UNIVERSITY OF BATH

Modelling of a Spacecraft Heat Shield Tile

Alex Howells, Hariram Gnanachandran, Callum Geissmar

Group 01

April 2024

Abstract

This report contains a comprehensive analysis and solution of the 1-dimensional heat equation through the use of forward differencing, DuFort-Frankel, backwards differencing and Crank-Nicolson numerical methods implemented in MATLAB. The best solution methods were determined to be backwards differencing and Crank-Nicolson due to their implicit nature and subsequent unconditional stability and high accuracy in both time and spatial dimensions. An optimal tile thickness of 0.07m was calculated to ensure the internal structure of the spacecraft was not damaged at tile location 597. Below this thickness, however, the tile itself was susceptible to heat damage, which was modelled in 2 spatial dimensions. The final model for the best method was displayed in a Graphical User Interface (GUI) where the external temperature boundary conditions were automatically detected and scanned from temperature graphs provided by NASA.

1 Introduction

The thermal protection tiles lining the exterior of a space shuttle serve as a crucial barrier against the extreme temperatures encountered during re-entry into Earth's atmosphere. To ensure the heat shield's integrity for a reusable spacecraft, the temperature variation through various tile thicknesses during re-entry were numerically modelled, from which an optimal thickness was determined. The relevant thermal properties of the tile, including thermal conductivity, density, and specific heat, form the basis of our mathematical model. External thermocouple temperature measurements taken by NASA on the Space Shuttle were also used as boundary conditions for the analysis. In this report the numerical modelling of the temperature variation across a tile is achieved using these key assumptions:

- Tile material is uniform and isotropic, with constant material properties.
- Tile (initially) modelled as one dimensional with heat acting through a tile from a prescribed outer surface temperature inwards.
- Heat transfer through the tile occurs via conduction only; radiation and convection transfer are not considered.

2 Theory

2.1 Partial Differential Equation

The numerical model for the heat shield is fundamentally built from a Partial Differential Equation (PDE), where the dependent variable temperature is related to two independent variables, time and position. To produce a solution, first the PDE describing the heat shield system was identified as the Heat Equation. The PDE was then discretized by time and spatial steps, Δt and Δx. Time steps govern the progression of temperature changes over time, while space steps determine the resolution of temperature distribution across space.

2.2 The Heat Equation

The Heat Equation describes the distribution of temperature in a given region over time. As the tile is modelled in 1D, the heat equation is described along a single spacial dimension, x, and is given by [1]:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \quad \text{where} \quad \alpha = \frac{k}{\rho C_P} \tag{1}$$

$u(x,t)$ = Temperature at position x and time t (^oC), $\frac{\partial u}{\partial t}$ = First Time Derivative of Temperature, $\frac{\partial^2 u}{\partial x^2}$ = Second Spatial Derivative of Temperature , α = Thermal Diffusivity (m^2/s), ρ = Density, (kg/m^3) C_P = Specific Heat Capacity, (J/kgK)

The numerical solution of temperature $u(x,t)$ at position x and time t is indexed with subscript i in the spacial dimension and superscript n time dimension, i.e. u_i^n represents current temperature value in time and space.

2.3 Explicit vs Implicit Methods

The methods employed to solve the Heat Equation can be categorised into two main approaches: explicit methods, including forward differencing and DuFort-Frankel, and implicit methods, including backwards differencing and Crank-Nicolson. Explicit methods calculate the system's state at a future time solely based on its current state. In contrast, implicit methods rely on both the current and future states of the system to predict its state at a later time. [2]

3 Method

3.1 Basic Task

To fully describe the heat equation, the thermal diffusivity variable, α, was calculated using material properties obtained from NASA [3] and remained constant throughout all models in accordance with our material assumptions. In addition, the tile's outside temperature boundary conditions were established using graphical data from the Space Shuttle provided by NASA, an example of which is shown in Figure 1.

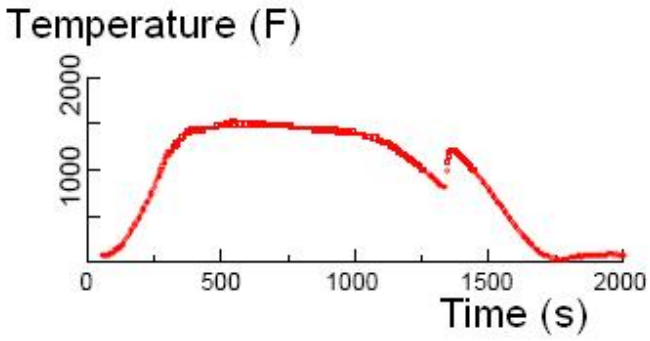

Figure 1: Exterior Tile Temperature during Re-Entry [4]

To extract the temperature data from Figure 1, the MATLAB script, *plottemp* was created. The script allows for an image of graphical data to be traced and processed using mouse inputs, by setting the origin and top-right corner reference points for scaling, the script takes multiple mouse inputs along the curve and stores the temperature and time data values as a .mat file with the same name as the image.

The function *calctemp* contains the calculated α variable and the external boundary conditions imported from the *plottemp* script. With Equation 1 now fully described, the first numerical method, forward differencing was implemented. The forward differencing method involves updating the temperature of the tile at each time step Δt and spacial step Δx. These two steps were combined into one non-dimensional timestep for every method, given as p, shown in Equation 2. The numerical approximation is calculated using known values at the current time step, hence its explicit nature. The approximation to the 1D heat equation using the forward differencing approach is given as [5]:

$$u_i^{n+1} = (1-2p)u_i^n + p(u_{i-1}^n + u_{i+1}^n) \quad \text{where} \quad p = \frac{\alpha \Delta t}{\Delta x^2} \tag{2}$$

The initialization of u values included the outside boundary condition [specified above], while the inside boundary condition was initially set to 0. Using the forward differencing update equation (Equation 2), the subsequent temperature values in space and time were explicitly determined. This process continued until the system was solved for 4000 seconds, with the final 2000 seconds of data representing a constant temperature input set to the final temperature value from Figure 1.

Following the implementation of forward differencing, the DuFort-Frankel method was incorporated into the *calctemp* function. The DuFort-Frankel method differs from forward differencing due to its central differencing derivation and use of values from previous timesteps; where values from two timesteps are used to calculate the new value. The approximation to the 1D heat equation using the DuFort-Frankel approach is given as [5]:

$$u_i^{n+1} = \frac{1}{1+2p}\left((1-2p)u_i^{n-1} + 2p(u_{i-1}^n + u_{i+1}^n)\right) \quad \text{where} \quad p = \frac{\alpha \Delta t}{\Delta x^2} \tag{3}$$

The same variable and boundary conditions as previously used were employed, except for the first temperature data value. This is due to the Dufort Frankel method relying on a previous timestep value, however, at the original location, a previous timestep does not exist. In this case, Equation 3 was modified to replace u_i^{n-1} with u_i^n.

With both explicit methods implemented, the first implicit method used to solve the heat equation was the backwards differencing method, added to the *calctemp* function and is given by the following update equation[6]::

$$u_i^{n+1} - u_i^n = p(u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}) \quad \text{where} \quad p = \frac{\alpha \Delta t}{\Delta x^2} \tag{4}$$

As seen in Equation 4, three future and therefore unknown values of u are involved, resulting in a set of simultaneous equations to solve for u_i^{n+1}. Solving this system of equations is carried out within the *tdm* function using the tri-diagonal matrix method, followed by back substitution to determine the new temperature value for the next time step. The *tdm* function takes the simultaneous equation inputs established in *calctemp* and provides the new temperature value as outputs. Similar to previous methods, it also uses a constant final temperature value beyond the 2000 seconds of provided temperature data.

The final method implemented in *calctemp* was the Crank-Nicolson method. Unlike the backward differencing method which approximated the heat equation at the new timestep (using values at the new time step) the Crank-Nicolson method centers itself at the mid-point between the current and next timestep, i.e. $t_n + \frac{\Delta t}{2}$, and projects forward by $\frac{\Delta t}{2}$ to the future timestep [6]. This approach provides the following approximation to the 1D Heat Equation [6]:

$$u_i^{n+1} - u_i^n = \frac{p}{2}\left(u_{i-1}^n - 2u_i^n + u_{i+1}^n + u_{i-1}^{n+1} - 2u_i^{n+1} + u_{i+1}^{n+1}\right) \quad \text{where} \quad p = \frac{\alpha \Delta t}{\Delta x^2} \tag{5}$$

Due to the implicit nature of the method, the update equation contains multiple unknowns leading to a system of simultaneous equations. These are solved using the *tdm* function which uses the set value inputs from *calctemp* in much the same way as the backwards differencing method.

With the four modelling methods implemented in the *calctemp* function, the model was improved by adjusting the boundary condition on the inner surface from a Dirichlet boundary to a Neumann boundary. Unlike the Dirichlet condition which sets the boundary value to a known constant (in this case 0); the Neumann boundary condition uses an unknown boundary value with a known gradient [6]. In the context of the heat equation, the Neumann boundary gives an inner surface with zero heat flow at an unknown temperature. In the explicit methods, this was achieved by introducing an inner boundary equation to calculate the temperature value at the inner surface with no thermal heat transfer($\frac{\partial u}{\partial x} = 0$). The new temperature value equations for the explicit methods were modified accordingly. For instance, in the case of Forward Differencing, the Neumann Boundary Condition Equation on the inner surface is expressed as:

$$u_i^{n+1} = (1 - 2p)u_i^n + 2pu_{i-1}^n \quad \text{where} \quad p = \frac{\alpha \Delta t}{\Delta x^2} \tag{6}$$

Here, $2pu_{i-1}^n$ has replaced $p(u_{i-1}^n + u_{i+1}^n)$ from Equation 2; as on the boundary u_{i-1}^n and u_{i+1}^n are equivalent due to the zero heat transfer condition. The same manipulation was implemented with the DuFort-Frankel method with the resulting update equation, (Equation 3).

To accommodate the new Neumann boundary condition for the implicit methods, the simultaneous matrix equations were changed, resulting in a modified input into the *tdm* function from *calctemp* for the right boundary for both the backward differencing and crank-nicolson methods.

In addition to the various numerical method solutions to the heat equation, each method's stability and accuracy were investigated within the functions *investMethodsTime* and *investMethodsSpace*. The function *investMethodsTime* plots the inner surface temperature at 4000s against the time step used to determine how the time step changes the final determined inner temperature. The allowable accuracy percentage was set at 5% and the function outputs the largest possible time step for each of the four methods before the desired accuracy is compromised. Similarly, the function *investMethodsSpace* completes the identical process except compares the final temperature against the spacial step size and uses an allowable accuracy percentage of 0.1% then outputs the largest spacial step before the desired accuracy is compromised. The function *detBestMethod* takes in the final time and spacial step

as inputs and traverses through each of the four methods determining the difference between the time and space steps. The time step is to be maximised and space step minimised to maximise overall accuracy and stability. The function then outputs the best method of the four numerical methods used.

