteps, dx, optsteps);
dystep = dxstep;

T_init = [];
Xinit = [];
Yinit = [];

for i = 1:optsteps
    [Tconv, X, Y, xc, yc, errsave] = HeatTransferProblem(dxstep(i),dystep(i), tol, xH,yH, T_Heat, T_Cool, T_init, Xinit, Yinit, optsteps, i);
    T_init = Tconv;
    Xinit = X;
    Yinit = Y;
end


%% Plot Result

%Surface Contour Plot
P = figure;
Fig1 = surf(X,Y,flipud(Tconv)); %flipud flips the matrix across the horizontal: needs to be done due to the way the matrix is defined.
xlabel('X Position / m');
ylabel('Y Position / m');
title('Temperature Distribution / K');

%Surface Contour Plot (Above View)
C = figure;
Fig1 = surf(X,Y,flipud(Tconv)); %flipud flips the matrix across the horizontal: needs to be done due to the way the matrix is defined.
view(2); %See from above
colorbar %Put in ColorBar
axis equal %Resize so units are the same for both axis
axis([0, 0.8, 0, 0.4]) %Change axis limits
xlabel('X Position / m');
ylabel('Y Position / m');
title('Temperature Distribution (Above View) / K');
zmax = max(max(Tconv));
line(xc, yc, zmax*ones(size(xc,2),1), 'Color', 'k');

%Plot Error Convergence
C = figure;
plot(1:size(errsave,2), errsave, '-b');
```

```
52 title('Change in Temperature vs. Iteration Number');
53 xlabel('Iteration Number');
54 ylabel('Error Change');
```

6.2 HeatTransferProblem.m

```
function [Tconv, X, Y, xc, yc, errsave] = HeatTransferProblem(dx,dy,tol,xH,
    yH,T_Heat, T_Cool, T_old, Xinit, Yinit, optsteps, currentloop)
%% Define Variables
length = 0.8;
width = 0.4;

nx=ceil(length/dx);
ny=ceil(width/dy);

%% Input Value Checks
if xH < 0.05 || xH > 0.75
    if yH  < 0.05 || yH > 0.35
        disp('The coordinate values for the centre of the circle are not
            within the required range')
    end
end

if currentloop==optsteps
    if ceil(length/dx)~=length/dx || ceil(width/dy)~=width/dy
        disp('The dx or dy values provided are not exactly divisible by the
            lengths or width of the plate.');
    end
end

%% Plate Dimensions
length = 0.8;
width = 0.4;
a=0.1; % Horizontal length of plate being heated or cooled
b = 0.2; % Vertical length of plate being heated or cooled
R = 0.05; %Radius of Hole

%% Define Mesh

%Calculate Location of Mesh Grid Points
[X,Y] = meshgrid(linspace(0, length, nx) , linspace(0, width, ny));

%Define T_old by interpolating between old values and new values
if ~isempty(Xinit)
    T_old = griddata(Xinit, Yinit, T_old, X, Y);
end

%Calculate Number of Nodes to represent Boundary Conditions
apos=ceil((a/length)*nx); % Number of Mesh Nodes for Horizontal Length of
    plate being Heated or Cooled
bpos=ceil((b/width)*ny); % Number of Mesh Nodes for Vertical Length of
    plate being Heated or Cooled

%Find Nodes Representing the Circle
deg = pi/180 * (0:0.1:360); %Discretize Circle Angles
xc = xH + R*cos(deg);
yc = yH + R*sin(deg);

Mat_c = flipud(inpolygon(X, Y, xc, yc)); % Uses xc and yc to create a
    polygon to represent a circle, and evaluates whether the coordinates X,Y
     are in the polygon. Returns a matrix of 0s or 1s based on if the
    coordinate is in the polygon.
```

```
49
50 % Find the coordinates on the edge of the circle
51 Mat_c_o=zeros(ny,nx);
52 for i=1:nx;
53     for j=1:ny;
54         if Mat_c(j,i)==1;
55             if Mat_c(j,i+1)==0 || Mat_c(j,i-1)==0 || Mat_c(j+1,i)==0 ||
                   Mat_c(j-1,i)==0;
56                 Mat_c_o(j,i) = 1;
57             end
58         end
59     end
60 end
61
62 No_Coord_c_o = sum(sum(Mat_c_o)); %Number of Coordinates on the edge of the
       circle
63 No_Coord_c = sum(sum(Mat_c)); %Number of Coordinates in Circle
64
65 xrange = 2:ny-1;
66 yrange = 2:nx-1;
67
68 %% Define Storage Vectors and Variables
69
70 %Only for the first Optimization Loop
71 if isempty(T_old)
72     T_old = zeros(ny, nx);
73 end
74
75 err=tol+1;
76 errsave = zeros(1,10000); %Create vector for saving Delta T, set length so
       vector length doesn't change every loop
77 count = 0; %Counter for Iteration Number
78
79 %% Set Initial Boundary Conditions
80 % Initial boundary conditions for rectangular plate
81 %Dirichlet Boundary Conditons
82 T_old((1:bpos),1) = T_Heat; %Left Side
83 T_old(1,(1:apos)) = T_Heat; %Top Side
84 T_old((ny-bpos+1):ny,nx) = T_Cool; %Right Side
85 T_old(ny,nx-apos+1:nx) = T_Cool; %Bottom Side
86
87 T_old = SetBoundaryConditions(T_old, bpos, apos, ny, nx, Mat_c_o,
       No_Coord_c_o, Mat_c); %Set Insulation and Circle Boundary Conditions
       Just in Case
88
89 T_new = T_old; %Done because the Dirichelet Boundary Conditions only need
       to be set once
90
91 %% Set Up Loop
92 while err >tol
93
94     count = count + 1; %Increment Loop Counter
95
96     %For every position other than the boundary, calculate T_new
97     T_new(xrange, yrange) = 0.25*(T_old(xrange+1, yrange) + T_old(xrange-1,
           yrange) + T_old(xrange, yrange+1) + T_old(xrange, yrange-1));
98
99     % Set Boundary Conditions
```

