Modelling the Temperature Through the Thickness of a Heat Resistant Tile During Re-entry into the Earth's Atmosphere

William Powell

Student Code: wfp21

Modelling Techniques 2 (ME20021)

Word Count: 1920

Referencing Type: IEEE

April 2021

Abstract

This paper examines four numerical methods in the modelling of temperature dissipation through Shuttle Columbia's heat resistant tiles, during atmospheric re-entry. The Crank-Nicolson method proved to be most accurate at large timesteps and highly stable. Through investigation, an optimised timestep and spatial step were determined to be 66.7s and 6.25mm respectively to maintain a $\pm 0.5K$ tolerance. To prevent the aluminium annealing, from temperatures above 450K, a minimum tile thickness of 57.7mm is required at the shuttle's wing.

Contents

1 Introduction

1.1 Objectives

1. Estimate the inner temperature of the tile during re-entry assuming zero heat flow.
2. Determine a suitable tile thickness for the space shuttle at different locations.
3. Evaluate and select a suitable method, timestep and spatial step for an efficient, accurate and stable solution.'
4. Enhancement of the modelling through the design of a GUI; an automated data extraction tool; a 2 dimensional representation and an automated tile thickness estimator.

1.2 Context of investigation

The 2003 Space Shuttle Columbia suffered "severe overheating and disintegrated during atmospheric entry", an explanation from a NASA case study after the event [1]. 81 seconds into launch, a piece of foam from the external fuel tank broke off and hit the heat resistant panel on the left wing at a relative speed of 500mph. Most Space Shuttle missions are subject to foam impacts, usually contacting the more fragile LI-900 silica tiles, however in this instance it hit the carbon-carbon (RCC) tiles on the bleeding edge of the wing. This caused significant damage, since despite having superior heat resistance, the panels were more brittle than the LI-900 tiles.

Evidently, material choices are crucial for the success of space missions. Thorough experimentation and modelling of shuttle tiles is required to determine materials with suitable properties, such as high yield stress and low thermal conductivity, whilst minimising mass and cost.

2 Background

2.1 Thermomechanical Effects and the Heat Equation

Upon re-entry into the upper regions of the atmosphere, a hypersonic shock wave is formed, resulting in compression at the shock and stagnated boundary layer and friction due to the air's viscosity. At these high velocities, the 'nose shock closely envelops the surface' of the space shuttle resulting in 'thermomechanical effects that can be devastating' [2].

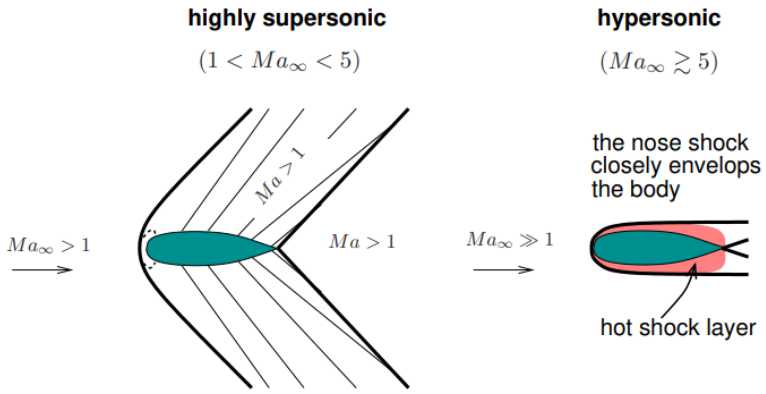

Figure 1: Flow streamlines around a supersonic and hypersonic body [2]

The majority of the shuttle's kinetic energy will be converted to heating of the gas in the shock wave bow, a fraction of which will be conducted to the tiles. However this still results in a significant thermal load.

The temperature dissipation can be modelled with the one dimensional heat equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \qquad\qquad \dot{u} = \alpha \nabla^2 u \tag{1}$$

Both equations are congruent, where u is the temperature distribution in the one dimensional plane, x, with respect to time, t, and α is the thermal diffusivity given by:

$$\alpha = \frac{k}{\rho C_p} \tag{2}$$

where k is thermal conductivity, c_p is the specific heat capacity and ρ is the density of the material.

2.2 Partial Differential Equations

Equation (1) is a Parabolic Partial Derivative Equation (PDE), involves an unknown function dependent on two or more variables and partial derivatives with respect to the independent variables, in this case, time and thickness [3]. To solve, two boundary conditions are required - a function in the time domain and one in the dimension domain, x.

It is sometimes possible to solve PDEs analytically, including the heat equation, using the separation of variables technique to break it down to a pair of ODEs, however this requires a great deal of computation, and numerical approximations can often be calculated with negligible error.

2.3 Numerical Approximation Methods

To model the variation of temperature through a thickness with respect to time, values of temperature, u, are calculated at discrete points in space, x, and time t, rather than a 'mathematical function which can be evaluated at all points in the domain '[4].

The most general method is Forward Differencing, where the new discrete value can be estimated by the linear, first order derivative. By applying Taylor's series for the heat equation's partial time derivative, the next value of u at $t + \Delta t$ can be determined.

$$u(x, t + \Delta t) = u(x, t) + \Delta t \frac{\partial u}{\partial t}(x, t) + \frac{\Delta t}{2} \frac{\partial^2 u}{\partial t^2}(x, t) + \frac{\Delta t}{6} \frac{\partial^3 u}{\partial t^3}(x, t) + \ \cdots \tag{3}$$

This can be rearranged and simplified to:

$$\frac{\partial u}{\partial t}(x, t) \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \tag{4}$$

This simplification neglects high order terms and in this case the equation is of first order and is equivalent to Euler's method.

Similarly, Taylor series can also be applied to the PDE with respect to spatial derivative, x, rearranged and simplified to first order.

$$\frac{\partial^2 u}{\partial x^2}(x, t) \approx \frac{u(x - \Delta x, t) - 2u(x, t) + u(x + \Delta x, t)}{\Delta x^2} \tag{5}$$

Equating the partial derivatives in equation (3) and (4) and rearranging:

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

where i and n denotes the discrete values of x and t respectively. This is a explicit method since each new value u_i^{n+1} is calculated from the current state of the system.

The implementation of this function requires a timestep, Δt and spatial step, Δx. Increasing these values reduces the computation power, but can result in loss of accuracy and sometimes stability.

In addition to the Forward method, there are a number of other numerical approximation methods which are illustrated in the table below.

Numerical Approximation Methods			
Method	Description	Accuracy	Stability
Forward Differencing	• Explicit • Most simplistic PDE numerical method.	$O(\Delta t, \Delta x^2)$	Conditionally stable - unstable for large timesteps when $0 < p < 0.5$, where $p = \frac{\alpha \Delta t}{\Delta x^2}$
DuFort-Frankel	• Explicit • Modified from the Leap-frog equation. • Multi-step method	$O(\Delta t^2, \Delta x^2, (\frac{\Delta t}{\Delta x})^2)$	Unconditionally Stable, however $\frac{\partial^2 u}{\partial t^2}$ can cause oscillations. Heat equation does not have this term but wave equation does. Jagged/Spikey in nature.
Backward Differencing Method	• Implicit • Requires matrix manipulation, through tri-diagonal matrix solving	$O(\Delta t, \Delta x^2)$	Conditionally stable – more stable than forward differencing.
Crank-Nicolson	• Implicit • Requires matrix manipulation, through tri-diagonal matrix solving	$O(\Delta t^2, \Delta x^2)$	Unconditionally Stable, but can result in small overshoots

Table 1: Properties of different Numerical Approximation Methods

Truncation errors are due to the neglection of high order terms. First order error methods has the property that by halving the timestep Δt, the error is approximately halved. N^{th} order errors are denoted by the Big O notation - $O(\Delta t^N)$. For example, the Forward method has first order accuracy in time and second order accuracy in space, therefore denoted as $O(\Delta t, \Delta x^2)$. Both explicit methods result in a constant inner surface temperature if timestep is less than spatial step since a perturbation can only propagate through by one spatial step per timestep.

3 Modelling Procedure

For complete MATLAB scripts, please view Appendix 7.4.

Descriptive summary of each script:

1. **Shuttle** - One dimensional temperature dissipation model implemented with the four numerical methods illustrated in Table 1.
2. **Shuttle 2D Function** - Two dimensional model, length of tile included in solution with forward and backward method.
3. **Timestep Investigation** and **Spatial Step Investigation**- Investigation to determine optimal numerical method and optimal timestep and spatial step.
4. **Tri-diagonal Matrix** - Tri-diagonal matrix solver, required for implicit methods.
5. **Automated Image Extractor** - To extract data from NASA's Aeroheating Flight Experiment. [5]
6. **Minimum Tile Thickness** - Calculates the minimum tile thickness required to prevent a temperature overshoot using the shooting method.
7. **GUI** - Interface for user-friendly representation of all scripts above.