With the most accurate method determined, the final step was to determine the smallest tile thickness without any damage occurring to the internal frame of the shuttle. This was achieved within the *detTileThickness* function which takes inputs of the best method alongside time and temperature data and time and spacial step sizes all determined previously while returning the final tile thickness in meters. The temperature at which the internal frame can safely withstand depends on its material properties and was assigned a value of 120^oC based on the NASA data for the internal temperature [7].

The script *main* contains and connects all the basic task code and functions to provide a central script that computes and produces the entirety of the basic modelling task.

3.2 Extension Tasks

While *plottemp* was a sufficient method of extracting the initial temperature and time data from the different tile plots, an automated method was devised to read and scale the data in the graph consisting of *ImgScan* and *DataScale*. *ImgScan* converts the selected *.jpg* image file to a binary matrix achieved by analysing the blue colour content of each pixel to determine whether the data point is white or not, the results from this analysis are stored in a matrix p where 1's dentoe a white pixel and 0's black. A grayscale image was analysed as opposed to an RGB image to decrease the computation time and memory used to analyse the image. Another colour analysis is performed this time recording the location of all red pixels within a range which is stored in the matrix q. The location of the graph origin is determined by summing the rows and columns in the p matrix, the location of the lowest row and column sums represent the origin of the figure. For scaling purposes the projected top right corner *(TRC)* of the axes is found by iterating through the p matrix to find the length of the longest continuous row and column of 0's, these represent the axes length which are used to find a projected *TRC*. Knowing this, all data not encompassed by the graph axes and *TRC* is removed from the q matrix which now contains only the time and temperature plot pixel locations and is free of other information. Finally, the line width spanning more than one pixel in each graph is accounted for by taking only the highest temperature reading from the q matrix found by taking the lowest row in each column from all the data entries.

DataScale uses the origin and TRC locations as well as the 2000 second time axis and 2000 Fahrenheit temperature axis to scale the data extracted from the image from the MATLAB coordinate system to the one in the image. The temperature data is also converted from degrees Fahrenheit to degrees Celsius for the calculations. The resulting data is stored in two arrays *tempData* and *timeData* containing the scaled temperature and time data respectively which will be used as the Dirichlet boundary for the outside of the tile. *ImgScan* and *DataScale* work for any of the eight graphs from the eight different tile locations in figure 2.

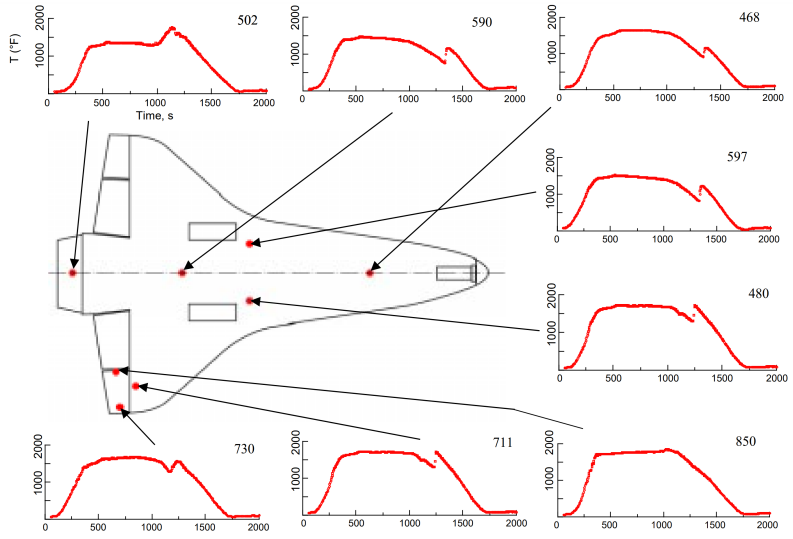

Figure 2: Tile Locations and corresponding Thermocouple Data [4]

The 1D model of the tile was then improved by considering the tile in 2 dimensions, both through its thickness and across its surface to account for heat damage that affects the tile's performance protecting the inner structure. This was achieved by altering the *calctemp* function to *caltemp2D* and was implemented for every method. The modification to 2D resulted in the use of an extended heat equation in 2 dimensions, with an additional spatial dimension $\frac{\partial^2 u}{\partial y^2}$. This addition to the heat equation resulted in alternative update equations for each method; for example, the new forward differencing approximation to the 2D heat equation is given by [6]:

$$u_{i,j}^{n+1} = (1-4p)u_{i,j}^n + p(u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n) \quad \text{where} \quad p = \frac{\alpha \Delta t}{h^2} \quad (h = \Delta x = \Delta Y) \tag{7}$$

In Equation 7, the subscript j represents the index in the second spatial dimension. The explicit methods were altered simply by the extension into the extra spacial dimension changing each method's update equation, however, the implicit solutions fail when expanded to 2 dimensions as the resulting matrix is not tri-diagonal. Consequently, an iterative approach, such as Gauss-Seidel, became necessary to solve the new matrix. Through iterative adjustments of temperature values at each grid point based on their neighboring points, Gauss-Seidel gradually converges towards a solution for the heat equation in two dimensions. To control the iterative process, a maximum iteration limit of 100 and an error tolerance of 0.0004% were imposed, outputting the temperature result upon reaching either condition.

New internal Neumann boundary conditions were implemented to compute the methods in two dimensions to represent the internal surface in 2D. This was achieved in the explicit methods by individually computing a new Neumann boundary for each side of the tile being the north, south, west, north-west and south-west boundaries computing them individually at each boundary within *calctemp2D*. Note, that the north-west and south-west corners were individually processed as the index vectors did not include these grid points. Similarly, for the implicit methods, the Neumann boundary was processed for the north, south and west boundaries which are the boundary values for Gauss-Seidel to iterate towards.

In order to analyse the heat damage done to the tile, the function *damageAnalysis* was created, which took inputs of the temperature values at each internal section of the tile generated for each method. The function then determines whether that section of the tile is damaged by checking if it exceeds the maximum allowable temperature or if the tile subsection has already been damaged. The maximum temperature of the tile was set at 660 $^\circ C$ which can be adjusted in the code to suit. The function then outputs the area in which the tile is damaged, which was used in a graphical plot to demonstrate the positions, and at what time the tile failed.

A Graphical User Interface (GUI) was developed to allow for easy comparison and display of results for the user. First, a GUI was implemented for the 1D models in a function called *main*. The GUI enables the user to select and simulate the heat flow through tiles at 8 different tile locations, seen in Figure 2, and observe the output parameters generated including the number of time and spatial steps and the tile thickness used to complete the model. Furthermore, the GUI displays the best-calculated method from function *DetBestMethod* [see previous] and displays the full graphical solution to that method in an interactive 3D plot. The GUI is programmed within the *main* function with a series of 'uicontrol' textboxes and output boxes to display the relevant data from the basic task functions such as *calctemp*. Similarly, a 2D GUI was also created displaying the 2 spatial dimensions of the tile against its temperature including the effect of the tile's heat damage - where once the temperature is too high, the tile melts and is destroyed. This data was pulled from the *damageAnalysis* function. The GUI allows for a choice of tile similar to the 1D GUI and also allows the choice of which numerical method is used. The GUI was not constructed in the app generator being instead programmed manually to allow for a better implementation within the main code rather than saving and loading values from the main code into the app generator.

4 Results

4.1 Basic Task

Appropriate values for the number of time steps, nt, and the number of space steps, nx, were then found for Tile 597 for each of the four methods. These were calculated using Figures 4 and 5 which were given tolerances of **5%** and **0.1%** from their starting values. With the best numerical method obtained being backward differencing, nt and nx were **41** and **8**, respectively.

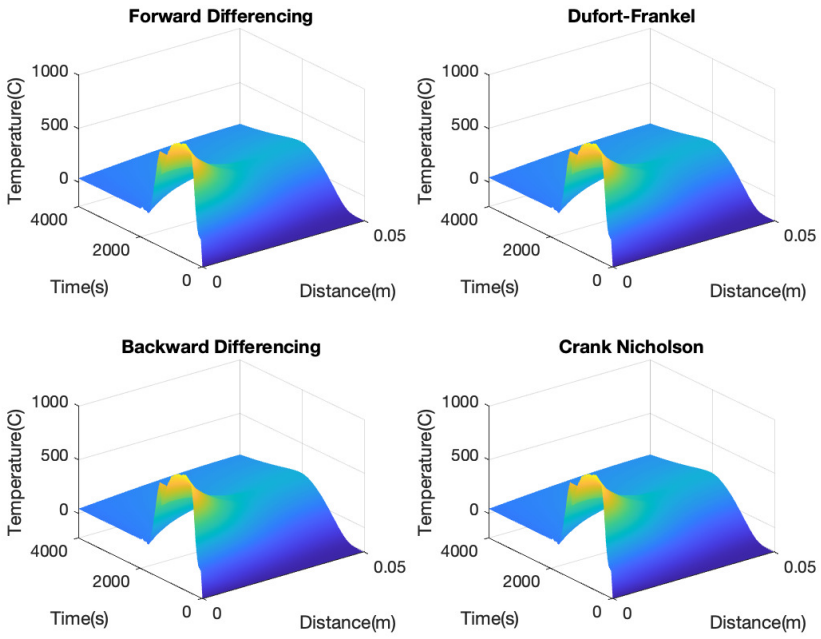

Figure 3: Plots for the Four Numerical Methods before any changes to nx, nt or Tile Thickness

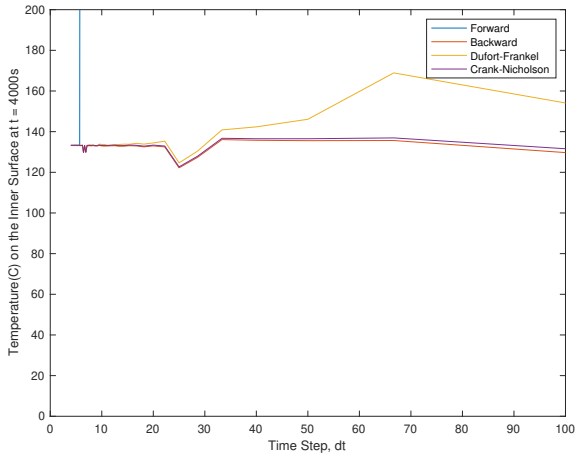

Figure 4: Investigation in Stability and Accuracy of Time Step

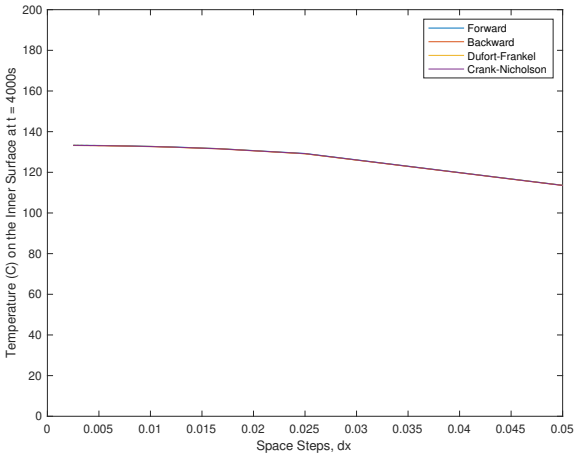

Figure 5: Investigation in Stability and Accuracy of Number of Space Steps

The investigation for an appropriate tile thickness found that for a thickness of **0.07m**, the temperature on the inner surface does not exceed 150°C, shown in Figure 6.

