ME40064 Coursework 2 - Transient FEM Modelling

University of Bath - Department of Mechanical Engineering

Reece Bell - Candidate No 13771

Introduction

Finite Element Modelling (FEM) is an industry standard methodology of modelling complex physical interactions that can be defined by differential equations. It is used to predict behaviour where empirical investigation is impractical or too expensive. This report covers an exercise in developing a suite of functions that can solve transient diffusion-reaction problems in 1D space. These tools were then applied to a specific problem of modelling heat transfer through a human skin structure to infer the physical damage that this could cause. Extra emphasis has been placed on the verification and validation of the numerical methods used, as well as the confidence in results that this practice is able to provide.

Contents

I. Exercise 1

A. Developing the initial solver

The first task was to modify a previously developed FEM tool to solve the diffusion-reaction equation in its transient form, rather than simply the steady state solution. The steady state and transient expressions of these equations are included in Equations 1 and 2 respectively.

$$0 = D\frac{\partial^2 c}{\partial x^2} + \lambda c + f \tag{1}$$

$$\frac{\partial c}{\partial t} = D\frac{\partial^2 c}{\partial x^2} + \lambda c + f \tag{2}$$

The steady state solver was modified to operate over a time period by altering the low level functions to produce different local element matrices that are very similar to their previous versions. These new matrices are of the same dimensions as before and are therefore easy to implement within the same algorithmic structure used to construct and then solve the global matrix and vector.

With these changes, the solver is capable of computing a solution for the entire domain in parallel for each static point in time. In order to develop a transient solution finite differencing methods such as Euler and Crank Nicholson schemes were used to produce solutions at discrete steps in time. Stepping in time, the governing equation to connect the current and future solution matrices is defined in Equation 3. In this expression, θ is a value of either 0,0.5 or 1 depending on the differencing method selected.

$$[M + \theta\Delta t K]c^{n+1} = [M - \theta\Delta t K]c^{n} + \theta\Delta t[F^{n+1} + F^{n} + NBc^{n+1} + NBc^{n}] \tag{3}$$

Variable	Quantity
M	Mass matrix
Δt	Time step
K	Stiffness matrix
C	Solution matrix (at the current and next time-step)
F	Source vector (at the current and next time-step)
NBc	Von-Neumman boundary conditions (at the current and next time-step)

The global mass and stiffness matrices are formed using many local element matrices that are computed in sub-functions to evaluate the contributions of diffusion and reaction as appropriate. Initially, these integrals were solved analytically, as was demonstrated in lecture and tutorial examples.

For example, the diffusion matrix local element contribution can be proven to be defined by the diffusion coefficient and element jacobian as shown in Equation 4. Calculating each of these local element matrices (LEMs) allows them to be summed into the global matrix for solving.

$$D_{LocalElement} = \begin{bmatrix} \frac{D}{2J} & \frac{-D}{2J} \\ \\ \frac{-D}{2J} & \frac{D}{2J} \end{bmatrix} \tag{4}$$

B. Results and initial verification

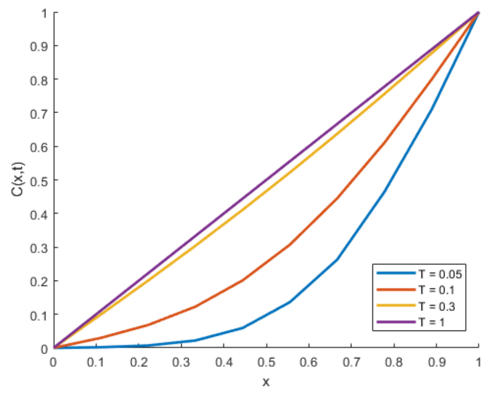
The first test for the transient solver was the application to the simplified problem of the transient diffusion problem, a manifestation of Equation 2 where λ and f are zero, and the diffusion co-efficient is 1. Using Dirichlet boundary conditions as shown, the analytical solution for this equation is shown in Equation 7.