4 Results

4.1 Timestep and Spatial Step Investigation

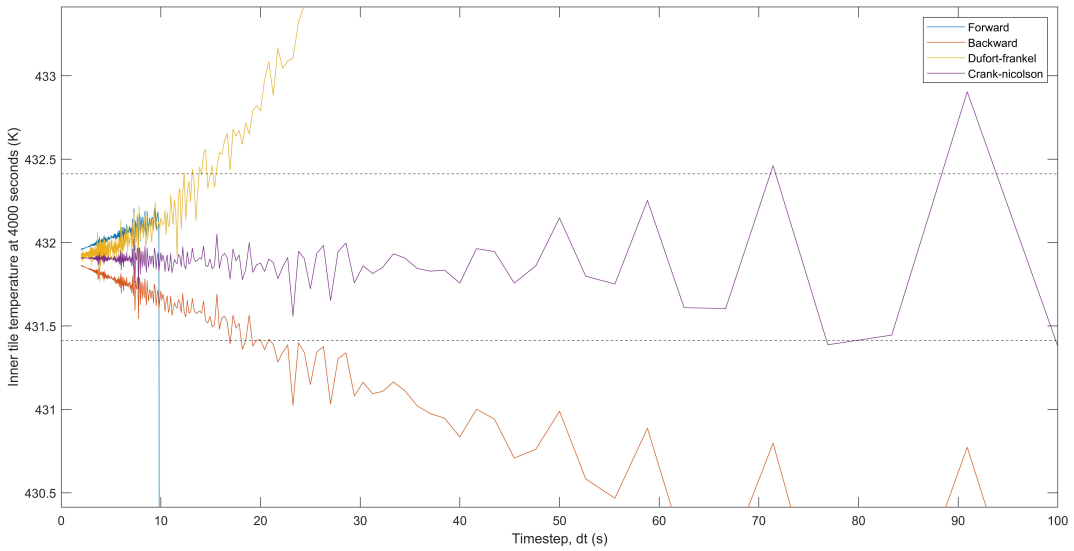

Figure 2: Variation of Temperature with Timestep across Different Numerical Methods

The final temperature at the inside of the tile converges to an average value of 431.9 Kelvin. The dotted lines illustrate the tolerance for accuracy, in this instance ±0.5 Kelvin. With increasing timestep, the temperature for all methods diverge with increasing error. First order methods, Forward and Backward, diverge linearly, whilst second-order, Dufort-Frankel and crank-nicolson, diverge parabolically. It is evident that Forward method requires the smallest timestep to remain accurate, then Dufort-Frankel, Backward and lastly Crank-Nicolson. All methods demonstrate oscillatory behaviour. The Forward method collapses at timesteps greater than 9.8 seconds.

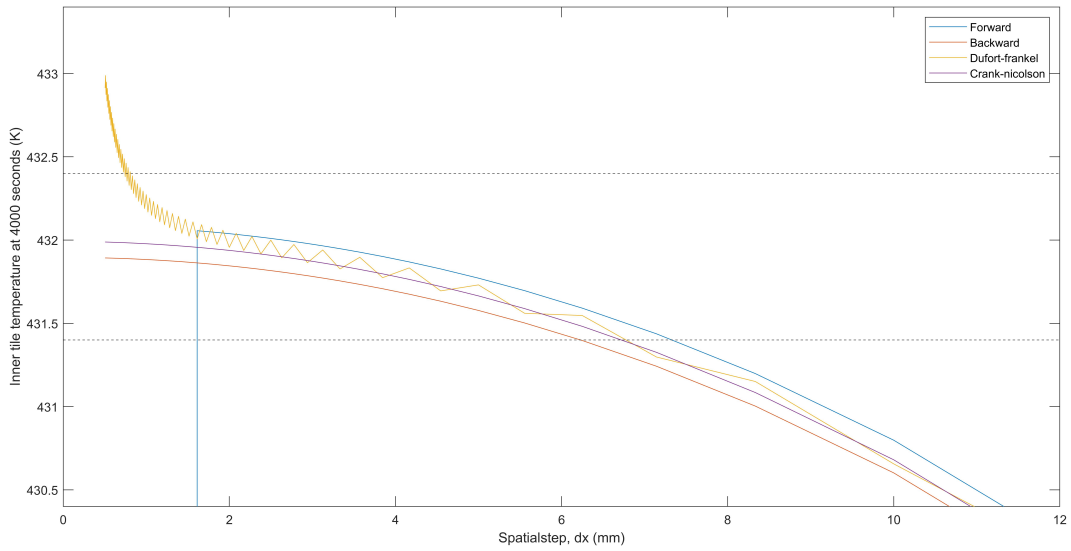

Figure 3: Variation of Temperature with Spatial Step across Different Numerical Methods

With the same converging temperature as Figure 3, increasing the spatial step causes a parabolic decline in temperature and increasingly larger error for all methods. The spatial step has increasing inaccuracies in both directions for forward, since it collapses at low spatial steps, and Dufort-Frankel methods, where it deviates at low spatial step. Conversely, all methods suffer with increasing timestep after approximately 2mm. The range of spatial step for -0.5K accuracy is between 5.8 to 7.0mm. At spatial steps lower than 1.61mm, the stability of the forward method collapses and increases to larger values. Dufort-Frankel shows an oscillatory behaviour, and deviates increasingly at low timesteps.

4.2 One dimensional Heat Equation Numerical Approximation

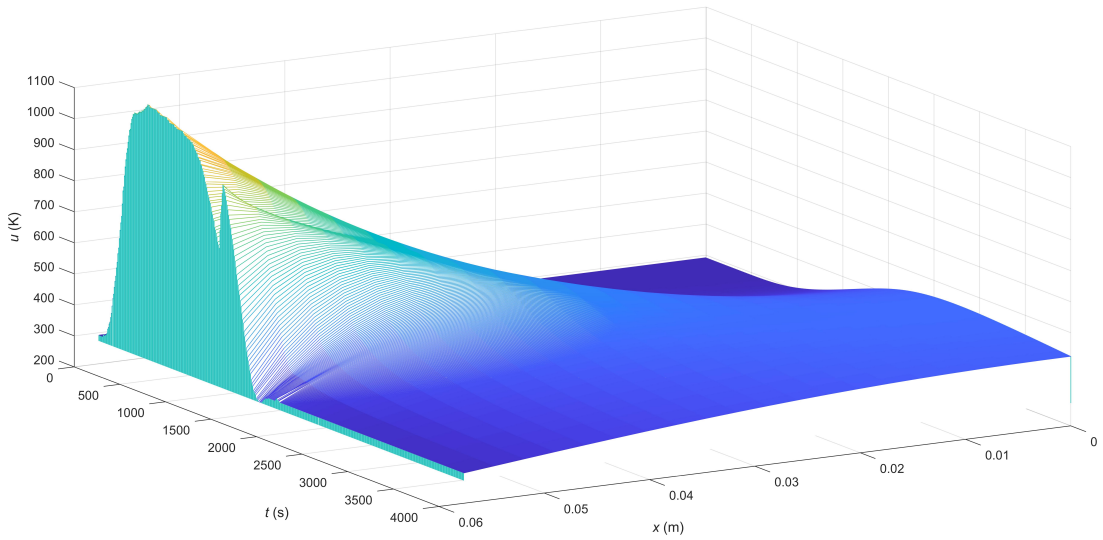

Figure 4: One-Dimensional Plot of Temperature Dissipation through the Thickness of Tile '597'.

With a tile of thickness of 57.7mm, calculated in X, the boundary condition temperature is illustrated in green and is extracted data from NASA's Aeroheating Flight Experiment [5]. The maximum outside temperature is 1096K at the boundary condition, see Appendix 7.2.

Through the LI900 material, temperature dissipates to a maximum of 450K at 2770 seconds at the inside of the tile, a delay of approximately 1675 seconds from the outside maximum, see Appendix 7.2. The GUI shows a two dimensional animated plot, with both thickness and width parameters, see Appendix 7.3.

5 Discussion

5.1 Evaluation of the Stability and Accuracy of Numerical Methods

The Forward method shows to be unstable in both the time and spatial domain. This is coherent with Table 1, where there is stability for timesteps below 9.81 seconds, due to the condition $0 < p < 0.5$, and similarly 1.59mm for spatial step, see Appendix 7.1 for derivation. Due to this characteristic, the Forward method is undesirable. l

The Dufort-Frankel method is stable across the domain range selected in this investigation and mathematically stable for all values. However, the method does show to be inaccurate and extremely oscillatory. It is the second method after Forward to exceed the accuracy tolerance for timestep and inaccurate at small spatial steps due to the $O(\frac{\Delta t}{\Delta x})^2$ term. The timestep must therefore be significantly smaller than spatial step. Careful choice of step values must taken to reduce error.

The backward method shows to be very stable along the step value ranges in this investigation. Due to its implicit form, it has significantly more stability than the Forward method. However, it only has first order accuracy, and is unideal for precise, large calculations such as CFD. It is more desirable than Forward, although will require more computational power.

