

Computational Methods

2D DIFFUSION EQUATION MODEL USING THE EXPLICIT
FINITE DIFFERENCE METHOD

Table of Contents

1. Introduction	3
2. The Equation	4
2.1 One dimensional Diffusion	4
2.2 Two-Dimensional Diffusion	4
3. Numerical Method	5
3.1 Taylor series	5
3.2 Directional differences	6
3.3 Explicit Finite Difference scheme	7
3.4 Stability criterion	8
3.5 Grid spacing	8
3.6 Time Steps	8
4. The initial, boundary conditions and analytical solution	9
5. Codes details and explanations	10
6. Code Validation	15
6.1 Importance	15
6.1.1 Accurate Results:	15
6.1.2 Efficiency:	15
6.1.3 Maintainability:	15
6.1.4 Duplicability:	15
6.1.5 Risk Management:	15
6.1.6 Confidence:	15
6.2 Methods	15
6.2.1 Manual Testing:	15
6.2.2 Automated Testing:	15
6.2.3 Debugging tools	15
6.2.4 Code Analyser	16
6.2.5 Documentation	16
7. Analysis of the code results	17
7.1 Change in Step	17
7.2 Change in Tolerance	18
7.3 Change in Grid Spacing	18
8. Limitations	20
8.1 Taylor Series	20
8.1.1 Local Approximation	20
8.1.2 Computational Complexity	20

8.1.3 Analytical Derivatives.....	20
8.1.4 Handling Cutoffs.....	20
8.2 EFD.....	20
8.2.1 Stability Issues.....	20
8.2.2 Accuracy.....	20
8.2.3 Accumulation of Errors	20
8.2.4 Boundary Condition Handling	21
8.2.5 Non-linear Equations	21
8.2.6 Dimensions.....	21
9. Conclusion	22
10. References	23
11. Appendix (MATLAB Script).....	24

1. Introduction

The aim of this report is to use a form of coding in MATLAB in order to model and calculate the temperature distribution through an aluminium plate over a set amount of time, until thermal equilibrium. I will aim to monitor the temperature changes in the domain (centre) of the plate. The model is to keep registering temperature changes until thermal equilibrium has been reached. I will have created a grid (mesh) structure on the plate model, creating nodes. This will help to define the different location temperatures. The coding will need to provide a user input to define the mesh size/number of nodes. During this report I will create an equation to measure the temperature.

This report topics including:

- Reviewing the general basis of the heat equation.
- Exploring the numerical method of calculating the temperature distribution.
- Analysis of the code and the results it produces.
- Discuss the evaluation of this modelling method.

The Aluminium plate being tested (figure 1) has the following material values:

Length: $L=6\text{m}$
Width: $W=6\text{m}$
Conductivity: $K=143\text{ W/c.C}$
Density: $D=2.8 \times 10^3\text{ Kg/m}^3$
Specific Heat: $C=795\text{ J/Kg.C}$

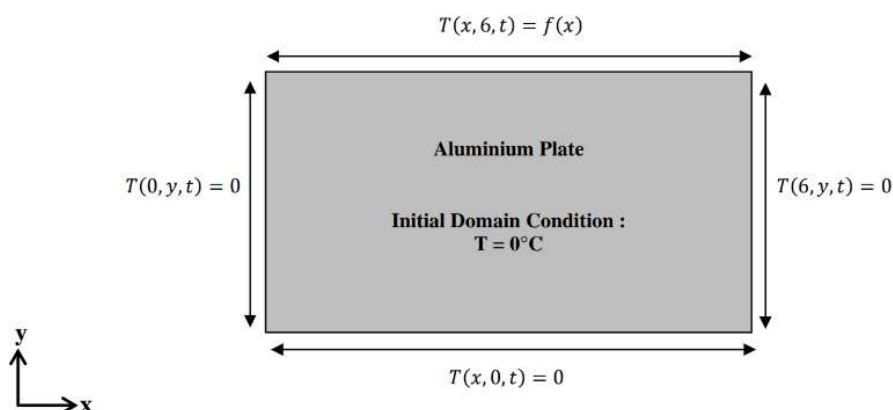


Fig. 1

I have created a code that will calculate the temperature distribution when heat is applied on the top edge of the material until thermal equilibrium has been reached (Appendix 1). The output of this code will produce distribution graphs and result figures. From these outputs I will then evaluate how the user inputs, Grid Spacing and Time Steps, can affect the outcomes of the code.

There are a few assumptions that have been made during this report:

- Thickness of the aluminium is negligible.
- Initial temperature of the material is 0°C
- Temperature of the edges will remain constant.
- Heat source on the top edge will always be 50°C
- Cooling device will be located on the other three edges.

2. The Equation

To create a code to determine the heat distribution of the aluminium plate, the governing equation should first be understood. The equation required to calculate the thermal diffusion can be referred to as 'Fourier's Law'.

Fourier's Law explains how heat moves through a material. According to the law, the temperature differential across a material directly affects the rate of heat transfer through it, which is also controlled by the material's thermal conductivity. (Byjus.com, 2023)

2.1 One dimensional Diffusion

The thermal diffusion equation for one dimensional forms of diffusion can be expressed in the following format:

$$\frac{\partial T}{\partial t} = \alpha \frac{\partial^2 T}{\partial x^2}$$

With the values equate to:

T = the Temperature

t = the Time

x = Spatial coordinate

$\frac{\partial T}{\partial t}$ = the rate of change of temperature in respect of time

$\frac{\partial^2 T}{\partial x^2}$ = the second spatial derivative of temperature in respect of spatial coordinates.

α = the Alpha, is the thermal diffusivity of the material. α can be defined as $\frac{k}{\rho c_p}$ where k is thermal conductivity ρ is the density, and c_p is the specific heat.