$$\frac{\partial c}{\partial t} = \frac{\partial^2 c}{\partial x^2} \tag{5}$$

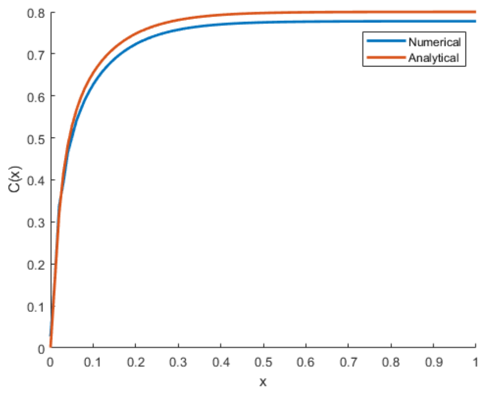
$$\text{Where: } x = [0,1],\ \ c(x,0) = 0,\ \ c(0,t) = 0,\ \ c(1,t) = 1 \tag{6}$$

$$c(x,t) = x + \frac{2}{\pi}\sum_{n=1}^{\infty}\frac{(-1)^n}{n}e^{-n^2\pi^2 t}sin(n\pi x) \tag{7}$$

Figure 1a shows the results of the transient solver at a number of time steps. The diffusion through space can be clearly seen over time, before settling to a steady state in equilibrium beyond $t = 1$. Viewing this for a single slice of the domain over the period allows for easy comparison to the known analytical solution, as can been seen in Figure 1b.



(a) Diffusion in space at discrete times



(b) Comparison of numerical and analytical solution

Fig. 1: Transient diffusion problem solution - Crank Nicholson differencing

Unit tests were used to verify that the functions developed to produce these results are performing as intended, segregating the sources of problems as finely as possible. This was useful during this project for simplifying the debugging process of a complex model with many interacting functions, in an industry setting however this practice is even more critical, as teams of people working on a single model will require even greater confidence in the functions they are using.

Full code for all functions used to generate these results is included in Appendix A, Any relevant unit tests that were used for verification purposes are also included here.

II. Improvements to the basic solver

With the initial realisation of the solver developed and verified, further steps to improve its performance and versatility were taken to expand the ease of applying it to various use cases. Before doing this, functions to perform an assessment of the L2 norm and carry out a convergence study were also developed as a final and ultimate verification of the basic solver and its functions on a system wide scale. Successfully verifying this provided the confidence needed to move forward in introducing further complexity into the solver system.

All code modified to accomplish these improvements is included in Appendix B.

A. Improvement 1 - L2 Norm assessment and convergence verification

With a known analytical solution for the problem as shown in 7, simple assessments of the error implicit in the numerical FEM solution can be made from inspection of graphs such as Figure 1b or by evaluating the difference in the two solutions at exact nodal points along the mesh.

For a more definite quantification of error, Gaussian quadrature (GQ) was used to exactly integrate the area between the two solution curves, providing a precise evaluation of the error in the numerical solution over the whole mesh. A function to create a Gauss scheme was adapted from tutorial session to do this here, as well as aiding in developing the versatility of the solver later.

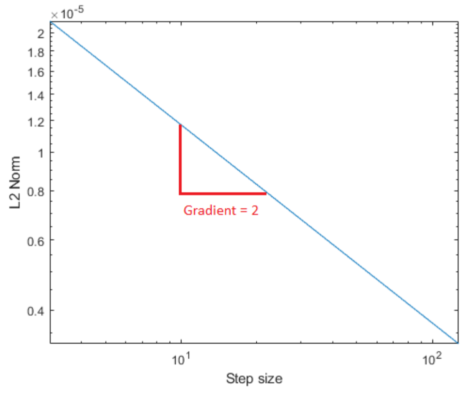
Linear interpolation of both the FEM solution and the x location within the basis function's ξ domains was used to quantify the solution at each Gauss point within each element and sum their weighted contributions. Summing these values after taking their root mean square (RMS), the "L2 Norm" can be found, which is equal to the total magnitude error between the two solutions for a slice of space or time. The size of this L2 Norm is thus dependent upon the run parameters of mesh size, number of time steps, and the differencing method used, along with the particular temporal or spatial point at which it is calculated.