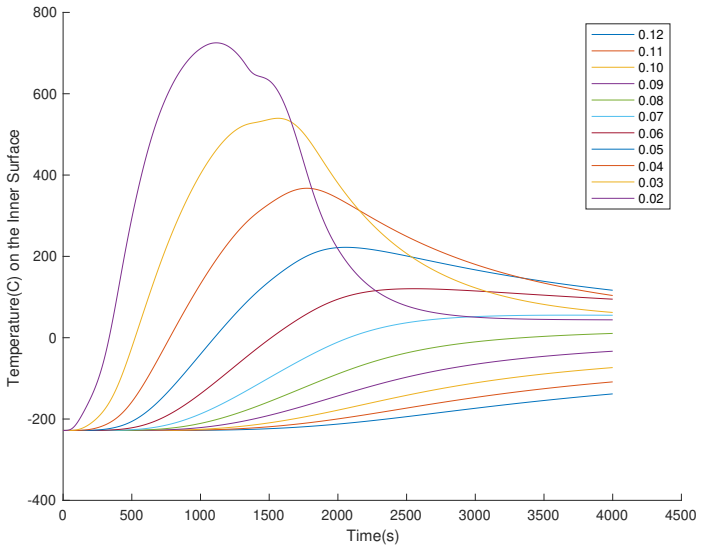

Figure 6: Investigation of Different Tile Thickness

4.2 Extension Tasks

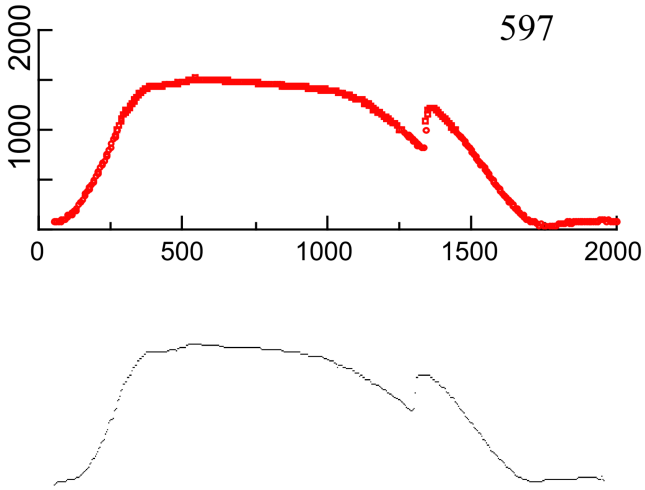

Figure 7: Original Plot for Tile 597 with the Plot scanned by the Autoread Function

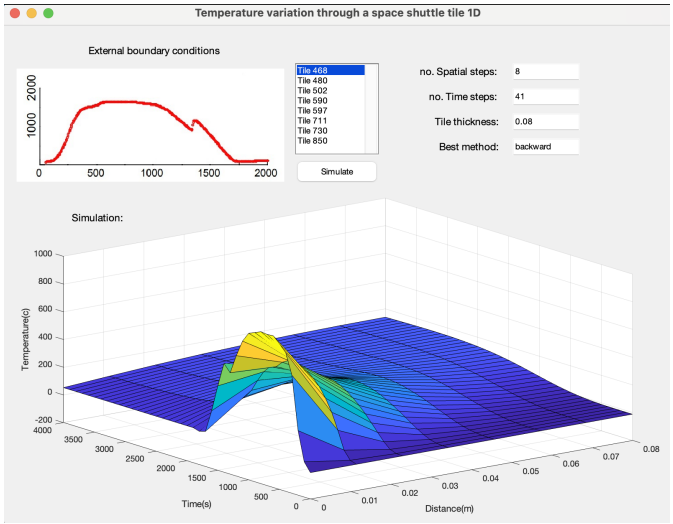

Figure 8: GUI of the Simulation of Heat Flow in the Tile - 1D Problem

A 2D simulation of the damage in the tile can be found **HERE** (Google Drive Video Link).

5 Discussion

Firstly, the forward differencing method displays conditional stability and is unstable for a timestep greater than $\frac{\Delta x^2}{2\alpha}$ [6], shown by the sudden instability in Figure 4 past a timestep of 6s. The method is based on a first-order time approximation and second-order spatial approximation, resulting in first-order accuracy in time and second-order in space ($O(\Delta t), \Delta x^2$). This parabolic accuracy in the spatial dimension is true for all methods and is seen in Figure 5. While forward differencing offers relatively simple implementation and computational efficiency with a simple update equation, its accuracy is limited making the method inefficient and time-consuming [5].

The DuFort-Frankel method utilises a central differencing approach which provides second-order accuracy in time and space making it a more accurate method than forward differencing. Furthermore, the method is unconditionally

stable for all time and spacial steps observed in Figure 4. However, due to its unique derivation utilising an averaged temperature value at the central time step rather than the discrete value introduces an additional numerical error given by $\frac{\Delta t}{\Delta x}$,[8] which is squared to a second order error. This error generally introduces oscillation at large time steps forcing accuracy to be achieved at lower step sizes despite its unconditional stability.

The backward differencing method due to its implicit nature is unconditionally stable for every size step. The implicit nature of the method does result in increased computational and implementation time which is not desired. Similarly to forward differencing the method has first-order accuracy in time and second in space but tends to be more accurate than forward differencing. Inaccuracies are projected backward in time in comparison to forward where they are inclined to accumulate and grow. The backward differencing method was determined the best method for every tile location. In actuality, the backward method produced identical stability and accuracy data compared with the crank-nicolson method and was only selected as the best method due to its position in the data array appearing before the Crank-Nicolson method.

The Crank-Nicolson method is also unconditionally stable but has the advantage over the backward method by having second-order accuracy in time and space. It is therefore theoretically more accurate than the backwards method at the cost of more complexity and computational time. It is likely the reason the two methods were deemed equal by *DetBestMethod* was due to the simplistic nature of the 1D model not requiring the more accurate solution crank-nicolson provides.

Figure 6 shows inner surface temperatures over time for various thicknesses of the shield tile. The function *detTileThickness* considered the limiting temperature at which the internal structure would be unsafe to provide the ideal thickness of 0.07m at tile location 597 (every tile location thickness provided within GUI). In addition, as investigated in the 2-dimensional extension, the damage to the tile due to the extreme temperature is substantial as seen through the hyperlink in section 4.2. The animation shows the tile disappearing once it has reached the critical 660 $^{\circ}C$ which represents the tile melting. To mitigate this damage a thicker tile would be utilised to more effectively disperse the heat throughout a thicker tile to reduce the maximum temperature reached.

An assumption made in the process of the shield modelling was to simulate a tile as having constant material properties, remaining isotropic throughout. This assumption may be inaccurate as heat tiles are commonly made from composite materials, resulting in layers of different material properties throughout their thickness. This consideration may be included to improve the model by allowing for changes in material properties through the 1D spatial dimension. Furthermore, the heat equation PDE may be expanded to include convention and radiation heat transfer, using the Stephan-Boltzman law of radiation to derive a more realistic PDE.

6 Conclusion

In conclusion, the numerical modelling of a space heat shield tile was modeled in 1 dimension using the heat equation to investigate different numerical methods and the effect of the different number of time and space steps for each method. The optimal numerical methods was determined to be the backwards differencing and crank-nicolson methods, both of which are implicit methods with unconditional stability. Additionally, our analysis revealed the importance of selecting an optimal tile thickness to mitigate heat damage to both the tile and internal structure during re-entry, which was selected as 0.07m at tile 597, to not damage the internal aluminium structure. Overall our findings were consistent with theoretical ideas of stability and accuracy, while the overall model could be improved using more accurate assumptions; such as varying tile material properties through the tile width and further development of the fundamental heat equation PDE into higher spacial dimensions and considering additional thermal heat transfer methods.

References

[1] Nagle R, Saff E, Snider A. Fundamentals of Differential Equations, Global Edition. Harlow: Pearson Education, Limited; 2018.

[2] Professessor D.M. Causon, Professor C.G. Mingham. Introductory Finite Difference Methods for PDEs. Ventus Publishing APS; 2010.

[3] Structures and Materials: Space Shuttle Tiles [Internet]. [cited 2024 Apr 18]. Available from: $https://www.nasa.gov/wp-content/uploads/2023/06/shuttle-tiles-5-8v2.pdf$

[4] Blanchard et al., Infrared Sensing Aeroheating Flight Experiment: STS-96 Flight Results, AIAA 2001-0352, 39th AIAA Aerospace Sciences Meeting & Exhibit, Jan 2011, Reno

[5] Adak, M. (2020). Comparison of Explicit and Implicit Finite Difference Schemes on Diffusion Equation. In: Bhattacharyya, S., Kumar, J., Ghoshal, K. (eds) Mathematical Modeling and Computational Tools. ICACM 2018. Springer Proceedings in Mathematics & Statistics, vol320. Springer,Singapore.$https://doi.org/10.1007/978-981-15-3615-1_15$

[6] Chapra S, Canale R. Numerical Methods for Engineers. New York: McGraw-Hill US Higher Ed ISE; 2020.