The Crank-Nicolson method proves to be stable and highly accurate at large timesteps. Due to its implicit form and 2^{nd} order accuracy, it maintained the $\pm 0.5K$ tolerance until an approximate timestep and spatial step of 66.7 seconds and 6.25mm respectively. Thus proving it's durability for computationally taxing problems, whilst remaining stable and accurate. Crank-Nicolson should therefore be selected as the best method for solving this heat problem.

Since it has a significantly larger timestep than the other methods, the timestep should be maximised before the optimisation of the spatial step. Hence, the maximum timestep of 66.7 seconds, leads to a maximum spatial step of 5.62mm to remain in the tolerance range. This change leads to a reduction in computation time of 15%.

5.2 Material Choice and Tile Thickness

From NASA's Orbiter Thermal Protection System Report [6], the shuttle's aluminium airframe, the material on the inside of the shuttle, must "not exceed this 350-degree [Fahrenheit] limit" where the aluminium "anneals, or softens".

The majority of the tiles on Columbia used LI-900, "made from 99.9 percent pure silica glass fibers, and consist of 94 percent by volume of air." This highly insulated material, with low density and thermal conductivity, leaves it very suitable for minimising heat flow. Furthermore it can be "heated to 2,200 F and plunged into cold water without damage". A higher strength LI-2200 material was placed in high-stress areas, although "not without an undesirable weight penalty". NASA then developed a "fibrous refractory composite insulation", FRCI-12, which was considerably lighter than the LI-2200, despite having a lower thermal shock resistance. Many areas of the vehicle were replaced with FRCI-12, however it is unclear which locations these were in, therefore it should be assumed that LI-900 is used.

To calculate the tile thickness required near Columbia's wing, the minThickness script, implemented using the shooting method, was employed with a targeted inside maximum temperature of 350F. This leads to a minimum thickness of 57.7mm for tile '597', as shown in the GUI figure.

5.3 Assumptions made in the model

To calculate thickness of the tiles, a one dimensional model was used whereby the length and width were disregarded. This assumes that the temperature on each tile, extracted from NASA's flight experiment [5], is constant. Stated in 'Space Shuttle Orbiter Systems' by KSC [7], each HSRI, High-Temperature Reusable Surface Insulation, was sized 6 x 6 inches, and the temperature is unlikely to vary considerably in this area.

The two dimensional model assumed constant temperature along the external length of the tile. If a temperature gradient is found this boundary condition can be implemented. This would then demonstrate how a variation in temperature along the length of the external tile, influences the inner tile temperature.

The modelling is currently using a Neumann insulated boundary condition, where the normal derivative is given. However, this assumes instantaneous temperature change at the surface of the tile. In reality, there is a delay in heat transfer due to radiation.

6 Conclusions

The Crank-Nicolson method shows to be most suitable for temperature dissipation modelling due to its second order and implicit nature that allows for high accuracy at large timesteps and is very stable. Due to the small tile area and thus near constant temperature profile, the one dimensional model should still be an accurate representation but could be improved with a 3 dimensional solution.

Further consideration should be taken for the manufacturing tolerances of the material and a safety factor to prevent another disaster like the Columbia mission. Stress analysis must be simulated to chose the most suitable material illustrated in 5.2. By implementing a Robin boundary condition using the radiation equation, this will no longer assume instantaneous temperature change at the surface and lead to a more accurate result.

It is evident that a great deal of care is required when selecting numerical methods and step size to ensure model accuracy and consistent repeatably for the surplus of 2000 heat resistive tiles [7].

References

[1] Rocha, 2011. *Accident Case Study of Organizational Silence Communication Breakdown: Shuttle Columbia, Mission STSA107* [Online] Available at: `https://www.nasa.gov/pdf/553084main_Case_Study_Silence_Breakdown_Columbia_Rocha.pdf` [Accessed 9th April 2021]

[2] Urzay, 2020. *The physical characteristics of hypersonic flows* [Online] Available at: `https://web.stanford.edu/~jurzay/hypersonicsCh2_Urzay.pdf` [Accessed 9th April 2021]

[3] WolframMathWorld, 2016. *Partial Differential Equation* [Online] Available at: https://mathworld.wolfram.com/PartialDifferentialEquation.html [Accessed 9th April 2021]

[4] Johnston N., 2020. *Engineering Applications and Numerical Solution of Partial Differential Equations* [Booklet, Offline]

[5] Blanchard et Al., 2001. *Infrared Sensing Aeroheating Flight Experiment: STS-96 Flight Results* [Online] Available at: https://www.cs.odu.edu/ mln/ltrs-pdfs/NASA-aiaa-2001-0352.pdf [Accessed 9th April 2021]

[6] Kennedy Space Center, 2006. *Orbiter Thermal Protection System* [Online] Available at: `https://www.nasa.gov/centers/kennedy/pdf/167473main_TPS-06rev.pdf`[Accessed 9th April 2021]

[7] Kennedy Space Center, 2000. *Space Shuttle Orbiter Systems - Thermal Protection System* [Online] Available at: `https://science.ksc.nasa.gov/shuttle/technology/sts-newsref/sts_sys.html#sts-hrsi`[Accessed 9th April 2021]

7 Appendix

7.1 Forward Stability for Model Conditions

Forward differencing is conditionally stable when $0 < p < 0.5$ where $p = \frac{\alpha \Delta t}{\Delta x^2} < 0.5$ and $\alpha = \frac{k}{\rho C_p}$. The values used in the timestep and spatial step investigations, where $dx = 2.5mm$ and $dt = 4s$ respectively.

$$0 < \Delta t < \frac{0.5\Delta x^2}{\alpha}$$
$$0 < \Delta t < \frac{0.5 * 0.0025^2}{\frac{0.141}{351*1261}}$$
$$0 < \Delta t < 9.81s$$

and for spatial step stability:

$$\Delta x > \sqrt{\frac{\alpha \Delta t}{0.5}}$$
$$\Delta x > \sqrt{\frac{\frac{0.141}{351*1261} * 4}{0.5}}$$
$$\Delta x > 1.56mm$$

7.2 Additional Results

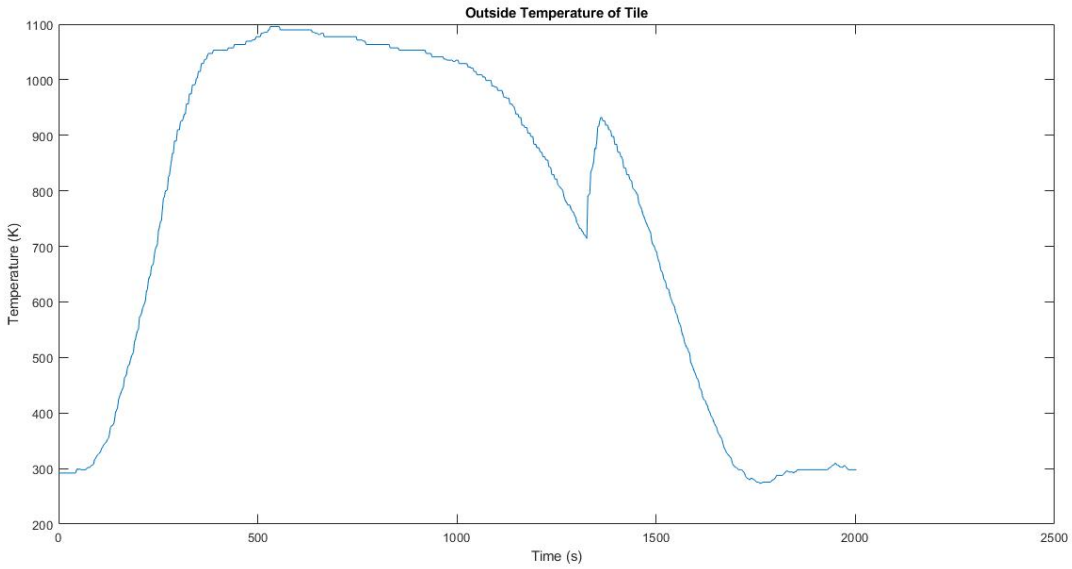

Figure 5: External Temperature at Surface of Tile '597'