2.2 Two-Dimensional Diffusion

However, as the requirement of this report is to calculate over a two-dimensional surface, the equation will need to account for the second dimension. To achieve this, a second set of spatial derivative needs to be added to the calculation. This is expressed in the following format:

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

(Physlab.org, 2016)

The addition of the $\frac{\partial^2 T}{\partial y^2}$ to the equation means that we can work out the diffusion in both the x and now the y plains.

When the alpha is defined in the sum of its parts, the formula can then be written in the following:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{\rho c_p}{k} * \frac{\partial T}{\partial t}$$

The temperature figure for each of the temperature nodes in the grid on the aluminium plate come in the form of $T(x,y,t)$, as shown on the material edge temperature values in figure 1. When the code is run, each temperature node at each passing time step will be referenced in this format.

3. Numerical Method

3.1 Taylor series

The Taylor series is a technique used to illustrate a mathematical function when infinite mathematical terms are added together. Brook Taylor was a mathematician who introduced the idea in the early 1700s. In calculus and mathematical analysis, the Taylor series is a helpful tool when it comes to approximating functions. (*Maplesoft.com, 2023*). Calculus and mathematical analysis both use the Taylor series method.

The Taylor series in terms of function $f(x)$ when talking about a point, such as $x = x_0$, can be viewed as such:

$$f(x) = f(x_0) + f^{(1)}(x_0) \frac{(x - x_0)^1}{1!} + f^{(2)}(x_0) \frac{(x - x_0)^2}{2!} + f^{(3)}(x_0) \frac{(x - x_0)^3}{3!} + \dots$$

where:

$f(x_0)$ = Function at point $x = x_0$

$f^{(1)}(x_0)$ = 1st derivative at $x = x_0$

$f^{(2)}(x_0)$ = 2nd derivative at $x = x_0$

$f^{(3)}(x_0)$ = 3rd derivative at $x = x_0$ (*testbook.com, 2023*)

The Taylor series can go on infinitely, depending on the number of terms required. However, this formula can be written in a compact form, such as:

$$f(x) = \sum_{n=0}^{\infty} f^{(n)}(x_0) * \frac{(x - x_0)^n}{n!}$$

With n representing the number of derivatives in the equation and $n!$ Referring to the factorial of n .

$f^{(n)}(x_0)$ refers to the derivative of f evaluated to the point of x_0

Having this compact formula will greatly reduce the time it takes to input the data request into a system. Rather than inputting every derivative manually, such as the original equation, you can input a singular line in a code.

3.2 Directional differences

The derivative of a function at a point can be calculated by using different numerical methods. These are the Forward, backward and central differences.

The forward difference is determined by calculating the slope of the secant line that passes between the coordinates (x_0) and $(x_0 + h)$ where h is a step size (m).

The backward difference acts similarly to the forward, just with negative steps. For example, the secant line would pass through the coordinates (x_0) and $(x_0 - h)$ where h is a step size (m).

The central difference uses both previous differences with the aim to gain a more accurate approximate of the derivative. So, this would be based off the function at $(x_0 + h)$ and $(x_0 - h)$.

To work out the forward difference, you will need to complete the following steps:

$$f(x) = f(x_0) + f^{(1)}(x_0) \frac{(x - x_0)^1}{1!} + f^{(2)}(x_0) \frac{(x - x_0)^2}{2!} + f^{(3)}(x_0) \frac{(x - x_0)^3}{3!} + \dots$$

↓ Swap all $x - x_0$ values for h

$$f(x_0 + h) = f(x_0) + f^{(1)}(x_0) \frac{h^1}{1!} + f^{(2)}(x_0) \frac{h^2}{2!} + f^{(3)}(x_0) \frac{h^3}{3!} + \dots$$

↓ right side is divided by h value.

$$f(x_0 + h) - f(x_0) = \frac{f^{(1)}(x_0) * h^1 + f^{(2)}(x_0) \frac{h^2}{2!} + f^{(3)}(x_0) \frac{h^3}{3!}}{h}$$

↓ take h value over.

$$\frac{f(x_0 + h) - f(x_0)}{h} = f^{(1)}(x_0) * h^1 + f^{(2)}(x_0) \frac{h^2}{2!} + f^{(3)}(x_0) \frac{h^3}{3!}$$

↓ Make $f^{(1)}(x_0)$ the subject.

$$f^{(1)}(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}$$

The backwards difference has the same process; however, the result will have a slight difference in that the positive h value will now be negative. You will end up with the following:

$$f^{(1)}(x_0) \approx \frac{f(x_0 - h) - f(x_0)}{h}$$

The central difference will instead have both the positive and negative values in its final equation. The h value will also now be $2h$ to account for both directions.

$$f^{(1)}(x_0) \approx \frac{f(x_0 + h) - f(x_0 - h)}{2h}$$

University of Manitoba, 2023)

3.3 Explicit Finite Difference scheme

Explicit finite difference schemes can be used for approximating the solution of partial differential equations (PDEs). This method is especially useful when dealing with time-dependent issues. Problems where the temperature distribution changes over an area during a length of time, such as heat conduction, are often addressed using it.

The formula given in this assignment, and as mentioned previously, was the following equation:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{1}{\alpha} * \frac{\partial T}{\partial t}$$

To work out the second derivative approximations, the central difference method can be used. Where the x axis has been swapped out for the East and West directions and the y axis have been swapped out for the North and South. This results in the following equations:

x axis at point P

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_E^t - 2.T_p^t + T_W^t}{h^2}$$

y axis at point P

$$\frac{\partial^2 T}{\partial y^2} = \frac{T_N^t - 2.T_p^t + T_S^t}{h^2}$$

The 1st derivative would therefore become:

$$\frac{\partial T}{\partial t} = \frac{T_p^{t+\Delta t} - T_p^t}{\Delta t}$$

With these equations solved, they can be substituted into the thermal diffusion equation to give:

$$T_p^{t+\Delta t} = Fo (T_E^t + T_W^t + T_N^t + T_S^t - \left(4 - \frac{1}{Fo}\right) \cdot T_p^t)$$

(A.Marotin, 2023)

Δt refers to the time step.