[7] 16 - Metal matrix, fibre–metal and ceramic matrix composites for aerospace applications, Editor(s): Adrian P. Mouritz, Introduction to Aerospace Materials, Woodhead Publishing, 2012,

[8] 7.6 truncation error, consistency and Convergence [Internet]. [cited 2024 Apr 18]. Available from: $https://folk.ntnu.no/leifh/teaching/tkt4140/_main067.html\#ch5:sec5$

7 Appendix A - 1D Code

main.m

```
% Retrieves the time and temperature data points
[origin, TRC, time, temperature] = ImgScan('temp468R.jpg');
[tempData, timeData] = DataScale(origin, TRC, time, temperature);

%Determines the temperature at each distance for each time step for the 4
%different methods
[x1, t1, u1] = calctemp(4000, 1001, 0.05, 21, 'forward', timeData, tempData);
[x2, t2, u2] = calctemp(4000, 501, 0.05, 21, 'dufort-frankel', timeData,
   tempData);
[x3, t3, u3] = calctemp(4000, 501, 0.05, 21, 'backward', timeData, tempData);
[x4, t4, u4] = calctemp(4000, 501, 0.05, 21, 'crank-nicholson', timeData,
   tempData);

%Plots a subplot for forward differencing
subplot(2,2,1)
surf(x1, t1, u1)
title('Forward Differencing')
xlabel('Distance(m)')
ylabel('Time(s)')
zlabel('Temperature(C)')
shading interp

%Plots a subplot for Dufort-Frankel
subplot(2,2,2)
surf(x2, t2, u2)
title('Dufort-Frankel')
xlabel('Distance(m)')
ylabel('Time(s)')
zlabel('Temperature(C)')
shading interp

%Plots a subplot for backward differencing
subplot(2,2,3)
surf(x3, t3, u3)
title('Backward Differencing')
xlabel('Distance(m)')
ylabel('Time(s)')
zlabel('Temperature(C)')
shading interp

%Plots a subplot for Crank Nicholson
subplot(2,2,4)
surf(x4, t4, u4)
title('Crank Nicholson')
xlabel('Distance(m)')
ylabel('Time(s)')
zlabel('Temperature(C)')
shading interp

%Investigates the Stability and Accuracy of the Four Methods
[finalTimeStep] = investMethodsTime(tempData, timeData);
[finalSpaceStep] = investMethodsSpace(tempData, timeData);
```

```
%Determines which of the Four Methods is the Most Accurate
[bestMethod, index] = detBestMethod(finalTimeStep, finalSpaceStep);

%Determines the Values of dt and nt that are within the Required Accuracy
%whilst also Saving Computational Power
nx = finalSpaceStep(index);
dt = finalTimeStep(index);
nt = round((4000 / dt) + 1);

%Determines the Final Thickness of the Tile that will not affect the Tile
n = detTileThickness(nt, nx,bestMethod, timeData, tempData);

%Plots the Final Graph
[x, t, u] = calctemp(4000, nt, n, nx, bestMethod, timeData, tempData);
figure(5)
surf(x,t,u)
title('Final Graph')
xlabel('Distance(m)')
ylabel('Time(s)')
zlabel('Temperature(C)')
```

ImgScan.m

```
function [origin, TRC, time, temperature] = ImgScan(image_name)

% ImgScan - Scans an image of the inital boundary conidtions for a space
% shuttle tile recording the pixel locations of all the data in the image
% along with the origin and top right corner for scaling.
%
% Input arguments:
% image_name       - Name of image for inital conditions i.e. 'temp597.jpg'
%
% Ouput arguments:
% origin           - 1 x 2 matrix, graph origin
% TRC              - 1 x 2 matrix, graph Top Right Corner
% time             - Time matrix containing all read time values
% temperature      - Temperature matrix containing all read time values


    % Import image for processing and turn it into a grayscale matrix
    image = imread(image_name);
    GreyImage = image(:,:,3);

    % Preallocate matrix size for speed
    [x, y] = size(GreyImage);
    p = zeros(x, y); % Used to find axes
    q = zeros(x, y); % Used to find data

    for n = 1:x
        for m = 1:y
            % Use colour filters to identify white and non white points
            if  GreyImage(n,m) < 100
                p(n,m) = 0; % Marks non-white points with a 0
            else
                p(n,m) = 1; % Marks white points with a 1
            end
```

```
        % Use colour filters to identify red points
        if GreyImage(n,m) < 100 &&  20 < GreyImage(n,m)
           q(n,m) = 0;  % Marks red points with a 0
        else
           q(n,m) = 1;  % Marks non-red points with a 1
        end
    end
end


% Finding Origin
% Sum rows and columns
rowsum = sum(p, 2);
colsum = sum(p, 1);

% Lowest sums will be the Axes
[row_max_0, loc_row_max_0] = min(rowsum);
[col_max_0, loc_col_max_0] = min(colsum);

% Use this to determine the origin
origin = [loc_col_max_0, loc_row_max_0];

% Finding Top Right Corner (TRC) for scaling

% column and row data for analysis
columnData = p(:, origin(1));
rowData = p(origin(2),:);

% Initialise variables
maxZeroLengthR = 0;
currentZeroLength = 0;

% Analysing rows
for i = 1:height(columnData)
    % If data is zero, increment currentZeroLength count
    if columnData(i) == 0
        currentZeroLength = currentZeroLength + 1;
    else
        % Non-zero element, update maxZeroLength if required
        if currentZeroLength > maxZeroLengthR
            maxZeroLengthR = currentZeroLength;
        end
        % Reset currentZeroLength
        currentZeroLength = 0;
    end
end

% Update maxZeroLength if the longest sequence is at the end
if currentZeroLength > maxZeroLengthR
    maxZeroLengthR = currentZeroLength;
end

% Reset variables
maxZeroLengthC = 0;
currentZeroLength = 0;
```

```
% Analysing columns
for j = 1:size(rowData, 2)
    % If data is zero, increment currentZeroLength count
    if rowData(j) == 0
        currentZeroLength = currentZeroLength + 1;
    else
        % Non-zero element, update maxZeroLength if required
        if currentZeroLength > maxZeroLengthC
            maxZeroLengthC = currentZeroLength;
        end
        % Reset currentZeroLength
        currentZeroLength = 0;
    end
end

% Update maxZeroLengthC if the longest sequence is at the end
if currentZeroLength > maxZeroLengthC
    maxZeroLengthC = currentZeroLength;
end

% Using origin and length of axes to find top right corner (TRC) of graph
   axes
TRC = [origin(1) + maxZeroLengthC, origin(2) - maxZeroLengthR];


% Reformatting q using p to only include data within the axes

% Adjust columns
q(:, 1:origin(1)+1) = 1;
% Adjust rows
q(origin(2):x, :) = 1;
q(1:TRC(2), :) = 1;

% Removing double data locations, causes errors
% Largest data point, worst case (highest temperature)
for m = 1:length(q)
    for n = 1:height(q)
        if q(n,m) == 0
            MaxZeroRow = n;      % Record which row the first zero occurs
               in
            q(:,m) = 1;          % Set all other values in the column to 1
            q(MaxZeroRow,m) = 0; % Reset location of highest zero
        end
    end
end

% Storing row and column data of zeros
% Preallocate time and temperature arrays
time = [];
temperature = [];

% Loop through q array
for i = 1:height(q)
    for j = 1:length(q)
        % Identify 0 locations and record rows and columns as time and
```

```
                temperature
            if q(i,j) == 0
                time = [time, j];
                temperature = [temperature, i];
            end
        end
    end
end
```

DataScale.m

```
function [tempData,timeData] = DataScale(origin, TRC, time, temperature)
% DataScale - Scales the data extracted from ImgScan using the origin and
% TRC as scaling factors
%
% Input arguments:
% origin            - 1 x 2 matrix, graph origin
% TRC               - 1 x 2 matrix, graph Top Right Corner
% time              - Time matrix containing all read time values
% temperature       - Temperature matrix containing all read time values
%
% Output arguments
% tempData          - Scaled temperature data
% timeData          - Scaled time data

% Determine multiplier to scale pixel co-ordinates

xDiff = TRC(1) - origin(1);
yDiff = TRC(2) - origin(2);

xMultiplier = 2000 / xDiff;
yMultiplier = 2000 / yDiff;

% Determine actual locations of the points to be used

for n = 1:length(time)

    % Scaling time data
    timeData(n) = ((time(n) - origin(1)) * xMultiplier);

    % Scaling and converting temperature data F -> C
    tempData(n) = (((temperature(n) - origin(2)) * yMultiplier) - 32) /1.8;

end
```

calctemp.m

```
function [x, t, u] = calctemp(tmax, nt, xmax, nx, method, timeData, tempData)
% Function for modelling temperature in a space shuttle tile
% D N Johnston  14/02/24
%
% Input arguments:
% tmax    - maximum time (s)
% nt      - number of timesteps
% xmax    - total thickness (m)
```

```
% nx     - number of spatial steps
% method - solution method ('forward', 'backward' etc)
% timeData - time vector for surface temperatures (s)
% tempData - surface temperature vector (C or K)
%
% Return arguments:
% x      - distance vector (m)
% t      - time vector (s)
% u      - temperature matrix (C or K)
%
% For example, to perform a  simulation with 501 time steps
%   [x, t, u] = calctemp(4000, 501, 0.05, 21, 'forward', timeData, tempData);
%

% Set material properties and derived values (LI-900)
% Obtained from NASA document: Structures and Materials: Space Shuttle Tiles,
   Grades 5-8 - NASA
% Note that we're assuming constant properties.
thermCon = 0.0484; % W/m K; 0.028 BTU/ft/hr/F at 70F and 1 atm
density  = 144;    % kg/m^3; 9 lb/ft^3
specHeat = 628;    % J/kg/K; 0.15 Btu/lb/F

% Initialise everything.
dt = tmax / (nt-1);
t = (0:nt-1) * dt;
dx = xmax / (nx-1);
x = (0:nx-1) * dx;
u = zeros(nt, nx);

iVec = 2:nx-1;
a = 1:nx;

%Determine value of p
thermDiff = thermCon / (density * specHeat);
p = (thermDiff * dt) / (dx ^ 2);

% Use interpolation to get outside temperature at time vector t
% and store it as left-hand boundary vector L.
L = interp1(timeData, tempData, t, "linear", "extrap");

% set initial conditions equal to boundary temperature at t=0.
u(1, :) = L(1);

temp2000 = 0;

% Select method and run simulation.
switch method
    %Simulates the Forward Differencing Method
    case 'forward'

        % Outside boundary condition
        u(:, 1) = L;

        %Traverses through each Time Step (Rows)
        for n=1:nt-1
            %Data only goes up to 2000s, so only Calculates Values for this
```

```
        %Range only
        if (n * dt) < 2000

            temp2000 = u(n+1,1);

        else

            u(n+1,1) = temp2000;
        end

        %Determines the Value for the Next Temperature
        u(n+1,iVec) = (1 - 2 * p) * u(n,iVec) + p * (u(n,iVec-1) + u(n,iVec
            +1));

        %Determines the Values for the Internal Boundary
        u(n+1,nx) = (1-2*p) * u(n, nx) + 2*p*u(n, nx-1);
    end

%Simulates the Dufort-Frankel Method
case 'dufort-frankel'

    % Outside boundary condition
    u(:, 1) = L;

    %Traverses through each Time Step (Rows)
    for n = 1:nt-1
        %Data only goes up to 2000s, so only Calculates Values for this
        %Range only
        if (n * dt) < 2000

            temp2000 = u(n+1,1);
        else

            u(n+1,1) = temp2000;
        end

        %Determines whether the Method is on the Second Row where a
        %Slightly Different Equation is used to Prevent it from
        %Failing
        if (n + 1) == 2
            %Determines the Value for the Next Temperature
            u(n+1,iVec) = ((1-2*p) * u(n,iVec) + 2*p*(u(n,iVec-1) + u(n,
                iVec+1))) / (1+2*p);

            %Determines the Values for the Internal Boundary
            u(n+1,nx) = ((1 - 2*p) * u(n,nx) + 4*p*u(n,nx-1)) / (1+2*p);
        else
            %Determines the Value for the Next Temperature
            u(n+1,iVec) = ((1-2*p) * u(n-1,iVec) + 2*p*(u(n,iVec-1) + u(n,
                iVec+1))) / (1+2*p);

            %Determines the Values for the Internal Boundary
            u(n+1,nx) = ((1 - 2*p) * u(n-1,nx) + 4*p*u(n,nx-1)) / (1+2*p);
        end
    end
```

```
%Simulates the Backwards Differencing Method
case 'backward'

    %Traverses through each Time Step (Rows)
    for n=1:nt-1
        if (n*dt) < 2000

            %Sets Values to be used for the Tri-Diagonal Matrix Method
            b(1)    = 1;
            c(1)    = 0;
            d(1)    = L(n+1);
            a(iVec) = -p;
            b(iVec) = 1 + 2*p;
            c(iVec) = -p;
            d(iVec) = u(n,iVec);
            a(nx)   = -2*p;
            b(nx)   = 1 + 2*p;
            d(nx)   = u(n,nx);

            %Updates the Next Column with the Calculated Temperature
            %Values
            u(n+1,:) = tdm(a,b,c,d);

            temp2000 = L(n+1);

        else
            %Sets Values to be used for the Tri-Diagonal Matrix Method
            b(1)    = 1;
            c(1)    = 0;
            d(1)    = temp2000;
            a(iVec) = -p;
            b(iVec) = 1 + 2*p;
            c(iVec) = -p;
            d(iVec) = u(n,iVec);
            a(nx)   = -2*p;
            b(nx)   = 1 + 2*p;
            d(nx)   = u(n,nx);

            %Updates the Next Column with the Calculated Temperature
            %Values
            u(n+1,:) = tdm(a,b,c,d);
        end

    end

%Simulates the Crank-Nicholson Method
case 'crank-nicholson'

    %Traverses through each Time Step (Rows)
    for n=1:nt-1

        if (n*dt) < 2000
            %Sets Values to be used for the Tri-Diagonal Matrix Method
            b(1)    = 1;
            c(1)    = 0;
            d(1)    = L(n+1);
```

```
                a(iVec) = -p/2;
                b(iVec) = 1 + p;
                c(iVec) = -p/2;
                d(iVec) = (p/2) * u(n,iVec - 1) + (1-p) * u(n,iVec) + (p/2) * u
                    (n,iVec + 1);
                a(nx)   = -p;
                b(nx)   = 1 + p;
                d(nx)   = p * u(n, nx-1) + (1-p)*u(n, nx);

                %Updates the Next Column with the Calculated Temperature
                %Values
                u(n+1,:) = tdm(a,b,c,d);

                temp2000 = L(n+1);
            else
                %Sets Values to be used for the Tri-Diagonal Matrix Method
                b(1)    = 1;
                c(1)    = 0;
                d(1)    = temp2000;
                a(iVec) = -p/2;
                b(iVec) = 1 + p;
                c(iVec) = -p/2;
                d(iVec) = (p/2) * u(n,iVec - 1) + (1-p) * u(n,iVec) + (p/2) * u
                    (n,iVec + 1);
                a(nx)   = -p;
                b(nx)   = 1 + p;
                d(nx)   = p * u(n, nx-1) + (1-p)*u(n, nx);

                %Updates the Next Column with the Calculated Temperature
                %Values
                u(n+1,:) = tdm(a,b,c,d);
            end
        end

    %Returns an Error Message if an Appropriate Method is not Entered
    otherwise
        error (['Undefined method: ' method]);
end
```