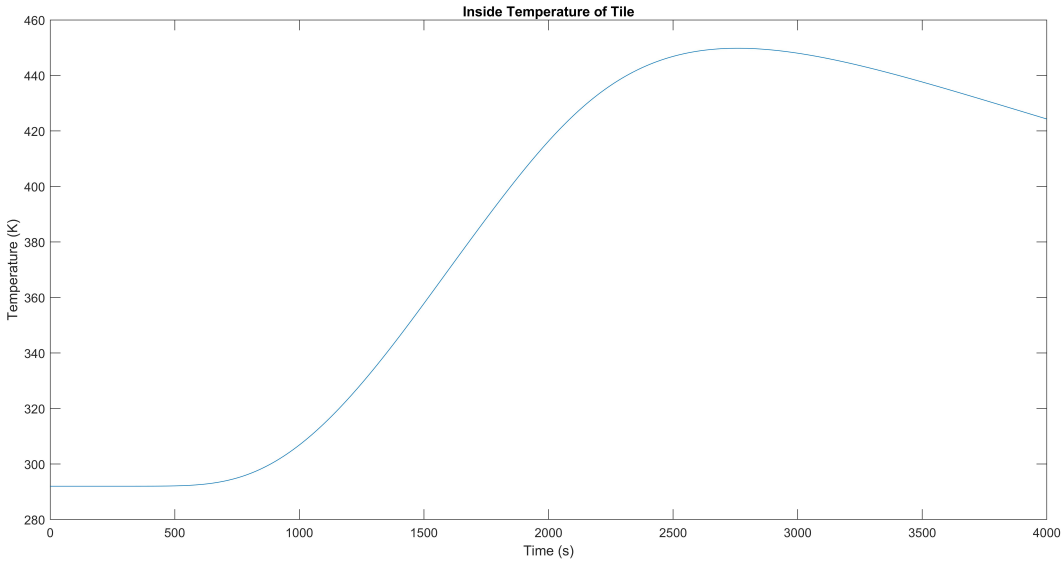

Figure 6: Inner Temperature of Tile '597' with optimised thickness to prevent the tile overshooting the 450K limit

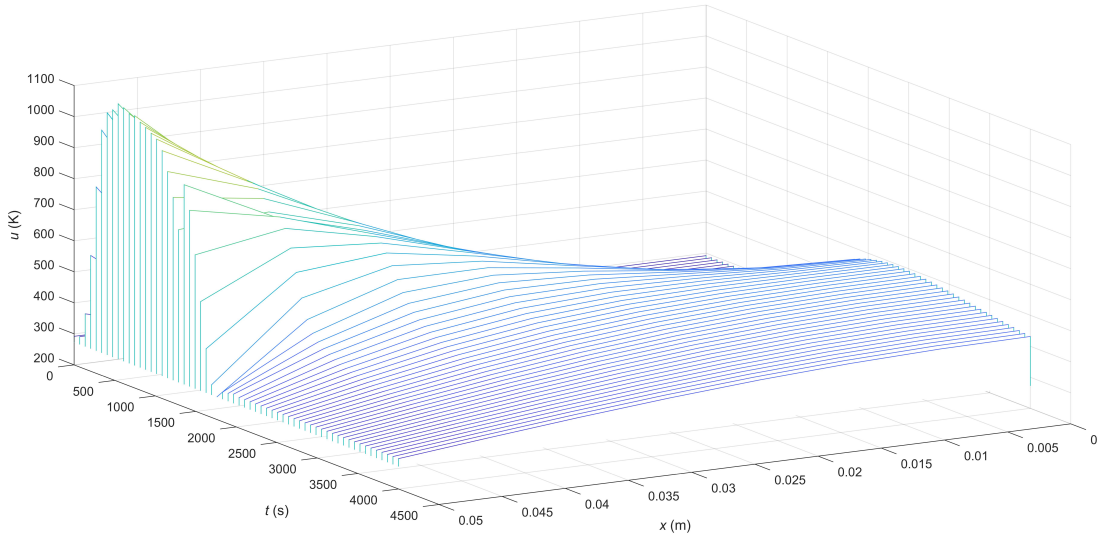

Figure 7: One-Dimensional Plot of Temperature Dissipation through the Thickness of Tile '597' with optimised step size values