Beyond error assessment, this L2 norm can be used to plot a convergence study on the error for differently sized meshes. By proof it can be shown that for linear basis functions, errors take the form of $O(h^2)$. Calculating the L2 norm for different sized meshes and plotting a log/log graph of L2 norm versus step size, a gradient of 2 shows correct FEM solver operation, as the error is reducing with step size at the precise rate predicted by the mathematical theory.

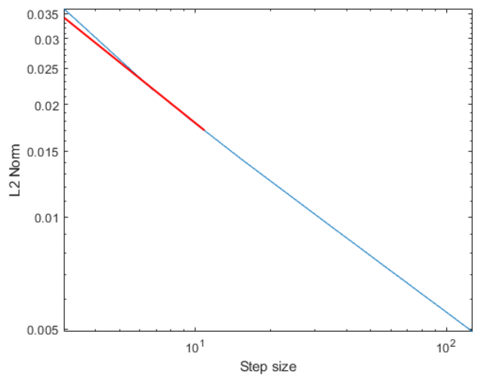
Figures 2a and 2b, along with Table I show this convergence verification. Although the mathematical proof implies that the gradient will be exactly 2 for linear basis functions, this was not observed to be possible here.

One reason for this the transient solution having to immediately react to a large step input of the enforced Dirichlet boundary condition, introducing extra oscillation error until the system settles to equilibrium, this is reflected by the deviation in the gradient at different time points in Table I. The closer the steady state solution the slice L2 norm is calculated, the more true to the proof the results are.

The other issue in the practical application of this convergence study is the inability to fully replicate the non-polynomial elements of the analytical solution. As a result, truncation errors caused by the numerical series representation of the $sin(n\pi x)$ term cause the gradient to deviate from the given theoretical solution.



(a) L2 norm convergence at steady state, t=1



(b) L2 norm convergence at transient, t=0.1

Fig. 2: L2 Norm convergence assessment

TABLE I: L2 norm convergence at difference points in time

t	Calculated gradient
1	1.999
0.5	1.990
0.1	1.881
0.01	1.298

B. *Improvement 2 - Gaussian quadrature and quadratic basis functions*

Using the infrastructure already developed to implement a Gaussian quadrature solution to integrals, the methodology of the solver was altered to use GQ to compute local element components for building the mass and stiffness global matrices. This removes the need to analytically solve local element integrals before implementing them into code and greatly increases the ease at which the solver can be applied to different FEM problems.

As long as sufficient Gauss points are specified, this additional flexibility is gained with 0 decrease in model accuracy. Throughout this project, 3 Gauss points was sufficient to achieve this.

With the Gauss scheme defined, the integrals to be solved were formed within the mesh by calculating ψ and $\frac{d\psi}{d\xi}$, needed to form components of the mass (M) and stiffness (K) matrices given by equations 8 and 9 respectively.

$$M_{element} = \int_{-1}^{1} \psi_n \psi_m J d\xi \tag{8}$$

$$K_{element} = \int_{-1}^{1} D \frac{d\psi_n}{d\xi}\frac{d\xi}{dx}\frac{d\psi_m}{d\xi}\frac{d\xi}{dx} J d\xi - \int_{-1}^{1} \lambda \psi_n \psi_m J d\xi \tag{9}$$

In the process of modifying the functions responsible for computing the LEMs, the basis functions used to define each matrix were also changed over to a quadratic scheme. Using nodal quadratic Langrange basis functions, 3 local nodes are used for each element, as defined in Equations 10 to 12.

$$\psi_o(xi) = \frac{\xi(\xi-1)}{2} \tag{10}$$

$$\psi_1(xi) = 1 - \xi^2 \tag{11}$$

$$\psi_2(xi) = \frac{\xi(\xi+1)}{2} \tag{12}$$

Using this 3 local node structure, the LEMs retain their shape but increase to a 3x3 size. Using dynamic loop and step sizing, LEMs of any size can be assembled into a global matrix within a single function.