tdm.m

```
function x = tdm(a,b,c,d)
%  Function to DetermineTri-diagonal Matrix Solution to Determine the
%  Next Temperature Values for the Backward Differencing and
%  Crank-Nicholson Methods
    n = length(b);

    % Eliminate a terms
    for i = 2:n
        factor = a(i) / b(i-1);
        b(i) = b(i) - factor * c(i-1);
        d(i) = d(i) - factor * d(i-1);
    end

    x(n) = d(n) / b(n);
```

```
    % Loop backwards to find other x values by back-substitution
    for i = n-1:-1:1
        x(i) = (d(i) - c(i) * x(i+1)) / b(i);
    end
```

investMethodsTime.m

```
function [finalTimeStep] = investMethodsTime(tempData, timeData)
%Function to Determine the Stability and Accuracy of the Four Methods
%through Determining the Inner Surface Temperature at Time = 4000s
%against the Timestep

%Input Arguments:
%tempData: Array containing initial temperature data
%timeData: Array containing initial time data

%Return Arguments:
%finalTimeStep - Array containing the Largest Possible Timestep for each of
%the Four Methods before the Desired Accuracy is Compromised


    %Initialises the Variables
    nx = 21;
    thick = 0.05;
    tmax = 4000;

    % numbers of timesteps
    nt = 41:20:1001;

    % caculate all timestep sizes
    dt = tmax./(nt-1);

    % preallocate result vectors for efficiency
    uf = zeros(size(nt));
    ub = zeros(size(nt));
    ud = zeros(size(nt));
    uc = zeros(size(nt));

    % Run simulations for each nt value for all four methods
    for i = 1:length(nt)
        disp (['nt = ' num2str(nt(i)) ', dt = ' num2str(dt(i)) ' s'])

        %Determines the Temperature at the Inner Surface Boundary for the
        %Forward Differencing Method
        [~, ~, u] = calctemp(tmax, nt(i), thick, nx, 'forward', timeData,
            tempData);
        uf(i) = u(end, nx);

        %Determines the Temperature at the Inner Surface Boundary for the
        %Backward Differencing Method
        [~, ~, u] = calctemp(tmax, nt(i), thick, nx, 'backward', timeData,
            tempData);
        ub(i) = u(end, nx);

        %Determines the Temperature at the Inner Surface Boundary for the
        %Dufort-Frankel Method
```

```
        [~, ~, u] = calctemp(tmax, nt(i), thick, nx, 'dufort-frankel', timeData
            , tempData);
        ud(i) = u(end, nx);

        %Determines the Temperature at the Inner Surface Boundary for the
        %Crank Nicholson Method
        [~, ~, u] = calctemp(tmax, nt(i), thick, nx, 'crank-nicholson',
            timeData, tempData);
        uc(i) = u(end, nx);
    end

    %Plots a figure to determine accuracy and stability of each method
    figure(2)
    plot(dt, [uf; ub; ud; uc])
    ylim([0 200])
    title('Investigation in Stability and Accuracy of Different Timesteps')
    xlabel('Time Step')
    ylabel('Temperature(C) on the Inner Surface at t = 4000s')
    legend ('Forward', 'Backward', 'Dufort-Frankel', 'Crank-Nicholson')

    %Compiles the Final Times for the Four Methods and Assigns to a New
    %Variable called 'time' and also extracts the time at the points where
    %the Time Step is at its' Lowest Test Value
    time = [uf;ub;ud;uc];
    startTime = [uf(end), ub(end), ud(end), uc(end)];

    %Sets Accuracy Desired for Retrieving a Suitable Time Step
    accuracy = 0.05;

    %Determines the Range in which the Temperature Values are Allowed to
    %Lie within
    for i = 1:4
        plus5 = startTime(i) + accuracy * startTime(i);
        minus5 = startTime(i) - accuracy * startTime(i);

        plusMinus5Time(2*i - 1) = plus5;
        plusMinus5Time(2*i) = minus5;
    end

    %Initialises the Array to Store the Largest Possible Step for each
    %Method before the Accuracy of the Solution is Compromised
    finalTimeStep = [100, 100, 100, 100];

    %Determines the Time Step allowed for Each Method before the Accuracy
    %of the Solution is Compromised
    for n = 1:4
        for x = 1:length(time(1,:))
            testCol = time(:,x);
            if testCol(n) > plusMinus5Time(2*n - 1) || testCol(n) <
                plusMinus5Time(2*n)
                 finalTimeStep(n) = 100 - 2 * x;
            end
        end
    end
end
```

investMethodsSpace.m

```
function [finalSpaceStep] = investMethodsSpace(tempData, timeData)
%Function to Determine the Stability and Accuracy of the Four Methods
%through Determining the Inner Surface Temperature at Time = 4000s
%against the Number of Spatial Steps

%Input Arguments:
%tempData: Array containing initial temperature data
%timeData: Array containing initial time data

%Return Arguments:
%finalSpaceStep - Array containing the Smallest Number of Space Step for
% each of the Four Methods before the Desired Accuracy is Compromised



    %Initialises the Variables
    nt = 1001;
    thick = 0.05;
    tmax = 4000;

    % numbers of spatial steps
    nx = 2:1:21;

    % caculate all spatial step sizes
    dx = thick./(nx-1);

    % preallocate result vectors for efficiency
    uf = zeros(size(nx));
    ub = zeros(size(nx));
    ud = zeros(size(nx));
    uc = zeros(size(nx));

    % Run simulations for each nx value for all four methods
    for i = 1:length(nx)

        disp (['nx = ' num2str(nx(i)) ', dx = ' num2str(dx(i)) ' m'])

        %Determines the Temperature at the Inner Surface Boundary for the
        %Forward Differencing Method
        [~, ~, u] = calctemp(tmax, nt, thick, nx(i), 'forward', timeData,
            tempData);
        uf(i) = u(nt,end);

        %Determines the Temperature at the Inner Surface Boundary for the
        %Backward Differencing Method
        [~, ~, u] = calctemp(tmax, nt, thick, nx(i), 'backward', timeData,
            tempData);
        ub(i) = u(nt, end);

        %Determines the Temperature at the Inner Surface Boundary for the
        %Dufort-Frankel Method
        [~, ~, u] = calctemp(tmax, nt, thick, nx(i), 'dufort-frankel', timeData
            , tempData);
        ud(i) = u(nt, end);
```

```
        %Determines the Temperature at the Inner Surface Boundary for the
        %Crank Nicholson Method
        [~, ~, u] = calctemp(tmax, nt, thick, nx(i), 'crank-nicholson',
            timeData, tempData);
        uc(i) = u(nt, end);
    end

    % Plots a figure to determine accuracy and stability of each method
    figure(3)
    plot(dx, [uf; ub; ud; uc])
    ylim([0 200])
    title('Investigation in Stability and Accuracy of Different Spatial Steps')
    xlabel('Number of Space Steps')
    ylabel('Temperature (C) on the Inner Surface at t = 4000s')
    legend ('Forward', 'Backward', 'Dufort-Frankel', 'Crank-Nicholson')

    %Compiles the Final Distances for the Four Methods and Assigns to a New
    %Variable called 'space' and also extracts the temperature at the points
    %where the Number of Space Step is at its' Highest Test Value
    space = [uf;ub;ud;uc];
    startSpace = [uf(end), ub(end), ud(end), uc(end)];

    %Sets Accuracy Desired for Retrieving a Suitable Number of Space Steps
    accuracy = 0.001;

    %Determines the Range in which the Temperature Values are Allowed to
    %Lie within
    for i = 1:4
        plus5 = startSpace(i) + accuracy * startSpace(i);
        minus5 = startSpace(i) - accuracy * startSpace(i);

        plusMinus5Space(2*i - 1) = plus5;
        plusMinus5Space(2*i) = minus5;
    end

    %Initialises the Array to Store the Smallest Possible Number of Space
    %Step for each Method before the Accuracy of the Solution is Compromised
    finalSpaceStep = [21,21,21,21];

    %Determines the Smallest Number of Space Steps allowed for Each Method
    %before the Accuracy of the Solution is Compromised
    for n = 1:4
        for x = 1:length(space(1,:))
            testCol = space(:,x);
            if testCol(n) > plusMinus5Space(2*n - 1) || testCol(n) <
               plusMinus5Space(2*n)
                finalSpaceStep(n) = x+1;
            end
        end
    end
end
```