7.3 Graphical User Interface

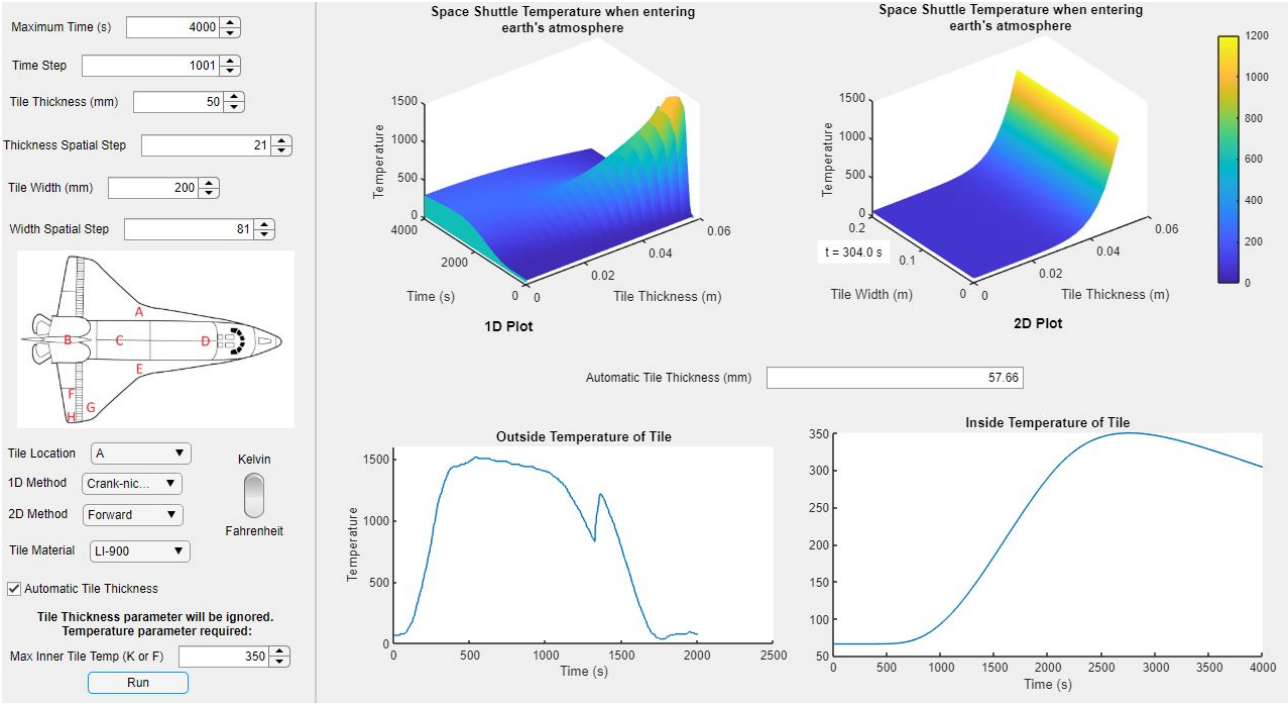

Figure 8: User Interface with automated tile thickness selected

7.4 MATLAB Scripts

7.4.1 Shuttle Function

```
function [x, t, u] = shuttle(tmax, nt, xmax, nx, method, varargin)

% Function for modelling temperature in a space shuttle tile

% W Powell  06/04/21
% Modified from code written by Dr Nigel Johnston

% Required input arguments:
% tmax   - maximum time
% nt     - number of timesteps
% xmax   - total thickness
% nx     - number of spatial steps in x axis
% method - solution method ('forward', 'backward' etc)

% Optional input arguments:
% doPlot - true to plot graph; false to suppress graph.
% tempUnitK - true if Kelvin, false if Fahrenheit
% tileMat - tile material
% fileName - graph image for data extraction
% GUI - plot to GUI; false to plot to window
% axes1, axes2, axes3 - handle for axes (only for GUI)

% Return arguments:
% x      - distance vector
% t      - time vector
% u      - temperature matrix
%
% For example, to perform a  simulation with 501 time steps
%   [x, t, u] = shuttle(4000, 501, 0.05, 21, 'forward', true);



% optional parameters
switch nargin
    case {1,2,3,4,5}
        doPlot = true;
        tempUnitK = true;
        tileMat = 'li900';
        fileName = '597';
        GUI = false;
    case 6
        doPlot = varargin{1};
        tempUnitK = true;
        tileMat = 'li900';
        fileName = '597';
        GUI = false;
    case 7
        [doPlot, tempUnitK] = varargin{1:2};
        tileMat = 'li900';
        fileName = '597';
        GUI = false;
    case 8
```

```
function [x, t, u] = shuttle(tmax, nt, xmax, nx, method, varargin)

% Function for modelling temperature in a space shuttle tile

% W Powell  06/04/21
% Modified from code written by Dr Nigel Johnston

% Required input arguments:
% tmax   - maximum time
% nt     - number of timesteps
% xmax   - total thickness
% nx     - number of spatial steps in x axis
% method - solution method ('forward', 'backward' etc)

% Optional input arguments:
% doPlot - true to plot graph; false to suppress graph.
% tempUnitK - true if Kelvin, false if Fahrenheit
% tileMat - tile material
% fileName - graph image for data extraction
% GUI - plot to GUI; false to plot to window
% axes1, axes2, axes3 - handle for axes (only for GUI)

% Return arguments:
% x      - distance vector
% t      - time vector
% u      - temperature matrix
%
% For example, to perform a  simulation with 501 time steps
%   [x, t, u] = shuttle(4000, 501, 0.05, 21, 'forward', true);



% optional parameters
switch nargin
    case {1,2,3,4,5}
        doPlot = true;
        tempUnitK = true;
        tileMat = 'li900';
        fileName = '597';
        GUI = false;
    case 6
        doPlot = varargin{1};
        tempUnitK = true;
        tileMat = 'li900';
        fileName = '597';
        GUI = false;
    case 7
        [doPlot, tempUnitK] = varargin{1:2};
        tileMat = 'li900';
        fileName = '597';
        GUI = false;
    case 8
```

```
        [doPlot, tempUnitK, tileMat] = varargin{1:3};
        fileName = '597';
        GUI = false;
    case 9
        [doPlot, tempUnitK, tileMat, fileName] = varargin{1:4};
        GUI = false;
    case 10
        [doPlot, tempUnitK, tileMat, fileName, GUI] = varargin{1:5};
    case 13
        [doPlot, tempUnitK, tileMat, fileName, GUI, axes1, axes2,
 axes3 ] = varargin{1:8};
end

% Set tile properties
switch tileMat
    case 'li900'
        thermCon = 0.0577; % W/(m K)
        density  = t;   % 9 lb/ft^3
        specHeat = 1261;  % ~0.3 Btu/lb/F at 500F
    case 'li2200'
        thermCon = 0.0577; % W/(m K)
        density  = 352.5;   % 22 lb/ft^3
        specHeat = 1261;  % ~0.3 Btu/lb/F at 500F
    case 'frcp12'
        thermCon = 0.0577; % W/(m K)
        density  = 192.2;   % 12 lb/ft^3
        specHeat = 1261;
        % ~0.3 Btu/lb/F at 500F
end

% loads data from graph image using image extraction function
[tempK, tempF, time] = imgExtraction(fileName);

% Initialise everything.
dt = tmax / (nt-1); % sets time step size
t = (0:nt-1) * dt; % time vector
dx = xmax / (nx-1); % sets thickness step size
x = (0:nx-1) * dx; % thickness of tile as vector
u = zeros(nt, nx); % pre allocates space for U (state vector for
 temperature)
alpha = thermCon/(density * specHeat); % thermal diffusivity
p = (alpha * dt)/(dx^2); % non dimensional timestep

% Use interpolation to get outside temperature at times t
% and store it as right-hand boundary R.
if tempUnitK
    tempUnit = 'K';
    R = interp1(time, tempK, t, 'linear', tempK(end));
else
    tempUnit = 'F';
    R = interp1(time, tempF, t, 'linear', tempF(end));
end
% set initial conditions equal to boundary temperature at t=0.
u(1, :) = R(1);
```

```
% Main timestepping loop.
for n = 1:nt-1

    % Select method.
    switch method
        case 'forward'
            % neuman boundary conditions
            i = 1:nx-1;
            im = [2 1:nx-2];
            ip = 2:nx;
            % set boundary conditions
            u(n+1,nx) = R(n+1);
            u(n+1,i) = (1 - 2*p) * u(n,i) + p * (u(n,im) + u(n,ip));
        case 'dufort-frankel'
            % neuman boundary conditions
            i=1:nx-1;
            im = [2 1:nx-2];
            ip = 2:nx;
            % set boundary conditions
            u(n+1,1) = R(1);
            u(n+1,nx) = R(n+1);
            % set index for 'old' point
            if n == 1
                nminus1 = 1; % at first timestep, old point doesn't
 exist as n-1 = 0.
                             % Use value at timestep 1 instead.
            else
                nminus1 = n-1; % after first timestep, proceed
 normally.
            end
            % calculate internal values using Leapfrog method
            u(n+1,i)= (u(nminus1,i) * (1-2*p) + 2 * p * (u(n,im) +
 u(n,ip)))/(1+2*p);
        case 'backward'
            ivec = 2:nx-1; % set up index vector
            % set boundary conditions
            l = R(1);
            r = R(n+1);
            ivec = 2:nx-1; % set up index vector
            %calculate internal values using backward differencing
            b(1)    = 1 + 2*p;
            c(1)    = -2*p;
            d(1)    = u(n,1);
            a(ivec) = -p;
            b(ivec) = 1 + 2*p;
            c(ivec) = -p;
            d(ivec) = u(n,ivec);
            a(nx)   = 0;
            b(nx)   = 1;
            d(nx)   = r;
            u(n+1,:) = triDiag(a,b,c,d); % calls tri diagonal matrix
 function
        case 'crank-nicolson'
```

```
            ivec = 2:nx-1; % set up index vector
            % set boundary conditions
            l = R(1);
            r = R(n+1);
            ivec = 2:nx-1; % set up index vector
            % calculate internal values using crank nicolson
            b(1)    = 1 + 2*p;
            c(1)    = -2*p;
            d(1)    = u(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)   = 0;
            b(nx)   = 1;
            d(nx)   = r;
            u(n+1,:) = triDiag(a,b,c,d); % calls tri diagonal matrix
 function
        otherwise
            error (['Undefined method: ' method])
            return
    end
end

InsideTemp = u(:, 1);

% plots 3d graph of tile temperature with respect to tile thickness,
 with time domain
if doPlot
    if GUI == false % if axes location is undefined plot normally with
 window
        % plots inside and outside temperature of tile in the time
 domain
        if tempUnitK
            figure(1);
            plot(time, tempK);
            xlabel('Time (s)');
            ylabel('Temperature (K)');
            title('Outside Temperature of Tile');
            figure(2);
            plot(t, InsideTemp');
            xlabel('Time (s)');
            ylabel('Temperature (K)');
            title('Inside Temperature of Tile');
            xlim([0, tmax]);

        else
            figure(1);
            plot(time, tempF);
            xlabel('Time (s)');
            ylabel('Temperature (F)');
            title('Outside Temperature of Tile');
            figure(2);
```

```
            plot(t, InsideTemp');
            xlabel('Time (s)');
            ylabel('Temperature (F)');
            title('Inside Temperature of Tile');
            xlim([0, tmax]);
        end
        figure(3);
        waterfall(x,t,u); % waterfall 3d graph plot
    else % if using GUI, plot in GUI not window
        % plots inside and outside temperature of tile in the time
 domain
        if tempUnitK
            plot(axes1, time, tempK);
            plot(axes2, t, InsideTemp');
        else
            plot(axes1, time, tempF);
            plot(axes2, t, InsideTemp');
        end
        waterfall(axes3,x,t,u); % waterfall 3d graph plot
    end
    view(150,30) % viewing angle of plot
    xlabel('\itx\rm (m)')
    ylabel('\itt\rm (s)')
    zlabel(['\itu\rm (' tempUnit ')'])

end
end
```

Published with MATLAB® R2020b

7.4.2 Shuttle 2D Function

```
function [x, t, u, uEnd] = shuttle2d(tmax, nt, xmax, nx, ymax, ny,
 method, varargin)

% Function for modelling temperature in a space shuttle tile
% W Powell  06/04/2021
% Co-Author: F Berteau

% Required input arguments:
% tmax    - maximum time
% nt      - number of timesteps
% xmax    - total thickness of tile
% nx      - number of spatial steps in x axis
% ymax    - width of tile
% ny      - number of spatial steps in y axis
% method - solution method ('forward', 'backward' etc)

% Optional input arguments:
% doPlot - true to plot graph; false to suppress graph.
% tempUnitK - true if Kelvin, false if Fahrenheit
% tileMat - tile material
% fileName - graph image for data extraction
% axes - axes handle to plot to GUI; false to plot to window

% Return arguments:
% x       - distance vector
% t       - time vector
% u       - temperature matrix
%
% For example, to perform a 2d simulation for 1001 time steps:
%   [x, t, u] = shuttle2d(4000, 1001, 0.05, 21, 0.2, 81, 'forward');


% optional parameters
switch nargin
    case {1,2,3,4,5, 6, 7}
        doPlot = true;
        tempUnitK = true;
        tileMat = 'li900';
        fileName = '597';
        axes = false;
    case 8
        doPlot = varargin{1};
        tempUnitK = true;
        tileMat = 'li900';
        fileName = '597';
        axes = false;
    case 9
        [doPlot, tempUnitK] = varargin{1:2};
        tileMat = 'li900';
        fileName = '597';
        axes = false;
    case 10
```