The **Fo** in the equation relates to the Fourier number. This can be represented in the equation format:

$$Fo = \frac{\alpha \cdot \Delta t}{h^2}$$

According to Thermopedia.com (2011), In heat transfer problems, the Fourier number (Fo) is a dimensionless number that is commonly used in the context of transient (time-dependent) heat conduction. It describes how a material's rate of thermal energy storage and rate of heat conduction are compared.

The Fourier number can help to indicate the rates of heat transfer a material is experiencing. For instance: a low number would indicate that there is a slow rate of heat transfer. Equally, a high number would indicate a speedier heat transfer. (Mechcontent.com,2020)

3.4 Stability criterion

Stability criteria are conditions that must be met in numerical methods, this helps to ensure that the results remain within the confines of the boundary limitations; thus, preventing any numerical instability.

An issue that can occur is that large roundoff errors can happen due to the numerical solution of differential equations. For this reason, the Fourier number is used due to it being a dimensionless quantity. For this to work, it is important that the Fourier number is calculated to ensure the stability of the equation. If the value of this number is outside of this quantity range the code will fail to produce accurate results.

Considering the grid spacing and the time step, the suitable value for the Fourier number to ensure stability for this code is as below:

$$0 < F_0 \leq \frac{1}{4} \quad \text{in } 2D$$

3.5 Grid spacing

The spatial domain of the aluminium plate will be split evenly across the surface area. The grid spacing should also be made small enough to get an accurate reading, but large enough that the amount of grid points is reasonable. If there are too few locations, the results that are returned will not provide enough data to be considered accurate enough to properly portray the thermal diffusion. If there are too many locations, although an accurate representation of the thermal diffusion will be produced, the amount of time and processing power it would take the MATLAB equipment to run it would be unreasonable.

As the length of the material is 6m x 6m. The number of nodes travelling in the x and y directions should therefore be spread out in locations (in meters) divisible by 6. For example, grid spaces of 1m would allow for 6 nodes to travel in the x and y dimensions, provided a total of 36 nodes.

3.6 Time Steps

The temporal domain for the code has a similar situation as the grid spacing. Too small of a time step and you will get a highly accurate result, however, it could take unreasonably long to allow the software to run. Too large of a time step, and time jumps could miss important stages of the heat transfer in the model. This would provide an inaccurate set of results. Clearly, there is a balance between the quality of the results and how reasonable of a wait time there should be to receive the data. Software and Hardware limitations should also be taken into consideration as the equipment may not have the processing power to safely run the code.

For example, if the software dictated that a set of parameters would take up to a day for equilibrium to be reached in the model: too low of a time step could mean that the software could take several hours to run a single test, whereas a reasonable number could have informed you of this in the space of a few minutes.

4. The initial, boundary conditions and analytical solution

The initial boundaries set for this analysis activity include the following:

- Dimensions are set for a 6m x 6m area.
- Nodes to be analysed are to fit within equal spaces within the 6m x 6m space.
- All edges are to be set at 0°C

Constant boundary conditions that come after

- Temperature on the West, East and South ends are to be set at 0°C.
- The North edge is to be consistently at a temperature value of 50°C.
- A Dirichlet boundary condition shown in Figure 2 is set. This describes the temperature variables' values at a certain border point. In this example; the north boundary $F(x)$ has the below rules put into place on different sections of the edge. If a node is more than 2/3rds from zero, the $F(x)$ value becomes $-25x + 150$. Less than 2/3rds it becomes $12.5x$.

$$f(x) = \begin{cases} 12.5x & \text{if } 0 \leq x \leq \left(\frac{2}{3} \times 6\right) \\ -25x + 150 & \text{if } \left(\frac{2}{3} \times 6\right) < x \leq 6 \end{cases}$$

5. Codes details and explanations

Below is a series of screenshots of the code created to run the program that will simulate the thermal diffusion of the aluminium plate.

At the start of the script, lines 2-4 are used to clear the command window of all existing programmes running. This ensures that the program is focused only on the current code and the operation window only shows the relevant information to the user.



```
1 %Clearing Command
2 clc
3 clear
4 close all
```

Fig. 3 Code, lines 1-4

Lines 7- 10 are a list of the properties of the aluminium plate. This information acts as a reference hub for the rest of the code.



```
6 % Aluminium Properties
7 K=143;
8 D=2.8*10^3;
9 C=795;
10 alpha=K/(D*C);
```

%Conductivity
%Density
%Specific heat
%Alpha calculated using above values

Fig. 4 Code, lines 5-1

Like the previously mentioned lines; lines 13 and 14 are used as reference data that give the dimensions of the plate.



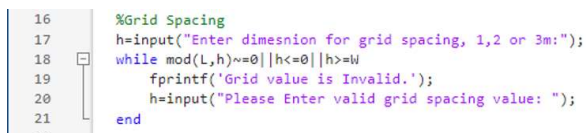
```
12 %Dimension
13 L= 6;
14 W= 6;
```

%X Variable
%Y Variable

Fig. 5 Code, lines 12-14

Line 17 will bring up a prompt to the user, asking them to input the grid spacing value.

The while code during Lines 18 to 21 will ensure that the input by the user is a figure divisible by, but not including, six. If the number is invalid, lines 19 &20 will produce the error message and request a new figure is entered. This will continue to loop each time an invalid figure is inputted until a valid value has been selected.



```
16 %Grid Spacing
17 h=input("Enter dimesnion for grid spacing, 1,2 or 3m:");
18 while mod(L,h)~=0 || h<=0 || h>=W
19     fprintf('Grid value is Invalid. ');
20     h=input("Please Enter valid grid spacing value: ");
21 end
```

Fig. 6 Code, lines 16-21

Line 24 will now prompt the user to input a time step value.