detBestMethod.m

```
function [bestMethod, index] = detBestMethod(finalTimeStep, finalSpaceStep)
```

```
%Function to conduct a simple analysis to determine the best method

%Input Arguments:
%finalTimeStep - stores the final time step found from the
%investMethodsTime function
%finalSpaceStep - stores the final space step found from the
%investMethodsSpace function

%Return Arguments:
%bestMethod - stores the best method of the four numerical methods
%index - stores the index value of the four methods used throughout the
%main code

    %Traverses through each of the four methods determining the difference
    %between the time and space steps, as time step is to be maximised and
    %space step is minimised ideally
    for i = 1:4
        finalMethod(i) = finalTimeStep(i) - finalSpaceStep(i);
    end

    %Determines which method has the highest final value
    [value, index] = max(finalMethod);

    switch index
        case 1
            bestMethod = 'forward';
        case 2
            bestMethod = 'backward';
        case 3
            bestMethod = 'dufort-frankel';
        case 4
            bestMethod = 'crank-nicholson';
    end
end
```

detTileThickness.m

```
function [n] = detTileThickness(nt, nx,bestMethod, timeData, tempData)

%Function for determining the smallest tile thickness without any
%damage occurring to the tile

%Input Arguments:
%nt - number of timesteps
%nx - number of spatial steps
%bestMethod - stores the best method found from the analysis
%timeData - time points from the data provided (s)
%tempData - temperature points from the data provided

%Return Arguments:
%n - final tile thickness (m)

    %Initialises the Required Thickness as 0
    tileThickness = 0;

    %Traverses through the Tile Thicknesses to be Tested and Determines a Value
```

```
    %that is Appropriate
    for n = 0.12:-0.01:0.02
        %Plots a Graph of Temperature at the Inner Boundary against Time for 10
        %Different Tile Thicknesses
        [x, t, u] = calctemp(4000, nt, n, nx, bestMethod, timeData, tempData);
        figure(4)
        hold on
        plot(t,u(:,end))
        title('Plot of Temperature at Inner Surface against Time with Suitable
           TimeStep and Spatial Step')
        xlabel('Time(s)')
        ylabel('Temperature(C) on the Inner Surface ')
        legend('0.12', '0.11', '0.10', '0.09', '0.08', '0.07', '0.06', '0.05',
           '0.04', '0.03','0.02')

        %Determines the First Tile Thickness at which the Maximum Temperature
        %Experienced by the Inner Surface is less than the Maximum Temperature
        %the Material of the Space Shuttle can Experience
        if max(u(:,end)) > 150 && tileThickness == 0
            tileThickness = n + 0.01;
        end
    end

    n = tileThickness;
end
```

8 Appendix B - 1D Code GUI

main.m

```
function main()

% Function to load a GUI enabling user to select and simulate the heat flow
    through tiles at different locations

    % Create a figure
    fig = figure('Position', [500, 400, 800, 600], 'MenuBar', 'none', 'ToolBar'
        , 'none', 'NumberTitle', 'off', 'Name', 'Temperature variation through a
         space shuttle tile 1D');

    % List of initial temperature conditions and images for each tile
    % location
    options = {'Tile 468', 'Tile 480', 'Tile 502', 'Tile 590', 'Tile 597', '
       Tile 711', 'Tile 730', 'Tile 850'};
    images = {'temp468R.jpg', 'temp480R.jpg', 'temp502R.jpg', 'temp590R.jpg', '
       temp597R.jpg', 'temp711R.jpg','temp730R.jpg', 'temp850R.jpg'}; % Paths
       to image files

    % Create listbox for tile locations
    listbox = uicontrol('Style', 'listbox', 'Position', [350, 445, 100, 110], '
       String', options, 'Callback', @listboxCallback);

    % Create axes for displaying images
    axesHandle = axes('Position', [0.02, 0.50, 0.4, 0.6], 'Title', 'Boundary
       Conditions');
    axesHandle2 = axes('Position', [0.09, 0.055, 0.85, 0.6]);

    % Simulation button
    button = uicontrol('Style', 'pushbutton', 'String', 'Simulate', 'Position',
         [350, 410, 100, 30], 'Callback', @buttonCallback)

    % Load 468 tile as inital tile & display parameters
    img = imread(images{1});
    [x,t,u,nt, nx, n, bestMethod] = simulation('temp468R.jpg');
    imshow(img, 'Parent', axesHandle);
    surf(x,t,u,'Parent', axesHandle2)
    xlabel(axesHandle2, 'Distance(m)')
    ylabel(axesHandle2, 'Time(s)')
    zlabel(axesHandle2, 'Temperature(c)')

    % Create lables for parameters
    textLabelBC = uicontrol(fig, 'Style', 'text', 'Position', [32, 550, 300,
       30], 'String', 'External boundary conditions', 'HorizontalAlignment', '
       center', 'FontSize', 12)
    textLabelFG = uicontrol(fig, 'Style', 'text', 'Position', [64, 350, 100,
       30], 'String', 'Simulation:', 'HorizontalAlignment', 'center', 'FontSize
       ', 12)
    textLabelnx = uicontrol(fig, 'Style', 'text', 'Position', [470, 525, 150,
       30], 'String', 'no. Spatial steps:', 'HorizontalAlignment', 'center', '
       FontSize', 12)
    textLabelnt = uicontrol(fig, 'Style', 'text', 'Position', [476, 495, 150,
       30], 'String', 'no. Time steps:', 'HorizontalAlignment', 'center', '
       FontSize', 12)
```

```
    textLabeln = uicontrol(fig, 'Style', 'text', 'Position', [466, 465, 150,
        30], 'String', 'Tile thickness (m):' , 'HorizontalAlignment', 'center',
        'FontSize', 12)
    textLabelBM = uicontrol(fig, 'Style', 'text', 'Position', [483, 435, 150,
        30], 'String', 'Best method:', 'HorizontalAlignment', 'center', '
        FontSize', 12)

    % Create output boxes for parameters
    outputFieldnx = uicontrol(fig, 'Style', 'edit', 'Position', [610, 535, 80,
        20], 'HorizontalAlignment', 'left', 'Enable', 'inactive');
    outputFieldnt = uicontrol(fig, 'Style', 'edit', 'Position', [610, 505, 80,
        20], 'HorizontalAlignment', 'left', 'Enable', 'inactive');
    outputFieldn = uicontrol(fig, 'Style', 'edit', 'Position', [610, 475, 80,
        20], 'HorizontalAlignment', 'left', 'Enable', 'inactive');
    outputFieldBM = uicontrol(fig, 'Style', 'edit', 'Position', [610, 445, 80,
        20], 'HorizontalAlignment', 'left', 'Enable', 'inactive');

    % Fill output boxes
    outputFieldnx.String = sprintf(num2str(nx))
    outputFieldnt.String = sprintf(num2str(nt))
    outputFieldn.String = sprintf(num2str(n))
    outputFieldBM.String = sprintf(num2str(bestMethod))

    % Callback function for listbox
    function listboxCallback(~, ~)
        % Get selected index
        selectedIndex = listbox.Value;
        % Get selected option
        selectedOption = options{selectedIndex};
        % Display selected option
        disp(['Selected Option: ', selectedOption]);
        % Load and display corresponding image
        img = imread(images{selectedIndex});
        imshow(img, 'Parent', axesHandle);
    end

    % Call back function for button
    function buttonCallback(~, ~)
        selectedIndex = listbox.Value;
        selectedOption = images{selectedIndex};
        [x, t, u, nt, nx, n, bestMethod] = simulation(selectedOption);
        surf(x,t,u,'Parent', axesHandle2)
        xlabel(axesHandle2, 'Distance(m)')
        ylabel(axesHandle2, 'Time(s)')
        zlabel(axesHandle2, 'Temperature(c)')

        % Output variables ot main window
        outputFieldnx.String = sprintf(num2str(nx))
        outputFieldnt.String = sprintf(num2str(nt))
        outputFieldn.String = sprintf(num2str(n))
        outputFieldBM.String = sprintf(num2str(bestMethod))
    end


end
```

simulation.m

```
function[x, t, u, nt, nx, n, bestMethod] = simulation(selectedOption)

% Function - simulation: Performs numerical analysis to simulate heat flow
% through a space shuttle tile. Determines appropriate spatial and time
% steps, tile thickness, and method
%
% Input arguments:
% selectedOption - Selected image containing tile inital locations
%
% Ouput arguments:
% x      - distance vector (m)
% t      - time vector (s)
% u      - temperature matrix (C or K)
% nt     - number of timesteps
% nx     - number of spatial steps
% n      - Tile thickness (m)
% bestMethod - Best numerical method for simulation




% Retrieves the time and temperature data points
[origin, TRC, time, temperature] = ImgScan(selectedOption);
[tempData, timeData] = DataScale(origin, TRC, time, temperature);

%Determines the temperature at each distance for each time step for the 4
%different methods
[x1, t1, u1] = calctemp(4000, 1001, 0.05, 21, 'forward', timeData, tempData);
[x2, t2, u2] = calctemp(4000, 501, 0.05, 21, 'dufort-frankel', timeData,
   tempData);
[x3, t3, u3] = calctemp(4000, 501, 0.05, 21, 'backward', timeData, tempData);
[x4, t4, u4] = calctemp(4000, 501, 0.05, 21, 'crank-nicholson', timeData,
   tempData);

%Investigates the Stability and Accuracy of the Four Methods
[finalTimeStep] = investMethodsTime(tempData, timeData);
[finalSpaceStep] = investMethodsSpace(tempData, timeData);

%Determines which of the Four Methods is the Most Accurate
[bestMethod, index] = detBestMethod(finalTimeStep, finalSpaceStep);

%Determines the Values of dt and nt that are within the Required Accuracy
%whilst also Saving Computational Power
nx = finalSpaceStep(index);
dt = finalTimeStep(index);
nt = round((4000 / dt) + 1);

%Determines the Final Thickness of the Tile that will not affect the Tile
n = detTileThickness(nt, nx,bestMethod, timeData, tempData);

%Plots the Final Graph
[x, t, u] = calctemp(4000, nt, n, nx, bestMethod, timeData, tempData);
figure(5)
surf(x,t,u)
title('Final Graph, Interactive')
```

```
xlabel('Distance(m)')
ylabel('Time(s)')
zlabel('Temperature(C)')
end
```