Verification of this two step addition of different basis functions and integral solving methods was done by direct comparison to the simple instantiation of the solver that was presented in I-A by checking that all nodal values calculated are the same between the two versions (Within a tolerance of 0.01% for different rounding, more accurate interpolation etc.). Comparison by manual inspection of the two solvers to the analytical solution provided was also used to confirm this.

With the simple version being verified extensively by both piece-wise unit tests, along with holistic L2 norm convergence assessment, the ability to replicate it's behaviour was seen as sufficient to view the new advanced solver with confidence in its results.

C. Improvement 3 - Differencing method and parameter optimisation

Finally, the solver can be optimised between computational cost and numerical error by using the previously set up L2 norm assessment to size temporal and spatial steps appropriately, as well as investigating any performance differences between the three finite difference models available. The structure of the solver uses θ notation to solve each global matrix equation at each time step, allowing the differencing method to be input as part of the function call.

Selecting somewhat arbitrary parameters of 10 nodes and 1000 timesteps in order to ensure stability for the forward Euler method, the L2 norm was calculated and summed along the spatial domain at regular intervals in time for each of the 3 numerical methods. Figure 3 shows this variation.

Simple summed error is not necessarily the best metric for performance, as consistent yet minor oscillation is often preferable to a single large spike in error for many modelling use cases, yet could yield a higher total error. For this assessment however, it is assumed that as long as stability is maintained, the general behaviour of each method is similar enough to use total error as a measure.

From Figure 3 the backwards Euler method can immediately discounted. As an implicit method it is just as complex and computationally expensive to implement as Crank-Nicholson, yet offers the worst performance.

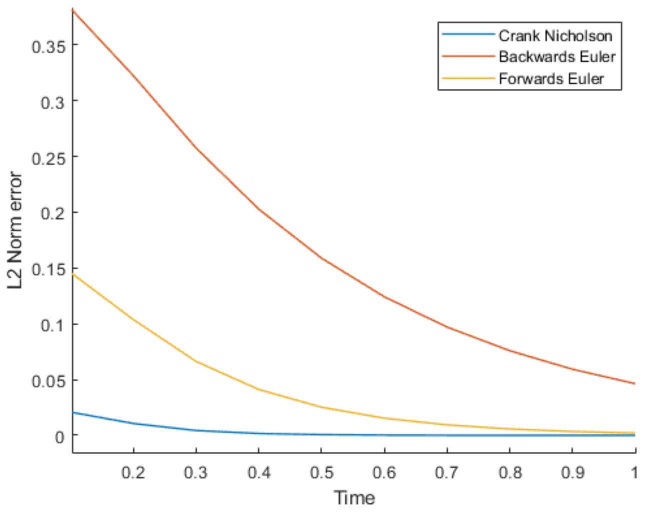


Fig. 3: Summed L2 norm for different differencing methods

As an explicit method, forward Euler was expected to compensate for its lack of accuracy compared to Crank Nicholson due to its simplicity of operation and lesser need for computation. However, a timing analysis showed that even for a run of 1000 timesteps, the difference in runtime for each model to be solved was very small, as shown in Table II. This is likely due to the generalised philosophy with which the solver was written, where use of θ notation define the differencing method leads to innefficient computation when $\theta = 0$.

TABLE II: Run time for different methods

Method	Runtime
Crank-Nicholson	13.88
Forward Euler	13.82
Backwards Euler	13.88

The Crank Nicholson method was selected as it gives the best performance with the added benefit of being unconditionally stable. Applying a basic sensitivity study, the number of elements was increased incrementally until the relative change in the l2 norm ΔE fell below a certain threshold, arbitrarily set at 1%, before doing the same for timestep.