Line 25 tells the code what the appropriate Fourier value range should be.

While code between lines 26 and 28 will continue to loop each time an invalid figure is inputted until a valid value has been selected.

```
23 %Time Steps
24 dt=input("Enter the value of time steps in seconds: ");
25 Fo=(alpha*dt)/(h^2);
26 while Fo<=0||Fo>0.25
27     fprintf("Time step is invalid.");
28     dt=input("Please Enter valid time step value: ");
29 end
```

%User Input for time step value
% Fourier Number

Fig. 7 Code, lines 23-29

The lines in figure 8 define the rows and column values to fit within the matrix.

```
31 %Defining rows and columns
32 NR=W/h+1;
33 NC=L/h+1;
```

Fig. 8 Code, lines 31-33

Line 37 is used to zero out the initial grid spacing.

```
36 %Creating initial T1
37 T1=zeros(NR,NC);
```

Fig. 9 Code, lines 36-37

This section of the code sets the initial boundary conditions of the matrix. Line 41 sets the locations of the nodes depending on what the input the user original gave.

Lines 42 to 46 use an 'if' function to implement the values that were shown in figure 2. Lines 41& 42 indicate if a node is Less than 2/3rds from 'd' the $F(x)$ value becomes $12.5x$. Lines 44 and 45 indicate that if 'd' is more than 2/3rds from zero, the $F(x)$ value becomes $-25x + 150$.

```
39 % Amend Bounrary Conditions Of Initial Matrix
40 for c=1:NC
41     d=(c-1)*h;
42     if d<=2/3*L
43         T1(1,c)=12.5*d;
44     else
45         T1(1,c)= -25*d+150;
46     end
47 end
```

Fig. 10 Code, lines 39-47

This section of the code is used to create a second matrix from the first. Line 51 gives the tolerance value for the equilibrium, currently at a value of 0.001. Line 52 indicates the maximum value difference that can occur before an error occurs. Line 53 sets the initial count to zero.

<pre> 49 %Create T2 Matrix from T1 Matrix with temp calculations 50 T2=T1; 51 Tol=0.001; 52 Dif=Tol+1; 53 Count=0; 54 </pre>	<pre> % Tolerance for equilibrium % Initial condition for max value in difference matrix (Err) for Initial condi </pre>
--	---

Fig.11 Code, lines 49-53

This while loop between lines 56 and 62 is there to tell the program to keep running until thermal equilibrium has been reached. Line 60 is particularly important as it sets the Taylor series formula for the program to follow. Line 63 will help the program to determine once equilibrium has been reached.

<pre> 55 %While loop set up to repeat calculations until the difference between T2 and T1 (Err) is less than 0.01 and therefore close to equilibrium. 56 while Dif>Tol 57 T1=T2; 58 for G=2:NR-1 59 for H=2:NC-1 60 T2(G,H)=Fo*(T1(G-1,H)+T1(G,H-1)+T1(G,H+1)+T1(G+1,H))-(4-(1/Fo))*T1(G,H); 61 end 62 end 63 Dif=max(abs(T2-T1)); </pre>	<pre> %Condition to initiate/continue loop %Dimension to ignore perimeter nodes %Dimension to ignore perimeter nodes %Taylor series formula used to calculate T1 values at the next time step %Formula to determine if thermal equilibrium has been reached </pre>
--	--

Fig. 12 Code, lines 55-63

This section of code is used to tell the program to create and display a graph with the results. May of the commands below will indicate ways to personalise the graph. For instance; Line 68 indicates the colour scheme of the graph. Lines 69 to 71 are commands detailing the description and location of a colour bar that will go beside the graph. Lines 67, 72 and 73 are all commands involving labelling of the graph.

<pre> 65 figure(1) 66 contourf(flipud(T2)) 67 title ('Temperature Distribution of plate'); 68 colormap ('parula') 69 c=colorbar; 70 c.Location = 'southoutside'; 71 c.Label.String = 'Temperature'; 72 xlabel('X distance in meters'); 73 ylabel('Y distance in meters'); 74 pause(0.2) 75 Count=Count+1; 76 end </pre>	<pre> %Command to create graph %Title for Fig1 %Color system used for figure 1 %Attaches color bar to figure 1, representing temp %Pauses after each step to update graph to visually show the change along plate % Update the amount of cycles completed </pre>
---	---

Fig. 13 Code, lines 65-76

The next set of commands are used to calculate the duration it would take the aluminium plate to reach thermal equilibrium based upon the inputs made by the user. Depending on the scaling of time, the if commands will give the results in the most appropriate units of time. The options start from seconds and scale up to days. Line 97 then commands the code to display the result.

```

78 % Thermal equilibrium time
79 eq=Count*dt; %Calculate the time taken to reach equilibrium by multiplying the count by the time step
80
81 if eq<60 %Will display in seconds
82     fprintf('It took %g seconds to reach thermal equilibrium\n', eq) % Display amount time taken to reach equilibrium in seconds
83
84 elseif eq>=60 && eq<3600 %Will display in minutes
85     eqh=eq/60;
86     fprintf('It took %g minutes to reach thermal equilibrium\n', eqh) % Display amount time taken to reach equilibrium in minutes
87
88 elseif eq>=3600 && eq<86400 %Will display in hours
89     eqh=(eq/60)/60;
90     fprintf('It took %g hours to reach thermal equilibrium\n', eqh) % Display amount time taken to reach equilibrium in hours
91
92 elseif eq>=86400 %Will display in days
93     eqd=((eq/60)/60)/24;
94     fprintf('It took %g days to reach thermal equilibrium\n', eqd) % Display amount time taken to reach equilibrium in days
95 end
96
97 fprintf('%g seconds\n', eq) %Time in seconds for analysis
98