1

```
function [x, t, u, uEnd] = shuttle2d(tmax, nt, xmax, nx, ymax, ny,
 method, varargin)

% Function for modelling temperature in a space shuttle tile
% W Powell  06/04/2021
% Co-Author: F Berteau

% Required input arguments:
% tmax   - maximum time
% nt     - number of timesteps
% xmax   - total thickness of tile
% nx     - number of spatial steps in x axis
% ymax   - width of tile
% ny     - number of spatial steps in y axis
% method - solution method ('forward', 'backward' etc)

% Optional input arguments:
% doPlot - true to plot graph; false to suppress graph.
% tempUnitK - true if Kelvin, false if Fahrenheit
% tileMat - tile material
% fileName - graph image for data extraction
% axes - axes handle to plot to GUI; false to plot to window

% Return arguments:
% x      - distance vector
% t      - time vector
% u      - temperature matrix
%
% For example, to perform a 2d simulation for 1001 time steps:
%   [x, t, u] = shuttle2d(4000, 1001, 0.05, 21, 0.2, 81, 'forward');


% optional parameters
switch nargin
    case {1,2,3,4,5, 6, 7}
        doPlot = true;
        tempUnitK = true;
        tileMat = 'li900';
        fileName = '597';
        axes = false;
    case 8
        doPlot = varargin{1};
        tempUnitK = true;
        tileMat = 'li900';
        fileName = '597';
        axes = false;
    case 9
        [doPlot, tempUnitK] = varargin{1:2};
        tileMat = 'li900';
        fileName = '597';
        axes = false;
    case 10
```

```
        [doPlot, tempUnitK, tileMat] = varargin{1:3};
        fileName = '597';
        axes = false;
    case 11
        [doPlot, tempUnitK, tileMat, fileName] = varargin{1:4};
        axes = false;
    case 12
        [doPlot, tempUnitK, tileMat, fileName, axes] = varargin{1:5};
end

% Set tile properties
switch tileMat
    case 'li900'
        thermCon = 0.0577; % W/(m K)
        density  = 144;   % 9 lb/ft^3
        specHeat = 1261;  % ~0.3 Btu/lb/F at 500F
    case 'li2200'
        thermCon = 0.0577; % W/(m K)
        density  = 352.5;   % 22 lb/ft^3
        specHeat = 1261;  % ~0.3 Btu/lb/F at 500F
    case 'frcp12'
        thermCon = 0.0577; % W/(m K)
        density  = 192.2;   % 12 lb/ft^3
        specHeat = 1261;  % ~0.3 Btu/lb/F at 500F
end

% loads data from graph image using image extraction function
[tempK, tempF, time] = imgExtraction(fileName);

% Initialise everything.
dt = tmax / (nt-1); % sets time step size
t = (0:nt-1) * dt; % time vector
dx = xmax / (nx-1); % sets thickness step size
x = (0:nx-1) * dx; % thickness of tile as vector
dy = ymax / (ny-1); % tile width step size
y = (0:ny-1) * dy; % tile width as vector
u = zeros(ny, nx, nt); % pre allocates space for U (state vector for
 temperature)
alpha = thermCon/(density * specHeat); % thermal diffusivity
p = (alpha * dt)/(dx^2); % non dimensional timestep

% Use interpolation to get outside temperature at times t
% and store it as right-hand boundary R.
if tempUnitK
    R = interp1(time, tempK, t, 'linear', tempK(end));
else
    R = interp1(time, tempF, t, 'linear', tempF(end));
end

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

% plots a animated graph. h is a 'handle' of the graph
if doPlot
```

```
    if axes == false
        figure(3);
        h = surf(x, y, u(:,:,1)); % surf plot
        pbaspect([1 4 1]) % sets the aspect ratio
        %displays the current time of animation
        h2=text(-0.05, 0.18, 280, ' t = 0 s   ', 'BackgroundColor',
[1,1,1]);
        shading interp % shades for clearer graphics
        % set y axis range dependent on Kelvin or Fahrenheit
        if tempUnitK
            tempUnit = 'K';
            zlim([280 1300])
        else
            tempUnit = 'F';
            zlim([44 1880])
        end
        % colour table to translate colour to a temperature value
        caxis ([0 1200]);
        colorbar
        % set axes labels
        xlabel('\itx\rm (m)')
        ylabel('\ity\rm (m)')
        zlabel(['\itu\rm (' tempUnit ')'])
    else
        h = surf(axes, x, y, u(:,:,1));
        set(h, 'EdgeColor', 'interp', 'FaceColor', 'interp'); % shades
for clearer graphics
        h2=text(axes, -0.03, 0.13, 280, ' t = 0 s
', 'BackgroundColor', [1,1,1]);
        % colour table to translate colour to a temperature value
        caxis (axes,[0 1200]);
        colorbar(axes)
    end
end

% Main timestepping loop.
for n = 1:nt-1

    % Select method.
    switch method
        % forward method with Neuman Boundary Condition
        case 'forward'
            % Boundary condition from outside temperature
            u(:,nx,n+1) = R(n+1);

            % Neuman Boundaries
            i = 1:nx-1;
            im = [2 1:nx-2];
            ip = 2:nx;

            j = 2:ny;
            jm = 1:ny-1;
            jp = [3:ny ny-1];
```

```
            % calculates U state vector with respect to thickness and
 width of the tile
            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));
            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));
        % backward method without Neuman Boundary
        case 'backward'
            maxiterations = 100; % timestep
            tolerance = 1.e-4; % for stability
            % Boundary condition from outside temperature
            u(:,nx,:) = R(n+1);
            % calculate internal values iteratively using Gauss-Seidel
            % Starting values are equal to old values
            u(:, :, n+1) = u(:, :, n);

            for iteration = 1:maxiterations
                change = 0;
                for i=2:nx-1
                    for j=2:ny-1
                        uold = u(j, i, n+1);
                        u(j, i, n+1) = ((u(j, i, n) + p * (u(j-1, i, n
+1) + u(j+1, i, n+1)...
                            + u(j, i-1, n+1) + u(j, i+1, n+1)))/
(1+4*p));
                        change = change + abs(u(j, i, n+1) - uold);
                    end
                end
                % exit backward method calculations if unstable
                if change < tolerance
                    break
                end
            end
            %disp(['Time = ' num2str(t(n)) ' s: Iterations = '
 num2str(iteration)]);
        otherwise
            error (['Undefined method: ' method])
            return
    end
    % update graph with new values
    set(h,'ZData', u(:, :, n+1));

    % display current time
    txt = sprintf(' t = %4.1f s ', t(n+1));
    set(h2, 'String', txt)
    % Refresh screen
    drawnow
end
uEnd = u(end);
end
```

Published with MATLAB® R2020b

7.4.3 Timestep Investigation Function

```
function  [bestMethod, maxdt] = timestepInv(tmax, ntMin, ntMax,
 ntIncr, thick, nx, tol)

% [bestMethod, maxTimestep]
% Function for determining the appropriate timestep
% to maintain stability and accuracy, using the numerical
% approximation methods - forward, backward, Du-fort Frankel and
% Crank-Nicolson

% W Powell  06/04/21
% Modified from code written by Dr Nigel Johnston

% Required input arguments:
% tmax    - maximum time
% ntMin, ntMax, ntIncr - minimum, maximum and incrementation of
 timesteps
%                        to test
% thick   - total thickness
% nx      - number of spatial steps in x axis
% tol     - error tolerance for accuracy

% Output arguments:
% bestMethod     - returns best method - method with maximum timestep
%                  whilst maintaining accuracy and stability
% maxdt    - returns maximum timestep for best method, with optimal
%            efficiency whilst remaing in the accepted error tolerance

% For example, to perform function with a max error of ±0.5 Kelvin:
% [bestMethod, maxdt] = timestepInv(4000, 41, 2001, 20, 0.05, 21, 0.5)


i = 0;
% iterate through values of nt for each method and extract final temp
 value
% at boundary
for nt = ntMin : ntIncr : ntMax
    i=i+1;
    dt(i) = tmax/(nt-1);
    disp (['nt = ' num2str(nt) ', dt = ' num2str(dt(i)) ' s']);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'forward', false);
    uf(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'backward', false);
    ub(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'dufort-frankel', false);
    udf(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'crank-nicolson', false);
    ucn(i) = u(end, 1);
end
% calculates average stable value of temperature for all method at
 smallest
% dt value
avgConvergence = (uf(end) + ub(end) + udf(end) + ucn(end))/4;
```