This hasty optimisation thus yields the parameters shown in Table III for moving forwards to model application.

TABLE III: Run time for different methods

Parameter	Value
Differencing method	Crank-Nicholson
Number of mesh elements	520
Number of time steps	1000

A full optimisation study is beyond the scope of this project, however, the simplistic methodology employed here provides most of the value this could yield. In an industrial setting, this would be a much deeper process, and factors such as computational resources available, along with the accuracy demanded by the model's use case would be fed into this trade space.

III. Exercise 2

A. Application for physical modelling

With the FEM solver developed, verified and optimised, it can now be applied practically to model a physical system. Here, the example of heat transfer through human skin tissue was chosen, informing the design of heat protective equipment to prevent burns. All code modified and developed to do this is included in Appendix C.

Human skin was modeled using a 1 dimensional scheme of discrete regions for the Epidermis, Dermis and Sub-cutaneous layers, each with their own material properties. To apply the solver to these changing material properties, the mesh data structure was expanded to include their values at the appropriate elements. This forms a mesh structure as shown in Figure 4

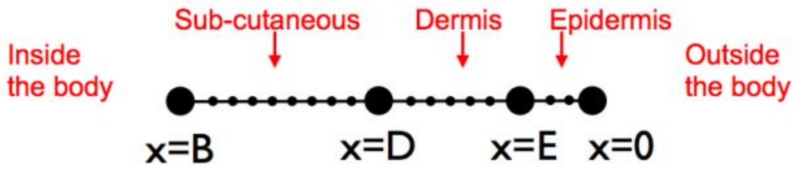


Fig. 4: Mesh structure used to represent human skin [1]

The material properties included in this model are as shown in Table IV and the governing equation that relates these to temperature change in Equation 13.

TABLE IV: Human skin material properties

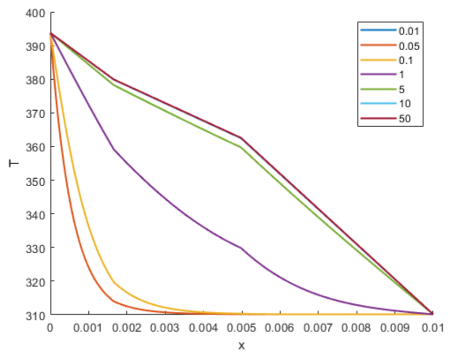
Parameter	Meaning	Epidermis	Dermis	Sub-Cutaneous
k	Thermal conductivity	25	40	20
G	Blood flow rate	0	0.0375	0.0375
ρ	Skin density	1200	1200	1200
c	Skin SHC	3300	3300	3300
ρ_b	Blood density	-	1060	1060
c_b	Blood SHC	-	3770	3770
T_b	Blood temp	-	310.15	310.15
x	Layer thickness	0.0016667	0.005	0.01

$$\frac{\partial T}{\partial t} = (\frac{k}{\rho c})\frac{\partial^2 T}{\partial x^2} - (\frac{G\rho_b c_b}{\rho c})T + (\frac{G\rho_b c_b}{\rho c})T_b \tag{13}$$

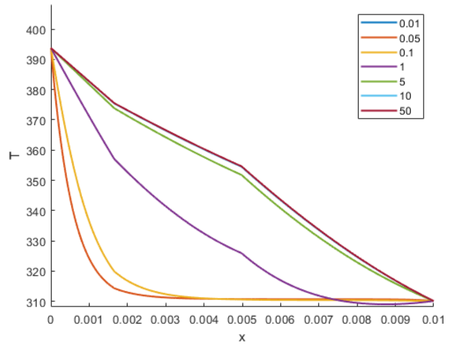
B. Transient results

Running the solver over a period of 50 seconds for initial conditions as defined in Equation 14 yields the results as shown in Figures 5a and 5b, which contrast the effect of the bloods ability to sink heat away from the skin's surface. This model was ran using the basically optimised run parameters found in part 1F, with the number of timesteps scaled to compensate for the larger run period. The number of spatial steps was kept constant in order to maintain a high level of mesh fidelity and reduce the impact of step changes in material properties.