9 Appendix C - 2D Code

main.m

```
% Retrieves the time and temperature data points
[origin, TRC, time, temperature] = ImgScan('temp730R.jpg');
[tempData, timeData] = DataScale(origin, TRC, time, temperature);

%Initialises Tile Width and Number of Spatial Steps across the Width
tileWidth = 0.2;
ny = 5;

%Simultes Forward Method
[x1, y1, t1, u1, values1] = calctemp2d(4000, 2001, 0.05, 21, 'forward',
    timeData, tempData, tileWidth, ny);
%animateGraph(x1, y1, t1, u1)

%Simultes Dufort-Frankel Method
[x2, y2, t2, u2, values2] = calctemp2d(4000, 501, 0.05, 21, 'dufort-frankel',
    timeData, tempData, tileWidth, ny);
% animateGraph(x2, y2, t2, u2)

%Simultes Backward Method
[x3, y3, t3, u3, values3] = calctemp2d(4000, 501, 0.05, 21, 'backward',
    timeData, tempData, tileWidth, ny);
%animateGraph(x3, y3, t3, u3)

%Simultes Crank-Nicholson Method
[x4, y4, t4, u4, values4] = calctemp2d(4000, 501, 0.05, 21, 'crank-nicholson',
    timeData, tempData, tileWidth, ny);
animateGraph(x4, y4, t4, u4)
```

calctemp2d.m

```
function [x, y, t, damageArr, u] = calctemp2d(tmax, nt, xmax, nx, method,
    timeData, tempData, tileWidth, ny)
% Function for modelling temperature in a space shuttle tile
% D N Johnston  14/02/24
%
% Input arguments:
% tmax   - maximum time (s)
% nt     - number of timesteps
% xmax   - total thickness (m)
% nx     - number of spatial steps
% method - solution method ('forward', 'backward' etc)
% timeData - time vector for surface temperatures (s)
% tempData - surface temperature vector (C or K)
% tileWidth - width of tile (m)
% ny - numer of spatial steps in the width of the tile
%
% Return arguments:
% x      - distance vector (m)
% t      - time vector (s)
% u      - temperature matrix (C or K)
%
% For example, to perform a  simulation with 501 time steps
```

```
%   [x, t, u] = calctemp(4000, 501, 0.05, 21, 'forward', timeData, tempData);
%

% Set material properties and derived values (LI-900)
% Obtained from NASA document: Structures and Materials: Space Shuttle Tiles,
   Grades 5-8 - NASA
% Note that we're assuming constant properties.
thermCon = 0.0484; % W/m K; 0.028 BTU/ft/hr/F at 70F and 1 atm
density  = 144;    % kg/m^3; 9 lb/ft^3
specHeat = 628;    % J/kg/K; 0.15 Btu/lb/F

% Initialise everything.
dt = tmax / (nt-1);
t = (0:nt-1) * dt;
dx = xmax / (nx-1);
x = (0:nx-1) * dx;
dy = tileWidth / (ny-1);
y = (0:ny-1) * dy;
u = zeros(ny, nx, nt);

%Determine value of p
thermDiff = thermCon / (density * specHeat);
p = (thermDiff * dt) / (dx ^ 2);

% Use interpolation to get outside temperature at time vector t
% and store it as left-hand boundary vector L.
L = interp1(timeData, tempData, t, "linear", "extrap");

% % set initial conditions equal to boundary temperature at t=0.
u(:,:,1) = L(1);

temp2000 = 0;

% Select method and run simulation.
switch method
    %Simulates the Forward Differencing Method
    case 'forward'

        % Outside boundary condition
        for q =1:ny
            u(q,1,:) = L;
        end

        % set up index vectors
        i  = 2:nx-1;
        im = 1:nx-2;
        ip = 3:nx;
        j  = 2:ny-1;
        jm = 1:ny-2;
        jp = 3:ny;
        %now loop through time
        for n=1:nt-1
            if (n * dt) < 2000
                temp2000 = u(:,1,n+1);
            else
                u(:,1,n+1) = temp2000;
```

```
        end

        % calculate internal values using forward differencing
        u(j, i, n+1) = (1 - 4 * p) * u(j, i, n) +  p * (u(j, im, n) + u(j,
            ip, n) + u(jm, i, n) + u(jp, i, n));

        %Neumann Boundary for north boundary (j=1)
        u(1,i,n+1) = (1-4*p)*u(1,i,n) + p*(u(1,im,n) + u(1,ip,n) + 2*u(2,i,
            n));
        %Neumann Boundary for south boundary (j=ny)
        u(ny,i,n+1) = (1 - 4*p) * u(ny,im,n) + p*(u(ny,im,n) + u(ny,ip,n) +
             2*u(ny-1,i,n));
        %Neumann Boundary for west boundary (i=nx)
        u(j,nx,n+1) = (1-4*p) * u(j,nx,n) + p*(2*u(j,nx-1,n) + u(jm,nx,n) +
             u(jp,nx,n));
        %Neumann Boundary for north-west boundary (j=1 & i=nx)
        u(1,nx,n+1) = (1-4*p) * u(1,nx,n) + p*(2*u(2,nx,n) + 2*u(1,nx-1,n))
            ;
        %Neumann Boundary for south-west boundary (j=ny & i =nx)
        u(ny,nx,n+1) = (1-4*p) * u(ny,nx,n) + p*(2*u(ny-1,nx,n) + 2*u(ny,nx
            -1,n));
    end

    %Analysis of Damage in the Tile
    [damageArr,u] = damageAnalysis(ny,nx,nt,u);

%Simulates the Dufort-Frankel Method
case 'dufort-frankel'

    % Outside boundary condition
    for q =1:ny
        u(q,1,:) = L;
    end

    % set up index vectors
    i  = 2:nx-1;
    im = 1:nx-2;
    ip = 3:nx;
    j  = 2:ny-1;
    jm = 1:ny-2;
    jp = 3:ny;

    %now loop through time
    for n=1:nt-1
        %Data only goes up to 2000s, so only Calculates Values for this
            Range only
        if (n * dt) < 2000
            temp2000 = u(:,1,n+1);
        else
            u(:,1,n+1) = temp2000;
        end

        %Slightly Different Equation is used to Prevent it from Failing
        if n ==1
            % calculate internal values using forward differencing
            u(j, i, n+1) = ((1 - 4 * p) * u(j, i, n) +  2*p * (u(j, im, n)
```

```
                + u(j, ip, n) + u(jm, i, n) + u(jp, i, n))) / (1+4*p);
            %Neumann Boundary for north boundary (j=1)
            u(1, i, n+1) = ((1 - 4 * p) * u(1, i, n) +  2*p * (u(1, im, n)
                + u(1, ip, n) + 2*u(2, i, n))) / (1+4*p);
            %Neumann Boundary for south boundary (j=ny)
            u(ny, i, n+1) = ((1 - 4 * p) * u(ny, i, n) +  2*p * (u(ny, im,
                n) + u(ny, ip, n) + 2*u(ny-1, i, n))) / (1+4*p);
            %Neumann Boundary for west boundary (i=nx)
            u(j, nx, n+1) = ((1 - 4 * p) * u(j, nx, n) +  2*p * (2*u(j, nx
                -1, n) + u(jm, nx, n) + u(jp, nx, n))) / (1+4*p);
            %Neumann Boundary for north-west boundary (j=1 & i=nx)
            u(1, nx, n+1) = ((1 - 4 * p) * u(1, nx, n) +  2*p * (2*u(1, nx
                -1, n) + 2*u(2, nx, n))) / (1+4*p);
            %Neumann Boundary for south-west boundary (j=ny & i =nx)
            u(ny, nx, n+1) = ((1 - 4 * p) * u(ny, nx, n) +  2*p * (2*u(ny,
                nx-1, n) + 2*u(ny-1, nx, n))) / (1+4*p);
        else
            % calculate internal values using forward differencing
            u(j, i, n+1) = ((1 - 4 * p) * u(j, i, n-1) +  2*p * (u(j, im, n
                ) + u(j, ip, n) + u(jm, i, n) + u(jp, i, n))) / (1+4*p);
            %Neumann Boundary for north boundary (j=1)
            u(1, i, n+1) = ((1 - 4 * p) * u(1, i, n-1) +  2*p * (u(1, im, n
                ) + u(1, ip, n) + 2*u(2, i, n))) / (1+4*p);
            %Neumann Boundary for south boundary (j=ny)
            u(ny, i, n+1) = ((1 - 4 * p) * u(ny, i, n-1) +  2*p * (u(ny, im
                , n) + u(ny, ip, n) + 2*u(ny-1, i, n))) / (1+4*p);
            %Neumann Boundary for west boundary (i=nx)
            u(j, nx, n+1) = ((1 - 4 * p) * u(j, nx, n-1) +  2*p * (2*u(j,
                nx-1, n) + u(jm, nx, n) + u(jp, nx, n))) / (1+4*p);
            %Neumann Boundary for north-west boundary (j=1 & i=nx)
            u(1, nx, n+1) = ((1 - 4 * p) * u(1, nx, n-1) +  2*p * (2*u(1,
                nx-1, n) + 2*u(2, nx, n))) / (1+4*p);
            %Neumann Boundary for south-west boundary (j=ny & i =nx)
            u(ny, nx, n+1) = ((1 - 4 * p) * u(ny, nx, n-1) +  2*p * (2*u(ny
                , nx-1, n) + 2*u(ny-1, nx, n))) / (1+4*p);
        end
    end

    %Analysis of Damage in the Tile
    [damageArr,u] = damageAnalysis(ny,nx,nt,u);

%Simulates the Backwards Differencing Method
case 'backward'

    % Outside boundary condition
    for q =1:ny
        u(q,1,:) = L;
    end

    maxiterations = 100;
    tolerance = 1.e-4;

    %now loop through time
    for n=1:nt-1
        %Data only goes up to 2000s, so only Calculates Values for this
            Range only
```

```
        if (n * dt) < 2000
            temp2000 = u(:,1,n+1);
        else
            u(:,1,n+1) = temp2000;
        end

        % calculate internal values iteratively using Gauss-Seidel
        % Starting values are equal to old values
        u(2:ny-1, 2:nx-1, n+1) = u(2:ny-1, 2:nx-1, n);

        for iteration = 1:maxiterations
            change = 0;
            for i=2:nx

                for j=1:ny
                    %Neumann Boundary for north boundary (j=1)
                    if j == 1
                        jm = 2;
                    else
                        jm = j -1;
                    end

                    %Neumann Boundary for south boundary (j=ny)
                    if j == ny
                        jp = ny - 1;
                    else
                        jp = j + 1;
                    end

                    %Neumann Boundary for west boundary (i=nx)
                    if i == nx
                        ip = nx - 1;
                    else
                        ip = i + 1;
                    end

                    uold = u(j, i, n+1);
                    % calculate internal values using forward differencing
                    u(j, i, n+1) = ((u(j, i, n) + p * (u(jm, i, n+1) + u(jp
                        , i, n+1) + u(j, i-1, n+1) + u(j, ip, n+1)))/(1+4*p)
                       );
                    change = change + abs(u(j, i, n+1) - uold);
                end
            end

            if change < tolerance
                break
            end
        end
    end

    %Analysis of Damage in the Tile
    [damageArr,u] = damageAnalysis(ny,nx,nt,u);

%Simulates the Crank-Nicholson Method
case 'crank-nicholson'
```

```
% Outside boundary condition
for q =1:ny
    u(q,1,:) = L;
end

maxiterations = 100;
tolerance = 1.e-4;

%now loop through time
for n=1:nt-1
    %Data only goes up to 2000s, so only Calculates Values for this
       Range only
    if (n * dt) < 2000
        temp2000 = u(:,1,n+1);
    else
        u(:,1,n+1) = temp2000;
    end

    % calculate internal values iteratively using Gauss-Seidel
    % Starting values are equal to old values
    u(2:ny-1, 2:nx-1, n+1) = u(2:ny-1, 2:nx-1, n);

    for iteration = 1:maxiterations
        change = 0;
        for i=2:nx

            for j=1:ny
                %Neumann Boundary for north boundary (j=1)
                if j == 1
                    jm = 2;
                else
                    jm = j -1;
                end

                %Neumann Boundary for south boundary (j=ny)
                if j == ny
                    jp = ny - 1;
                else
                    jp = j + 1;
                end

                %Neumann Boundary for west boundary (i=nx)
                if i == nx
                    ip = nx - 1;
                else
                    ip = i + 1;
                end

                uold = u(j, i, n+1);
                % calculate internal values using forward differencing
                u(j, i, n+1) = ((1-2*p) * u(j,i,n) + (p/2) * (u(j,i-1,n
                   ) + u(j,ip,n) + u(jm,i,n) + u(jp,i,n) + u(j,i-1,n+1)
                    + u(j,ip,n+1) + u(jm,i,n+1) + u(jp,i,n+1))) / (1+2*
                   p);
                change = change + abs(u(j, i, n+1) - uold);
```

```
                    end
                end

                if change < tolerance
                    break
                end
            end
        end

        %Analysis of Damage in the Tile
        [damageArr,u] = damageAnalysis(ny,nx,nt,u);


    %Returns an Error Message if an Appropriate Method is not Entered
    otherwise
        error (['Undefined method: ' method]);
end
```