```

Fig. 14 Code, lines 78-97

Lines 101 to 110 gives the parameters for the analytical solution portion of the test. Line 102 zero's out the starting values.

```

99 % Analytical Solution Calculation
100 % Calculate the analytical solution
101 num_terms = 100; % Number of terms in the series to include
102 T_analytic = zeros(NR, NC);
103 for G = 1:NR
104     for H = 1:NC
105         x = (G - 1) * h;
106         y = (H - 1) * h;
107         T_analytic(G, H) = calculate_analytical_temperature(x, y, num_terms);
108     end
109 end
110 T_analytic = rot90(T_analytic); % Rotate matrix 90 degrees counterclockwise for proper orientation
111

```

Fig. 15 Code, lines 99-110

These next two lines command the software to display the results of the analytical calculations.

```

112 % Display the analytical solution in the command window
113 disp('Analytical Temperature Distribution at Thermal Equilibrium:');
114 disp(T_analytic);
115

```

Fig. 16 Code, lines 112-114

Line 117 gets the program to calculate the difference between the results of the first program and the analytical calculations. Line 118 and 119 command the program to display these results.

```

116 % Calculate the difference between analytical and numerical solutions
117 difference = T1 - T_analytic; % Compute the difference between solutions
118 disp('Difference Between Numerical and Analytical solutions:');
119 disp(difference);
120

```

Fig. 17 Code, lines 116-119

The set of commands in figure 18 are duplicated of those during lines 65 to 76 with the exception of line 123 which indicates the graph be for the analytic results.

```

121 % Visualization of the analytical solution (Figure 2)
122 figure(2)
123 contourf(flip(T_analytic)) % Rotate by 90 degrees (clockwise) and then flip vertically
124 colormap("parula")
125 colorbar % Keep the colorbar on the right side
126 c=colorbar; %Attaches color bar to figure 1, representing temp
127 c.Location = 'southoutside';
128 c.Label.String = 'Temperature';
129 clim([0 50]); % Set colorbar limits (Replace 'min_value' and 'max_value' with your desired range)
130 xlabel('X distance in meters');
131 ylabel('Y distance in meters'); % Adjust the label accordingly
132 title('Analytical Temperature Distribution');

```

Fig. 18 Code, lines 121-132

The set of commands in figure 18 are duplicated of those during lines 65 to 76 with the exception of line 136 which indicates the graph be for the difference results.

```

134 % Visualization of the difference (Figure 3)
135 figure(3)
136 contourf(flip(difference)) % Flip matrix vertically for correct orientation
137 colormap("parula") % Use 'jet' colormap for clarity
138 colorbar % Add colorbar to denote value ranges
139 c=colorbar; %Attaches color bar to figure 1, representing temp
140 c.Location = 'southoutside';
141 c.Label.String = 'Temperature';
142 clim([-0.005 2]); % Set colorbar limits (Replace 'min_value' and 'max_value' with your desired range)
143 xlabel('X distance in meters');
144 ylabel('Y distance in meters');
145 title('Difference Between Numerical and Analytical Solutions');

```

Fig. 19 Code, lines 134-145

These final segments take the analytical results and create a temperature distribution chart.

```

147 % Function to calculate the analytical temperature distribution
148 function T_analytical = calculate_analytical_temperature(x, y, num_terms)
149     T_analytical = 0;
150     for n = 1:num_terms
151         term = (sin(2*n*pi/3) / (n^2 * sinh(n*pi))) * sin(n*pi*x/6) * sinh(n*pi*y/6);
152         T_analytical = T_analytical + term;
153     end
154     T_analytical = (450/pi^2) * T_analytical;
155 end

```

Fig. 20 Code, lines 147-155

6. Code Validation

6.1 Importance

In MATLAB, code validation is crucial to help guarantee that it functions properly, and scripts operate accurately, quickly, and error-free. The validation of the code can help to improve the following factors:

6.1.1 Accurate Results: Validation assists in guaranteeing that the code generates accurate and correct results, which is crucial, particularly in data analysis. It can also assist in error detection by locating and resolving faults and errors in the code before they cause inaccurate output or system malfunctions.

6.1.2 Efficiency: By identifying inefficiencies and performance issues through code validation; optimisation and quicker task execution is made possible. Efficient codes help to make the best use of computational resources, which is critical for large-scale simulations and data processing activities.

6.1.3 Maintainability: Readability of Code is important to help make the code maintainable. Validation often involves adhering to coding standards and practices that improve the readability and understandability of the code. As part of the validation process, a well-documented code helps to facilitate understanding, modification, and maintenance for both you and others in the future.

6.1.4 Duplicability: Validating code helps to guarantee that it generates consistent results when used in various runs and settings, this is essential for reproducibility.

6.1.5 Risk Management: Code validation lowers the possibility of unexpected errors that might result in serious problems. In industry, this can include monetary losses or jeopardised security in vital systems. Making sure that the code operates as intended under different circumstances aids with planning and risk management.

6.1.6 Confidence: Results produced by a validated code are more likely to be trusted and relied upon by users to give an accurate result.

6.2 Methods

MATLAB offers several strategies and tools to aid with code validation and debugging.

6.2.1 Manual Testing:

To check whether the results are what you expected, run functions and scripts with various sets of input data.

6.2.2 Automated Testing:

Use MATLAB's built-in testing framework. According to MathWorks.com (2024), You can develop test cases for the desired functions using the built-in unit testing framework in MATLAB.

6.2.3 Debugging tools

Breakpoints: Create breakpoints in your code to halt execution and inspect variables.