$$\text{Where: } T(x,0) = 310.15K,\ \ (x=B,t) = 310.15K,\ \ T(x=0,t) = 393.15K \tag{14}$$



(a) Temperature change with zero blood flow



(b) Temperature change with blood flow heat-sinking

Fig. 5: Heat transfer in human skin modelling results

These graphs predict the temperature profile within the skin that would occur if someone were to be in contact with a constant heat source over 50 seconds. The exterior of the skin remains at 390.15K for the full period, with heat rapidly diffusing into the tissue over time and raising its temperature above the internal body temperature of 310.15K. After 5 seconds in contact with the source, the piece-wise linear steady state temperature profile is almost entirely achieved.

Minor differences in the temperature profile between the two models can be seen. As expected, the model that accounts for blood flow has a greater decay in temperature through the tissue as heat is sunk into the blood and carried away. For example, halfway through the mesh at the Dermis layer, the temperature is 364.0K with 0 blood flow, and 354.4K with blood flow modelled. This yields a 2.2% change in the solution, the significance of which will depend on the use case of the model, as will be explored later.

As a final point on Figure 5b, the t=1 solution falls below the internal body temperature of 310.15K at the end of the domain. This is believed to be an oscillatory overshoot resulting from FEM methods, and highlights the inherent fallibility in their application.

C. Calculation of tissue damage

Using the Arrhenius rate equation to model the damage done to the skin, Equation 15 can be used to determine the level of burn that a person would suffer if this temperature profile occurred in their skin.

$$\Gamma = \int_{t_{max}}^{t_{burn}} 2 \times 10^{98} exp(-\frac{12017}{(T - 273.15)})dt \tag{15}$$

In order to accurately assess the level of heat damage at a certain point in space, a function was written that is able to calculate ξ and use the advanced quadratic basis functions to interpolate the solution between nodal points where necessary.

With a second degree burn occurring when Γ ¿ 1 at the Epidermis and a third degree when this happens at the Dermis, the solver can now product how serious a burn will be for different external heat inputs. With the default settings defined in Equation 14, this causes extraordinarily high values of Γ up to 10^{52}. As illustrated on Figure 6, this shows that without protective equipment a person would be very seriously injured if exposed to this heat, which is not surprising for on skin exposure to 120°C for 50 seconds.

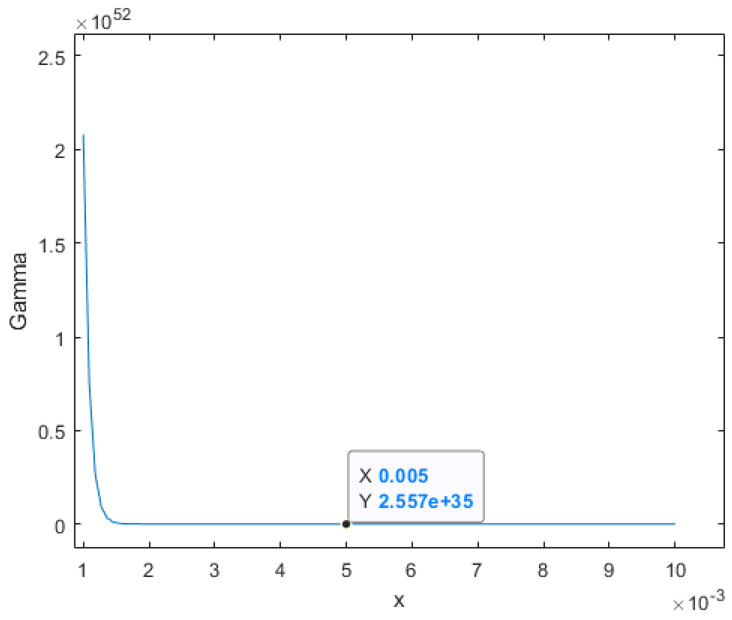


Fig. 6: Degree of burn damage, Γ throughout the skin with blood flow heat-sinking

From this, it is clear that a temperature reduction at the exterior boundary must be enforced to prevent significant harm to individuals in this environment. This reduction that must be achieved by protective equipment to prevent a burn can be found by using a basic search algorithm to calculate burn damage at incremental decreases to the boundary temperature.