```
100       T_new = SetBoundaryConditions(T_new, bpos, apos,ny,nx,Mat_c_o,
              No_Coord_c_o,Mat_c);
101
102       %Calculate error by finding difference between T_new and T_old
103
104       err = sum(sum(abs(T_new(2:end-1, 2:end-1)-T_old(2:end-1, 2:end-1))));
105       errsave(count) = err; %Save the error
106
107       %Swap old and new for next iteration after calculating error
108       T_old = T_new;
109
110   end
111
112   %Reduce Errsave Vector to Used Values
113   errsave = errsave(1:count);
114
115   Tconv = T_old; %Save final value of T_new to Output
116   end
117
118   %% Local Function: Boundary Conditions
119
120   function T_new = SetBoundaryConditions(T_new, bpos, apos, ny, nx, Mat_c_o,
          No_Coord_c_o, Mat_c)
121
122   %Insulated Boundary Conditions
123       T_new((bpos+1:ny),1) = T_new((bpos+1:ny),2); %Left Side
124       T_new(1,(apos+1:nx))= T_new(2,(apos+1:nx)); %Top Side
125       T_new(1:ny-bpos,nx) = T_new(1:ny-bpos,nx-1);    %Right Side
126       T_new(ny,1:nx-apos) = T_new(ny-1,1:nx-apos);%Bottom Side
127
128   %Boundary conditions for the circle
129       Avetemp=(sum(sum((T_new).*(Mat_c_o))))/No_Coord_c_o; %Find the average
              temperature of the edge values by summing and using matrix
              operations
130       T_new=(T_new.*(~Mat_c))+(Avetemp*Mat_c); %Use matrix operations to
              input this average temperature value into phinew
131   end
```

6.3 UserInput.m

```
%% Define Inputs

Default = input('Would you like to run the simulation using default values (Y/N) : ', 's');

%Set Defaults Inputs
DefaultInputs = [0.0025, 100, 0, 0.1, 0.3, 0.3, 4]; % [dx, T_Heat, T_Cool, tol, xH, yH, optsteps]
Inputs = zeros(1, 7);

if isequal(Default, 'Y') || isequal(Default, 'y') %If Yes, then set inputs to Default
    Inputs = DefaultInputs;

elseif isequal(Default, 'N') || isequal(Default, 'n')

    UserInputs1 = input('Please enter delta x: ');
    UserInputs2 = input('Please enter the boundary heating conditions: ');
    UserInputs3 = input('Please enter the boundary cooling conditions: ');
    UserInputs4 = input('Please enter the temperature convergence tolerance criterion: ');
    UserInputs5 = input('Please enter the X Position of the Centre of Hole within the provided range (0.05 to 0.75): ');
    UserInputs6 = input('Please enter the Y Position of the Centre of Hole within the provided range (0.05 to 0.35): ');
    UserInputs7 = input('Please enter the Number of Grid Resolution steps to be taken : ');

    Inputs = DefaultInputs; % Set Inputs to Default and then change input values if User has input

    %For not empty, change the value to the input value
    if ~isempty(UserInputs1)
        Inputs(1) = UserInputs1;
    end
    if ~isempty(UserInputs2)
        Inputs(2) = UserInputs2;
    end
    if ~isempty(UserInputs3)
        Inputs(3) = UserInputs3;
    end
    if ~isempty(UserInputs4)
        Inputs(4) = UserInputs4;
    end
    if ~isempty(UserInputs5)
        Inputs(5) = UserInputs5;
    end
    if ~isempty(UserInputs6)
        Inputs(6) = UserInputs6;
    end
    if ~isempty(UserInputs7)
        Inputs(7) = UserInputs7;
    end

elseif isempty(Default)
    messagebox('Simulation was terminated');
```

```
49      return;
50  end
51
52  %Unpack Inputs Matrix into relevant input variables
53  dx = Inputs(1);
54  T_Heat = Inputs(2);
55  T_Cool = Inputs(3);
56  tol = Inputs(4);
57  xH = Inputs(5);
58  yH = Inputs(6);
59  optsteps = Inputs(7);
```