Step Execution: Use the debugging toolbar or commands such as 'dbstep', 'dbcont' and 'dbquit' to step through your code line by line.

Variable Inspection: Inspect and change variables in the workspace while troubleshooting.

6.2.4 Code Analyser

The Code Analyzer in MATLAB examines your code for any problems and offers suggestions for enhancing code quality. It may be executed using the 'checkcode' function in the MATLAB Editor.

6.2.5 Documentation

Having clear documentation makes it easier to comprehend and maintain the code. Documentation blocks ('%{... %}') and comments are useful for explaining the use and functionality of scripts and functions

7. Analysis of the code results

I ran the code numerous times to test three variables: Time Step, Tolerance and Grid Spacing. For each variable I wanted to see how each would affect the result.

7.1 Change in Step

In this set of result I chose to alter just the time step value. This value dictates how many steps in seconds to skip for each reading. The smaller the number the more samples the programme needs to take. Looking at the results, the initial difference between the smaller jumps in time is vast with an initial sharp incline. However, as the value of the time step reaches around 600 seconds, the line slope begins to plateau. The graph suggests that if the time step was to increase further, there would be limited change in the thermal equilibrium result. However, this could be due to the time steps being so large and the equilibrium is possibly being met sometime between the last two large steps.

The steps on the small end may be causing the programme to presume that the material has met equilibrium prematurely if the temperature is not changing fast enough for the programme to pick up on.

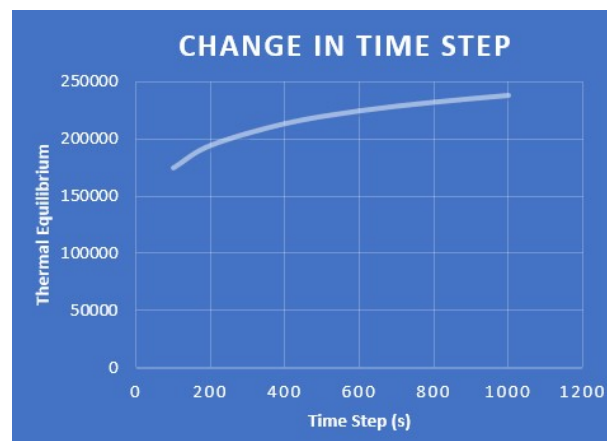


Fig 21. Graph of time step variable against thermal Equilibrium

Grid Spacing h (fixed)	Time Step	Tolerance (Fixed)	Thermal Equilibrium
0.75	1000	0.0001	238000
0.75	800	0.0001	232000
0.75	600	0.0001	224400
0.75	400	0.0001	213200
0.75	200	0.0001	194000
0.75	100	0.0001	174200

Fig 22. Table of results for time step variable

7.2 Change in Tolerance

In this set of result I chose to alter just the tolerance value. This value dictates how precise the results need to be for each reading. As you would think, the graph below shows that the more constraint the tolerances are, the longer it took to reach equilibrium. If the tolerance is made larger, the programme is able to run much faster. However, this comes at the cost of having results that are much less accurate.



Fig 23. Graph of Tolerance variable against thermal Equilibrium

Grid Spacing h (fixed)	Time Step (Fixed)	Tolerance	Thermal Equilibrium
1	500	0.1	29000
1	500	0.001	89500
1	500	0.0001	155500
1	500	0.00001	222000
1	500	0.000001	288000
1	500	0.0000001	354500

Fig 24. Table of results for tolerance variable

7.3 Change in Grid Spacing

In this set of result I chose to alter just the Grid spacing value. This value dictates how much space is between each node for each reading in meters. The more nodes in the area, the greater the number of opportunities to pick up temperature changes in the material. The results show the generally, the more space you put in between the nodes, the longer the equilibrium takes to reach. This could be because each node is having to pick up more area of changes, meaning that they pick up more temperature changes per node.

An outlier to the results is when the grid spacing is at 2 meters. The reasoning for this is unclear. There could be an issue with the model that is producing this result. The graph does show that the gradient of the line does begin to taper off as the grid spaces get smaller than 1.5 meters. This could suggest that after getting smaller than 0.75, it may not make a significant difference to the thermal equilibrium result.

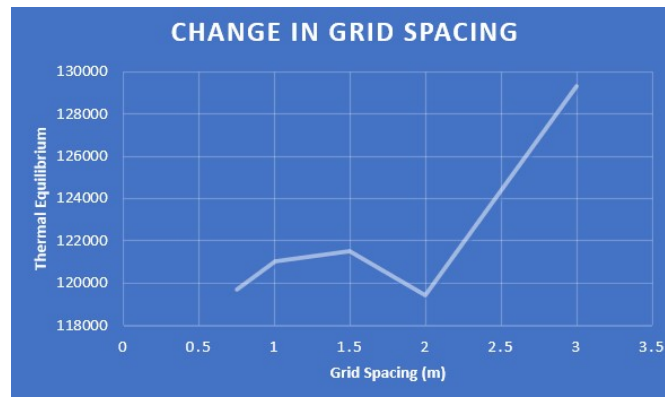


Fig 25. Graph of Grod Spacing variable against thermal Equilibrium

Grid Spacing h	Time Step (Fixed)	Tolerance (Fixed)	Thermal Equilibrium
3	150	0.0001	129300
2	150	0.0001	119400
1.5	150	0.0001	121500
1	150	0.0001	121050
0.75	150	0.0001	119700
0.5	150	0.0001	119100

Fig 26. Table of results for Grid Spacing variable

8. Limitations

For both methods Analytically and numerically, there are some limitations that can be experienced.

8.1 Taylor Series

8.1.1 Local Approximation

The Taylor series is a local approximation centred on a single place. Its accuracy decreases as you go away from this spot. It is not practical for functions with large changes or non-linear behaviour distant from the expansion point, since many terms are needed for precision.