By sizing Γ to be 1 at the relevant boundaries, this search algorithm found values of heat reduction needed by protective equipment, and hence the Dirichlet boundary condition that would successfully avoid a burn, as shown in Table V. This search was ran for a precision of 0.1°C, the reasons for which will be explored in the following section.

TABLE V: Temperature reductions found using different models

Degree of burn to prevent	Blood flow considered	Temperature reduction needed	Dirichlet boundary condition required
2nd	No	59.2	333.95
2nd	Yes	43.5	346.65
3rd	No	65.6	327.55
3rd	Yes	58.7	334.45

These results show that the effect of improving the accuracy of the model by incorporating blood flow effects has a significant impact on the output of this study, with a delta of almost 16°C between the two versions. As such, although only a minor change in nodal temperature is caused by improving the accuracy of the model by including blood flow, the impact of this on the use case of the model is much greater. This justifies the extra effort in incorporating blood, quadratic basis functions and interpolation as well as making it desirable to seek further model improvements.

With this in mind, it may also be advisable to re-run the optimisation study with a greater weighting on accuracy compared to computational demand. This is compounded by the safety critical nature of the equipment being developed, as with safety margins likely to be significant, any error is amplified further by the addition of these margins.

D. *Reflections on validity, accuracy and precision*

The solver used to compute this model has been verified extensively from a mathematical standpoint, however for a mechanistic and deterministic model such as this, the results are only as valid as the assumptions the model was built upon. Empirical validation of this model is the most reliable method of validating it, such as by using an instrumented test sample that has similar material properties to human skin to measure the heat transfer through it.

As discussed in part 2c. Improvements to the accurate representation of skin physiology have large impacts on the output of this study. Further possible additions that could improve the accuracy of these results include, but aren't limited to:

- Modelling changes in blood temperature as the blood supply becomes saturated with heat
- More detailed layering of skin, with specificity to the location on the body
- 2D and 3D implementation that models heat diffusion into surrounding skin, dependent on the source contact area

Finally, the inherent nature of FEM and its contribution of rounding error drift from iterative computation, along with truncation errors caused by differencing methods introduces further accuracy degradation.

The degree of this uncertainty is hard to definitively quantify and a full investigation of error is beyond the scope of this study, with rounding errors assumed to be negligible. Using $O(h)^3$ and $O(h)^2$ assessments of truncation errors in space and time respectively, the total numerical error is of the order of 1×10^{-6}. Despite the high accuracy that this implies, the high gain in the burn equation demonstrated by Equation 6 makes calculating to a precision beyond 0.1K pointless.

This implies that error in the solution is dominated by simplifications in the mechanistic model developed and until empirical investigation can minimise their uncertainty, hefty safety margins needs to be applied to these results.

Assuming the addition of blood flow was the largest improvement possible, the delta in temperature reduction needed can be used to estimate future margins. With the largest difference being 27.2%, rounding this to 30% and then doubling it due to the safety critical nature of the system gives a margin of 60%. A sensible and conservative estimate for a relatively basic and immature model such as this.

References

[1] Cookson, A 2021, *Lecture material, notes and tutorials* ,System Modelling and Simulation ME40064, University of Bath

[2] Liu, J et al 2020, *Balancing truncation and round-off errors in FEM: One-dimensional analysis* ,Journal of Computational and Applied Mathematics, Delft Institute of Applied Mathematics

Appendix A
Appendix A - Basic solver code

All functions used to achieve the basic FEM solver presented in part 1a. are included in this appendix. Scripts used for debugging and to produce plots have been omitted. Unit tests are also included here as evidence of efforts made in verification.