1

```
function  [bestMethod, maxdt] = timestepInv(tmax, ntMin, ntMax,
 ntIncr, thick, nx, tol)

% [bestMethod, maxTimestep]
% Function for determining the appropriate timestep
% to maintain stability and accuracy, using the numerical
% approximation methods - forward, backward, Du-fort Frankel and
% Crank-Nicolson

% W Powell  06/04/21
% Modified from code written by Dr Nigel Johnston

% Required input arguments:
% tmax    - maximum time
% ntMin, ntMax, ntIncr - minimum, maximum and incrementation of
 timesteps
%                         to test
% thick   - total thickness
% nx      - number of spatial steps in x axis
% tol     - error tolerance for accuracy

% Output arguments:
% bestMethod     - returns best method - method with maximum timestep
%                  whilst maintaining accuracy and stability
% maxdt    - returns maximum timestep for best method, with optimal
%            efficiency whilst remaing in the accepted error tolerance

% For example, to perform function with a max error of ±0.5 Kelvin:
% [bestMethod, maxdt] = timestepInv(4000, 41, 2001, 20, 0.05, 21, 0.5)


i = 0;
% iterate through values of nt for each method and extract final temp
 value
% at boundary
for nt = ntMin : ntIncr : ntMax
    i=i+1;
    dt(i) = tmax/(nt-1);
    disp (['nt = ' num2str(nt) ', dt = ' num2str(dt(i)) ' s']);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'forward', false);
    uf(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'backward', false);
    ub(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'dufort-frankel', false);
    udf(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'crank-nicolson', false);
    ucn(i) = u(end, 1);
end
% calculates average stable value of temperature for all method at
 smallest
% dt value
avgConvergence = (uf(end) + ub(end) + udf(end) + ucn(end))/4;
```

```
plot(dt, [uf; ub; udf; ucn])
hold on
yline(avgConvergence + tol, 'k--')
yline(avgConvergence - tol, 'k--')
hold off
ylim([avgConvergence - 3*tol, avgConvergence + 3*tol])
xlabel('Timestep, dt (s)')
ylabel('Inner tile temperature at 4000 seconds (K)')
legend ('Forward', 'Backward', 'Dufort-frankel', 'Crank-nicolson')

maxdt = 0;
i = 0;
methods = {'Forward', 'Backward', 'Dufort-frankel', 'Crank-nicolson'};
z = [uf; ub; udf; ucn];
% iterates through methods and finds method with largest timestep
 whilst
% still being in the tolerance range
for i = 1:length(methods)
    method = methods(i);
    intersectPos = [];
    intersectNeg = [];
    intersectPos = dt(find(z(i,:) > avgConvergence + tol,
 1, 'last')+1);
    intersectNeg = dt(find(z(i,:) < avgConvergence - tol,
 1, 'last')+1);
    if ~isempty(intersectPos) && intersectPos > maxdt
        maxdt = intersectPos;
        bestMethod = method;
    elseif ~isempty(intersectNeg) && intersectNeg > maxdt
        maxdt = intersectNeg;
        bestMethod = method;
    end
end

end
```

Published with MATLAB® R2020b

7.4.4 Spatial Step Investigation Function

```
function  [bestMethod, maxdx] = spatialStepInv(tmax, nxMin, nxMax,
 nxIncr, thick, nt, tol)

% Function for determining the appropriate Spatial Step
% to maintain stability and accuracy, using the numerical
% approximation methods - forward, backward, Du-fort Frankel and
% Crank-Nicolson

% W Powell  06/04/21
% Modified from code written by Dr Nigel Johnston

% Required input arguments:
% tmax   - maximum time
% nxMin, nxMax, nxIncr - minimum, maximum and incrementation of
 timesteps
%                         to test
% thick  - total thickness
% nt     - number of timesteps, use stable value for all methods
%          from timestepInv
% tol    - error tolerance for accuracy

% Output arguments:
% bestMethod    - returns best method - method with maximum timestep
%                 whilst maintaining accuracy and stability
% maxdx   - returns maximum spatial step for best method, with optimal
%           efficiency whilst remaing in the accepted error tolerance

% For example, to perform function with a max error of ±0.5 Kelvin:
%[bestMethod, maxdx] = spatialStepInv(4000, 5, 30, 1, 0.05, 1001, 0.5)


i = 0;
% iterate through values of nx for each method and extract final temp
 value
% at boundary
for nx = nxMin : nxIncr : nxMax
    i=i+1;
    dx(i) = thick / (nx-1);
    disp (['nx = ' num2str(nx) ', dx = ' num2str(dx(i)*1000) 'mm']);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'forward', false);
    uf(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'backward', false);
    ub(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'dufort-frankel', false);
    udf(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'crank-nicolson', false);
    ucn(i) = u(end, 1);
end
% calculates average stable value of temperature for all method at
 smallest
% dt value
% avgConvergence = (uf(end) + ub(end) + udf(end) + ucn(end))/4;
```

1

```
function  [bestMethod, maxdx] = spatialStepInv(tmax, nxMin, nxMax,
 nxIncr, thick, nt, tol)

% Function for determining the appropriate Spatial Step
% to maintain stability and accuracy, using the numerical
% approximation methods - forward, backward, Du-fort Frankel and
% Crank-Nicolson

% W Powell  06/04/21
% Modified from code written by Dr Nigel Johnston

% Required input arguments:
% tmax    - maximum time
% nxMin, nxMax, nxIncr - minimum, maximum and incrementation of
 timesteps
%                        to test
% thick   - total thickness
% nt      - number of timesteps, use stable value for all methods
%           from timestepInv
% tol     - error tolerance for accuracy

% Output arguments:
% bestMethod     - returns best method - method with maximum timestep
%                  whilst maintaining accuracy and stability
% maxdx    - returns maximum spatial step for best method, with optimal
%            efficiency whilst remaing in the accepted error tolerance

% For example, to perform function with a max error of ±0.5 Kelvin:
%[bestMethod, maxdx] = spatialStepInv(4000, 5, 30, 1, 0.05, 1001, 0.5)


i = 0;
% iterate through values of nx for each method and extract final temp
 value
% at boundary
for nx = nxMin : nxIncr : nxMax
    i=i+1;
    dx(i) = thick / (nx-1);
    disp (['nx = ' num2str(nx) ', dx = ' num2str(dx(i)*1000) 'mm']);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'forward', false);
    uf(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'backward', false);
    ub(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'dufort-frankel', false);
    udf(i) = u(end, 1);
    [~, ~, u] = shuttle(tmax, nt, thick, nx, 'crank-nicolson', false);
    ucn(i) = u(end, 1);
end
% calculates average stable value of temperature for all method at
 smallest
% dt value
% avgConvergence = (uf(end) + ub(end) + udf(end) + ucn(end))/4;
```

```
avgConvergence = 431.962935806185;
plot(dx*1000, [uf; ub; udf; ucn])
hold on
yline(avgConvergence + tol, 'k--')
yline(avgConvergence - tol, 'k--')
hold off
ylim([avgConvergence - 3*tol, avgConvergence + 3*tol])
xlabel('Spatialstep, dx (mm)')
ylabel('Inner tile temperature at 4000 seconds (K)')
legend ('Forward', 'Backward', 'Dufort-frankel', 'Crank-nicolson')

maxdx = 0;
i = 0;
methods = {'Forward', 'Backward', 'Dufort-frankel', 'Crank-nicolson'};
z = [uf; ub; udf; ucn];
% iterates through methods and finds method with largest timestep
 whilst
% still being in the tolerance range
% for i = 1:length(methods)
%     method = methods(i);
%     intersectPos = [];
%     intersectNeg = [];
%     intersectPos = dx(find(z(i,:) > avgConvergence + tol, 1,
 'last')+1);
%     intersectNeg = dx(find(z(i,:) < avgConvergence - tol, 1,
 'last')+1);
%     if ~isempty(intersectPos) && intersectPos > maxdx
%         maxdx = intersectPos;
%         bestMethod = method;
%     elseif ~isempty(intersectNeg) && intersectNeg > maxdx
%         maxdx = intersectNeg;
%         bestMethod = method;
%     end
% end

end
```

Published with MATLAB® R2020b

7.4.5 Image Extraction Function

```
function [tempK, tempF, time] = imgExtraction(fileName)
% Function to automatically extract data from graph images of
 temperature
% variation through different tile locations graphs given by
% NASA's Aeroheating Flight Experiment
% (https://www.cs.odu.edu/~mln/ltrs-pdfs/NASA-aiaa-2001-0352.pdf)

% W Powell  06/04/21
% Required input arguments:
% fileName   - image name of tile graph at desired location

% Output arguments:
% tempK    - temperature data in Kelvin
% tempF    - temperature data in Farenheit
% time     - time data

% For example, to perform function for tile 597, with file name
 597.png:
% [tempK, tempF, time] = imgExtraction('597');

% threshold to detect red/black in image
redThreshold = 80;
blackThreshold = 20;

% initial conditions that are no available from image plot
initTempK = 292;
initTempF = 66;
initTime = 0;

img = imread(['ShuttleImgs/' fileName '.png']); % load image from file

imgNoPlot = 255 - round((img(:,:,2) + img(:,:,3))); % removes red from
 image
imgAxes = imgNoPlot > blackThreshold; % convert to binary image of
 Axes

% xAxis and yAxis locator:

yAxis = 0; % yAxis is the column of img where the y axis is located
maxSumColumn = 0; % arbitary high value to ensure sumColumn is greater
 on first column
sumRow = zeros(size(img,1), 1);
prevCell = 1;
yLimCoord = [];

% loops through each cell and detects x axis and y axis by summing
 pixel
% values for column and row respectively
for column = 1:size(imgAxes,2)
    sumColumn = 0; % reset sum of pixels for new column to zero
    for row = 1:size(imgAxes,1)
```