8.1.2 Computational Complexity

Higher-order derivative calculations can be difficult and computationally costly. Because of this, the Taylor series approach may not be feasible for complicated functions or higher-order expansions.

8.1.3 Analytical Derivatives

Analytical expressions for derivatives are required, but they may not always be available or straightforward to get. This may limit the Taylor series' use to functions with known and easily computed derivatives.

8.1.4 Handling Cutoffs

Taylor series are unable to handle discontinuities or non-smooth functions well. Functions including discontinuities, abrupt corners, or non-analytic points cannot be correctly approximated.

8.2 EFD

8.2.1 Stability Issues

The EFD approach can be unstable for stiff differential equations unless small time increments are applied. Stability limitations might render the EFD approach unworkable for difficult situations, necessitating the use of implicit methods or specialised solvers.

8.2.2 Accuracy

The accuracy of the EFD approach is directly proportional to the step size. Smaller step sizes improve accuracy but add computational expense. There is a trade-off between precision and processing efficiency, and determining the best step size can be difficult.

8.2.3 Accumulation of Errors

Numerical errors can accrue over time, particularly in extended simulations or with small time increments. This might result in considerable deviations from the correct answer over extended time periods or over huge areas of space.

8.2.4 Boundary Condition Handling

Implementing boundary conditions correctly can be difficult, especially with complicated geometries or non-standard boundary conditions. Poor treatment of boundary conditions might create mistakes and reduce overall solution accuracy.

8.2.5 Non-linear Equations

It can be difficult to solve non-linear differential equations using the EFD approach; linearisation or reiterative solutions may be needed. This increases processing overhead and complexity, and convergence isn't always certain.

8.2.6 Dimensions

Higher-dimensional issues require more processing and complexity from the EFD approach. It takes a lot of computing effort to solve PDEs in 2D or 3D, and memory and processing power management becomes problematic.

9. Conclusion

The goal for this project was to create an efficient code that could compute the temperature distribution of a 6m x 6m aluminium plate over a desired length of time using the EFD method. Using the Explicit Finite Difference method in the MATLAB software was proven to be a great success, showing that it could easily work out the mass number of simultaneous equations required to simulate the task in a short length of time.

The results did show that when setting the boundaries for a code, if they are not set within a sensible set of parameters, it can easily lead to inaccurate findings. I found that Boundary conditions are essential to the precision of the results obtained in the model. Precise boundary conditions help to guarantee the solution's stability, meets the right answer, and provides an accurate representation of the test. Boundary condition mistakes or inaccuracies can have a big effect on the solution, producing inaccurate and untrustworthy results. This sentiment can be echoed for other inputs such as the time-step size and grid spacing. Each can greatly reduce the accuracy of the results if not inputted correctly.

Looking at the two methods of processing this kind of information, analytically with the Taylor Series and numerical with EFD, there are some pros and cons they both have.

The Taylor series does provide high accuracy for as long as there are sufficient terms, but it is better suited to approximating local functions. Where as the EFD method can be accurate but is dependent on the step size of the results. The smaller the step, the greater the accuracy of the results. Though, this method can have stability issues when faced with stiff equations. EFD certainly has its major advantage of being able to conduct a vast number of simultaneous calculations within a short window of time, however, the software is subject to round-off errors due to finite precision arithmetic. Over time, these small round-offs can accumulate and lead to much larger margin of errors.

It is also important to understand that the results are based off a theoretical simulation in perfect conditions. The software does not consider for environmental, material condition factors, and other variables that may have an impact.

The Taylor series is an effective technique for analytically approximating functions, especially for local approximations. On the other hand, the Explicit Finite Difference technique is a straightforward numerical method for solving differential equations. Each strategy has its advantages and is appropriate for different challenges. In MATLAB, selecting between them relies on the situation at hand and the required precision.

10. References

Byjus.org (2022) *Fourier's law - formula, derivation, definition, equation, BYJUS*. Available at: <https://byjus.com/physics/fouriers-law/> (Accessed: 19 January 2024).

Davies, T.W. (2023) *Fourier number, THERMOPEDIA*. Available at: <https://www.thermopedia.com/content/782/> (Accessed: 20 January 2024).

Maplesoft.com (2023) *Taylor series, Maplesoft*. Available at: <https://www.maplesoft.com/ns/math/taylor-series.aspx> (Accessed: 21 January 2024).

Pratik and Dikov, V. (2023) *Fourier number: Definition, formula, significance (heat transfer), Mech Content*. Available at: <https://mechcontent.com/fourier-number/> (Accessed: 22 January 2024).

Physlab (2023) *The Thermal Diffusion Equation, The Thermal Diffusion Equation - PhysLab*. Available at: <https://physlab.org/wp-content/uploads/2016/04/Doc.pdf> (Accessed: 20 January 2024).

Testbook.com (2022) *Taylor series: Formula, theorem with proof method & examples, Testbook*. Available at: <https://testbook.com/maths/taylor-series> (Accessed: 19 January 2024).

University of Manitoba (2022) *Numerical Differentiation - University of Manitoba, University of Manitoba*. Available at: <https://home.cc.umanitoba.ca/~farhadi/Math2120/Numerical%20Differentiation.pdf> (Accessed: 20 January 2024).