damageAnalysis.m

```
function [damageArr,u] = damageAnalysis(ny,nx,nt,u)

%Function to model the damage in the tile
%Input Arguments:
% ny - number of spatial steps in the thickness of the tile
% nx - number of spatial steps in the width of the tile
% nt - number of time steps
% u - temperatre at each subsection of the tile

%Return arguements:
% damageArr - Details the areas in which the tile is damaged, used to
% animate the graph
% u - stores the temperature values at each section of the tile

    damageArr = zeros(ny,nx,nt);

    %Assigns the temperates at time = 0 into the damage array
    damageArr(:,:,1) = u(:,:,1);
    %Determining whether the section of the tile is damaged by checking
    %whether it exceeds the maximum allowable temperature or if the tile
    %subsection has already been damaged
    for n = 2:nt
        for a = 1:nx
            for b = 1:ny
                damageArr(b,a,n) = u(b,a,n);
                if u(b,a,n) > 660 || isnan(damageArr(b,a,n-1))
                    u(b,a,n) = u(b,1,n);
                    damageArr(b,a,n) = NaN;
                end
            end
        end
     end
```

animateGraph.m

```
function [] = animateGraph(x, y, t, u)

%Function to animate the 2d graphs to plot a contour plot of width of tile
%against tile thickness against temperature in that section of the tile
%with varying time
%
% Input Arguments:
% x - Tile thickness (m)
% y - Width of tile (m)
% t - time(s)
% u - temperature at the section of the tile (C)

    [X, Y] = meshgrid(x, y);

    % Create a figure
    figure;

    % Loop for animation
    for i = 1:length(t)
        % Update the surface plot
        if i == 1
            U = u(:,:,i);
            % If first iteration, create the surface plot
            surf(Y,X, U);
            xlim([0 0.05])   % Limit graph axes
            colormap('jet');
            colorbar;
            xlabel('Thickness of Tile (m)');
            ylabel('Width of Tile (m)');
            zlabel('Temperature(C)');
            title(sprintf('Surface Plot at t = %.2f', t(i)));  % Example: set
                title with current time
            % Customize the plot appearance as needed
        else
            U = u(:,:,i);
            % For subsequent iterations, update the surface plot
            h = surf(X, Y, U);
            xlim([0 0.05])        % Limit graph axes
            title(sprintf('Surface Plot at t = %.2f', t(i)));
            xlabel('Thickness of Tile (m)');
            ylabel('Width of Tile (m)');
            zlabel('Temperature(C)');

            shading interp
            % set y axis range
            zlim([-0 1300])
            % set colour map range
            caxis ([0 1300]);
            colorbar
         end

        % Update the figure
        drawnow;

        % Pause for a short duration to control animation speed
        pause(0.001);  % Adjust as needed
```

```
end
```

10 Appendix D - 2D Code GUI

```
    function main()

% Function to load a GUI enabling user to select and simulate the heat flow and
    damage through tiles at different locations as they burn up.

    % Create a figure
    fig = figure('Position', [500, 400, 800, 600], 'MenuBar', 'none', 'ToolBar'
       , 'none', 'NumberTitle', 'off', 'Name', 'Temperature variation through a
        space shuttle tile 2D');

    % List of initial temperature conditions and images for each tile
    % location
    options = {'Tile 468', 'Tile 480', 'Tile 502', 'Tile 590', 'Tile 597', '
       Tile 711', 'Tile 730', 'Tile 850'};
    images = {'temp468R.jpg', 'temp480R.jpg', 'temp502R.jpg', 'temp590R.jpg', '
       temp597R.jpg', 'temp711R.jpg','temp730R.jpg', 'temp850R.jpg'}; % Paths
       to image files

    % Create listbox for tile locations
    listbox = uicontrol('Style', 'listbox', 'Position', [350, 445, 100, 110], '
       String', options, 'Callback', @listboxCallback);

    % Create axes for displaying images
    axesHandle = axes('Position', [0.02, 0.50, 0.4, 0.6], 'Title', 'Boundary
       Conditions');
    axesHandle2 = axes('Position', [0.09, 0.075, 0.85, 0.55]);

    % List of methods
    optionsM = {'forward', 'dufort-frankel', 'backward', 'crank-nicholson' };

    % Create listbox for methods
    listboxM = uicontrol('Style', 'listbox', 'Position', [470, 445, 100, 110],
       'String', optionsM, 'Callback', @listboxCallbackM);

    % Simulation button
    button = uicontrol('Style', 'pushbutton', 'String', 'Simulate', 'Position',
        [600, 480, 100, 30], 'Callback', @buttonCallback)

    % Load 468 tile as inital tile & display parameters
    img = imread(images{1});
    [x,y,t,u] = simulation('temp468R.jpg', 'forward');
    imshow(img, 'Parent', axesHandle);

    % Create lables for parameters
    textLabelBC = uicontrol(fig, 'Style', 'text', 'Position', [32, 550, 300,
       30], 'String', 'External boundary conditions', 'HorizontalAlignment', '
       center', 'FontSize', 12)
    textLabelFG = uicontrol(fig, 'Style', 'text', 'Position', [64, 350, 100,
       30], 'String', 'Simulation:', 'HorizontalAlignment', 'center', 'FontSize
       ', 12)


    % Callback function for listbox
    function listboxCallback(~, ~)
```

```
        % Get selected index
        selectedIndex = listbox.Value;
        % Get selected option
        selectedOption = options{selectedIndex};
        % Display selected option
        disp(['Selected Option: ', selectedOption]);
        % Load and display corresponding image
        img = imread(images{selectedIndex});
        imshow(img, 'Parent', axesHandle);
    end

% Callback function for listboxM
    function listboxCallbackM(~, ~)
        % Get selected index
        selectedIndex = listboxM.Value;
        % Get selected option
        method = optionsM{selectedIndex};
        % Display selected option
        disp(['Selected Option: ', method]);
    end


    % Call back function for button
    function buttonCallback(~, ~)
        selectedIndex = listboxM.Value;
        method = optionsM{selectedIndex};
        selectedIndex = listbox.Value;
        selectedOption = images{selectedIndex};
        [x, y, t, u] = simulation(selectedOption, method);
        [X, Y] = meshgrid(x, y);

    % Loop for animation
    for i = 1:length(t)
        % Update the surface plot
        if i == 1
            U = u(:,:,i);
            % If first iteration, create the surface plot
            surf(Y,X,U, 'Parent', axesHandle2);
            xlim([0 0.05])   % Limit graph axes
            colormap('jet');
            colorbar;
            xlabel('Thickness of Tile (m)');
            ylabel('Width of Tile (m)');
            zlabel('Temperature(C)');
            title(sprintf('Surface Plot at t = %.2f', t(i)));  % Example: set
               title with current time
            % Customize the plot appearance as needed
        else
            U = u(:,:,i);
            % For subsequent iterations, update the surface plot
            h = surf(X, Y, U, 'Parent', axesHandle2);
            xlim([0 0.05])        % Limit graph axes
            title(sprintf('Surface Plot at t = %.2f', t(i)));
            xlabel('Thickness of Tile (m)');
            ylabel('Width of Tile (m)');
            zlabel('Temperature(C)');
```

```
            shading interp
            % set y axis range
            zlim([-0 1300])
            % set colour map range
            caxis ([0 1300]);
            colorbar
        end

        % Update the figure
        drawnow;

        % Pause for a short duration to control animation speed
        pause(0.001);  % Adjust as needed
    end

    end



end
```

simulation.m

```
function[x,y,t,u] = simulation(selectedOption, method)

% Function - simulation: Simulates the damage across a tile using different
% numerical methods
%
% Input arguments:
% method - The method to be used in the numerical analysis
%
% Output arguments
% x      - distance vector (m)
% t      - time vector (s)
% u      - temperature matrix (C or K)
% y      - width of tile(m)



% Retrieves the time and temperature data points
[origin, TRC, time, temperature] = ImgScan(selectedOption);
[tempData, timeData] = DataScale(origin, TRC, time, temperature);

%Initialises Tile Width and Number of Spatial Steps across the Width
tileWidth = 0.2;
ny = 20;

%Simultes Forward Method
[x, y, t, u] = calctemp2d(4000, 2001, 0.05, 21, method, timeData, tempData,
   tileWidth, ny);


end
```