1

```
function [tempK, tempF, time] = imgExtraction(fileName)
% Function to automatically extract data from graph images of
 temperature
% variation through different tile locations graphs given by
% NASA's Aeroheating Flight Experiment
% (https://www.cs.odu.edu/~mln/ltrs-pdfs/NASA-aiaa-2001-0352.pdf)

% W Powell  06/04/21
% Required input arguments:
% fileName   - image name of tile graph at desired location

% Output arguments:
% tempK    - temperature data in Kelvin
% tempF    - temperature data in Farenheit
% time     - time data

% For example, to perform function for tile 597, with file name
 597.png:
% [tempK, tempF, time] = imgExtraction('597');

% threshold to detect red/black in image
redThreshold = 80;
blackThreshold = 20;

% initial conditions that are no available from image plot
initTempK = 292;
initTempF = 66;
initTime = 0;

img = imread(['ShuttleImgs/' fileName '.png']); % load image from file

imgNoPlot = 255 - round((img(:,:,2) + img(:,:,3))); % removes red from
 image
imgAxes = imgNoPlot > blackThreshold; % convert to binary image of
 Axes

% xAxis and yAxis locator:

yAxis = 0; % yAxis is the column of img where the y axis is located
maxSumColumn = 0; % arbitary high value to ensure sumColumn is greater
 on first column
sumRow = zeros(size(img,1), 1);
prevCell = 1;
yLimCoord = [];

% loops through each cell and detects x axis and y axis by summing
 pixel
% values for column and row respectively
for column = 1:size(imgAxes,2)
    sumColumn = 0; % reset sum of pixels for new column to zero
    for row = 1:size(imgAxes,1)
```

```
        cell = double(imgAxes(row,column)); % finds value of current
 cell
        if prevCell == 0 && cell == 1 % store all cells that
 transition from white to black
            yLimCoord(end+1,:) = [row column];
        end
        sumRow(row,1) = sumRow(row,1) + cell;
        sumColumn = sumColumn + cell; % sums all cell values
        prevCell = cell;
    end
    if sumColumn > maxSumColumn % axes are black therefore lowest sum
 of pixels will be at y axis column
        yAxis = column;
        maxSumColumn = sumColumn; % updates new largest summed column
    end

end

[~, xAxis] = max(sumRow); % sets xAxis to maximum pixel sum for the
 row (when row is black due to axes)
prevCell = 0;
xLim = 0;
% locates x axis limit
for column = 1:size(imgAxes,2)
    cell = imgAxes(xAxis,column);
    if prevCell == 1 && cell == 0 % store all cells that transition
 from white to black
            xLim = column - 1;
    end
    prevCell = cell;
end
yLim = yLimCoord(find((yLimCoord(:,2) == yAxis),1),1); % locates y
 limit

% used for graph transformations
setAxesTemp = [yLim, xAxis];
setAxesTime = [xLim, yAxis];

% crop image to axes
imgCrop = img((yLim : xAxis),(yAxis : xLim),:);
% extracts red plot line from image
imgRed = imgCrop(:,:,1) - round((imgCrop(:,:,2) + imgCrop(:,:,3)));
% converts image to logic matrix / binary image
imgBinary = imgRed > redThreshold;
imgBinary = flip(imgBinary); % flips matrix

avgYCoord = zeros(1,size(imgBinary,2)); % pre allocates space for
 vector
% averages all y values in each column, since multiple pixels in each
 column
for i = 1:size(imgBinary,2)
    sumYCoord = 0;
    sumColumn = 0;
    for j = 1:size(imgBinary,1)
```

```
        if imgBinary(j,i) == 1
                sumColumn = sumColumn + 1;
                sumYCoord = sumYCoord + j;
                avgYCoord(1,i) = (sumYCoord / sumColumn);
        end
    end
end

sizeYCoord = size(avgYCoord);
x = 1:sizeYCoord(2); % x axis of image is number of columns

% graph transformations
tempF = ((2000/(xAxis - yLim)) .* (avgYCoord)); % temp data in
 fahrenheit
time = ((2000 / (xLim-yAxis))  .* (x)); % time data in seconds

tempK = (tempF - 32) .* (5/9)+ 273.15; % temp data in kelvin

% initial conditions which are unkown from image
tempK(1) = initTempK;
tempF(1) = initTempF;
tempK(avgYCoord == 0) = initTempK; % set unkown data points at low
 time values
tempF(avgYCoord == 0) = initTempF;
time(1) = initTime;
```

Published with MATLAB® R2020b

7.4.6 Minimum Tile Thickness Function

```
function [thickness] = minThicknessInv(maxTemp, tmax, nt, nx, method,
 tempUnitK, doPlot)

% This function determines the minimum tile thickness required to
 prevent
% an overshoot of an inputted maximum temperature at the inside of the
 tile.

% W Powell  06/04/21

% Required input arguments:
% maxTemp - maximum temperature desired at inside of tile
% tmax    - maximum time
% nt      - number of timesteps
% xmax    - total thickness
% nx      - number of spatial steps in x axis
% method - solution method ('forward', 'backward' etc)
% tempUnitK - true if Kelvin, false if Fahrenheit
% doPlot - true to plot graph; false to suppress graph.

% Output arguments:
% thickness    - minimum tile thickness required to withold from
 maxTemp

% For example, to perform function for a maximim temperatre of 450K:
% [thickness] = minThicknessInv(450, 4000, 501, 21, 'crank-nicolson',
 true, true)


% initial guesses for minimum thickness
guess(1,1) = 0.05;
guess(1,2) = 0.1;

% Desired temperature lower than maximum temperature(K or F)
tempPrecision = 1;

% first guess for shooting method
[~, ~, u] = shuttle(tmax, nt, guess(1,1), nx, method, false);
guess(2,1) = max(u(:,1)); % finds maximum temperature

% second guess
[~, ~, u] = shuttle(tmax, nt, guess(2,1), nx, method, false);

% sets n to third guess
n = 3;

% Shooting Method will continue to iterate until the final guess has
% acheived the desired precision for maximum temperature of the tile
while guess(2,end) > maxTemp || guess(2,end) < (maxTemp -
 tempPrecision)

    % work out the linear gradient, m, of the last two guesses
```

1

```
function [thickness] = minThicknessInv(maxTemp, tmax, nt, nx, method,
 tempUnitK, doPlot)

% This function determines the minimum tile thickness required to
 prevent
% an overshoot of an inputted maximum temperature at the inside of the
 tile.

% W Powell  06/04/21

% Required input arguments:
% maxTemp - maximum temperature desired at inside of tile
% tmax    - maximum time
% nt      - number of timesteps
% xmax    - total thickness
% nx      - number of spatial steps in x axis
% method - solution method ('forward', 'backward' etc)
% tempUnitK - true if Kelvin, false if Fahrenheit
% doPlot - true to plot graph; false to suppress graph.

% Output arguments:
% thickness    - minimum tile thickness required to withold from
 maxTemp

% For example, to perform function for a maximim temperatre of 450K:
% [thickness] = minThicknessInv(450, 4000, 501, 21, 'crank-nicolson',
 true, true)


% initial guesses for minimum thickness
guess(1,1) = 0.05;
guess(1,2) = 0.1;

% Desired temperature lower than maximum temperature(K or F)
tempPrecision = 1;

% first guess for shooting method
[~, ~, u] = shuttle(tmax, nt, guess(1,1), nx, method, false);
guess(2,1) = max(u(:,1)); % finds maximum temperature

% second guess
[~, ~, u] = shuttle(tmax, nt, guess(2,1), nx, method, false);

% sets n to third guess
n = 3;

% Shooting Method will continue to iterate until the final guess has
% acheived the desired precision for maximum temperature of the tile
while guess(2,end) > maxTemp || guess(2,end) < (maxTemp -
 tempPrecision)

    % work out the linear gradient, m, of the last two guesses
```

```
    m = (guess(2, n-2) - guess(2, n-1)) / (guess(1,n-2) -
 guess(1,n-1));

    % find the y intercept, c, using m
    c = guess(2, n-1) - m * guess(1, n-1);

    % An intelligent new guess of tile thickness is added to the nth
 value
    % of tile thickness guesses. The next guess is computed using
    % the Secant root finding method
    guess(1, n) = (maxTemp - c) / m;

    % finds maximum temperature in either K or F at that tile
 thickness
    if tempUnitK
        [~, ~, u] = shuttle(tmax, nt, guess(1,n), nx, method, false);
        guess(2, n) = max(u(:,1));
    else
        [~, ~, u] = shuttle(tmax, nt, guess(1,n), nx, method, false,
 false);
        guess(2, n) = max(u(:,1));
    end
    % prepares for next guess
    n = n + 1;
end

% final guess is used for minimum thickness
thickness = guess(1, end);

% plots 3d graph of tile temperature with respect to tile thickness,
 with time domain
if doPlot
    if tempUnitK
        shuttle(tmax, nt, guess(1,end), nx, method, true);
    else
        shuttle(tmax, nt, guess(1,end), nx, method, true, false);
    end
end

end
```

Published with MATLAB® R2020b