11. Appendix (MATLAB Script)

```
%Clearing Command
clc
clear
close all

% Aluminium Properties
K=143;           %Conductivity
D=2.8*10^3;      %Density
C=795;          %Specific heat
alpha=K/(D*C);   %Alpha calculated using above values

%Dimension
L= 6;           %X Variable
W= 6;           %Y Variable

%Grid Spacing
h=input("Enter dimesnion for grid spacing, 1,2 or 3m:"); %User Input for grid
spacing value
while mod(L,h)~=0 || h<=0 || h>=W
    fprintf('Grid value is Invalid. ');
    h=input("Please Enter valid grid spacing value: ");
end

%Time Steps
dt=input("Enter the value of time steps in seconds: "); %User Input for time step
value
Fo=(alpha*dt)/(h^2); % Fourier Number
while Fo<=0 || Fo>0.25
    fprintf("Time step is invalid. ");
    dt=input("Please Enter valid time step value: ");
end

%Defining rows and columns
NR=W/h+1;
NC=L/h+1;

%Creating initial T1
T1=zeros(NR,NC);

% Amend Boundary Conditions Of Initial Matrix
for c=1:NC
    d=(c-1)*h;
    if d<=2/3*L
        T1(1,c)=12.5*d;
    else
        T1(1,c)= -25*d+150;
    end
end

%Create T2 Matrix from T1 Matrix with temp calculations
T2=T1;
Tol=0.001; % Tolerance for equilibrium
Dif=Tol+1; % Initial condition for max value in difference matrix (Dif) for
initial condition to be false.
Count=0;
```

```

%While loop set up to repeat calculations until the difference between T2 and T1
(Dif) is less than 0.01 and therefore close to equilibrium.
while Dif>Tol %Condition to initiate/continue loop
    T1=T2;
    for G=2:NR-1 %Dimension to ignore perimeter nodes
        for H=2:NC-1 %Dimension to ignore perimeter nodes
            T2(G,H)=Fo*(T1(G-1,H)+T1(G,H-1)+T1(G,H+1)+T1(G+1,H)-(4-
(1/Fo))*T1(G,H)); %Taylor series formula used to calculate T1 values at the
next time step
        end
    end
    Dif=max(max(abs(T2-T1)));
%Formula to determine if thermal equilibrium has been reached

figure(1)
contourf(flipud(T2))
%Command to create graph
title ('Temperature Distribution of plate'); %Title for Fig1
colormap ('parula') %Colour system used for figure 1
c=colorbar; %Attaches colour bar to figure 1, representing temp
c.Location = 'southoutside';
c.Label.String = 'Temperature';
xlabel('X distance in meters');
ylabel('Y distance in meters');
pause(0.2) %Pauses after each step to update graph to visually show the change
along plate
Count=Count+1; %Update the amount of cycles completed
end

% Thermal equilibrium time
eq=Count*dt; %Calculate the time taken to reach equilibrium by multiplying the
count by the time step

if eq<60 %Will display in seconds
    fprintf ('It took %g seconds to reach thermal equilibrium\n', eq)
% Display amount time taken to reach equilibrium in seconds

elseif eq>=60 && eq<3600 %Will display in minutes
    eqM=eq/60;
    fprintf ('It took %g minutes to reach thermal equilibrium\n', eqM)
% Display amount time taken to reach equilibrium in minutes

elseif eq>=3600 && eq<86400 %Will display in hours
    eqh=(eq/60)/60;
    fprintf ('It took %g hours to reach thermal equilibrium\n', eqh)
% Display amount time taken to reach equilibrium in hours

elseif eq>=86400 %Will display in days
    eqD=((eq/60)/60)/24;
    fprintf ('It took %g days to reach thermal equilibrium\n', eqD)
% Display amount time taken to reach equilibrium in days
end

fprintf ('(%g seconds)\n', eq) %Time in seconds for analysis

% Analytical Solution Calculation
% Calculate the analytical solution
num_terms = 100; % Number of terms in the series to include

```

```

T_analytic = zeros(NR, NC);
for G = 1:NR
    for H = 1:NC
        x = (G - 1) * h;
        y = (H - 1) * h;
        T_analytic(G, H) = calculate_analytical_temperature(x, y, num_terms);
    end
end
T_analytic = rot90(T_analytic); % Rotate matrix 90 degrees counterclockwise for
proper orientation

% Display the analytical solution in the command window
disp('Analytical Temperature Distribution at Thermal Equilibrium:');
disp(T_analytic);

% Calculate the difference between analytical and numerical solutions
difference = T1 - T_analytic; % Compute the difference between solutions
disp('Difference Between Numerical and Analytical solutions:');
disp(difference);

% Visualization of the analytical solution (Figure 2)
figure(2)
contourf(flip(T_analytic)) % Rotate by 90 degrees (clockwise) and then flip
vertically
colormap("parula")
colorbar % Keep the colorbar on the right side
c=colorbar; %Attaches color bar to figure 1, representing temp
c.Location = 'southoutside';
c.Label.String = 'Temperature';
clim([0 50]); % Set colorbar limits (Replace 'min_value' and
'max_value' with your desired range)
xlabel('X distance in meters');
ylabel('Y distance in meters'); % Adjust the label accordingly
title('Analytical Temperature Distribution');

% Visualization of the difference (Figure 3)
figure(3)
contourf(flip(difference)) % Flip matrix vertically for correct orientation
colormap("parula") % Use 'jet' colormap for clarity
colorbar % Add colorbar to denote value ranges
c=colorbar; %Attaches color bar to figure 1, representing temp
c.Location = 'southoutside';
c.Label.String = 'Temperature';
clim([-0.005 2]); % Set colorbar limits (Replace 'min_value' and
'max_value' with your desired range)
xlabel('X distance in meters');
ylabel('Y distance in meters');
title('Difference Between Numerical and Analytical Solutions');

% Function to calculate the analytical temperature distribution
function T_analytical = calculate_analytical_temperature(x, y, num_terms)
    T_analytical = 0;
    for n = 1:num_terms
        term = (sin(2*n*pi/3) / (n^2 * sinh(n*pi))) * sin(n*pi*x/6) *
sinh(n*pi*y/6);
        T_analytical = T_analytical + term;
    end
    T_analytical = (450/pi^2) * T_analytical;
end

```