A. Solver "Top" function

```
%Function to solve Laplace's equation for given parameters of diffusion and
%reaction coefficients. As a Laplacian problem the source terms are 0.
%
%Takes the following arguments:
%
%D - Diffusion Co-efficient (Float)
%Lamda - Reaction Co-efficient (Float)
%NNodes - Number of nodes in global mesh (NElements = NNodes - 1) (Int)
%BC0 - Type of node 0 boundary condition, 'DL' for Dirichlet or 'VN' for Von
%Nuemman (Str)
%BC0Val - Value of c or dc/dx for node 0 boundary condition (Float)
%BC1 - Node 1 boundary condition, same format as BC0
%BC1Val - Value of c or dx/dx for node 1 boundary condition (Float)
%DM - Differencing Method, can take 'CN','FE','BE' (String)
%E.g. SolveLaplaceTransient(1,-9,5,100,'DL',0,'DL',1,'CN')
%
%This is the start of the transient solution and has been extensively
%modified from the previously submitted "SolveLaplace.m"
%
%Note that the domain is currently hardcoded from x = 0 to x = 1

function [C, Domain, TDomain] = SolveLaplaceTransient(D,Lamda,NNodes,NTsteps,BC0,BC0Val,BC1,BC1Val,DM)

%Set domain
xmin = 0;
xmax = 1;

%Set time scheme
tmax = 1;
dt = tmax/NTsteps;

%Define theta dependent upon the difference method selected
if DM == 'CN'
    theta = 1;
elseif DM == 'FE'
    theta = 0;
elseif DM == 'BE'
    theta = 0.5;
end

% Initialise mesh
Mesh = OneDimLinearMeshGen(xmin,xmax,NNodes-1); % Elements will also be N-1 ;
%Size of global mesh effects local element values due to varying J scaling

%Initialise matrices
StiffnessMatrix = zeros(NNodes,NNodes);
```

```
MassMatrix = zeros(NNodes,NNodes);
GlobalMatrix = zeros(NNodes,NNodes);
GlobalVector = zeros(NNodes,1);
SourceVector = zeros(NNodes,1); % Source term is all 0s here
C = zeros(NNodes,NTsteps);

% Need two solutionvectors to implement timestepping
Ccurrent = zeros(NNodes,1); %Define initial conditions here
Cnext = zeros(NNodes,1);

Fcurrent = zeros(NNodes,1); %Capability for timevariant source term
Fnext = Fcurrent;

NBCcurrent = zeros(NNodes,1); %Capability for timevariant Von-Neumman
NBCnext = NBCcurrent;         %Boundary conditions

% Populate global stiffness matrix
StiffnessMatrix = GlobalStiffness(StiffnessMatrix,D,Lamda,Mesh);

% Populate a global mass matrix
MassMatrix = GlobalMass(MassMatrix,Mesh);

% Combine into an overall global matrix to constitute the LHS equation
GlobalMatrix = MassMatrix + theta*dt*StiffnessMatrix;

for idxt = 1 : NTsteps
    % Assemble the global source vector
    Fnext = GlobalSource(SourceVector,Mesh);
    % Construct Previous solution, source term and boundary RHS
    PrevSolution = (MassMatrix - (1-theta)*dt*StiffnessMatrix)*Ccurrent;
    SourceNew = dt*theta*(Fnext+NBCnext);
    SourceCurrent = dt*(1-theta)*(Fcurrent + NBCcurrent);
    CombinedRHS = PrevSolution + SourceNew + SourceCurrent;

    %Enforce boundary conditions at node 0
    switch BC0
        case 'VN'
            SourceVector(1) = SourceVector(1) + -BC0Val; % Need to add for VN
        case 'DL'
            GlobalMatrix(1,:) = 0;
            GlobalMatrix(1,1) = 1;
            CombinedRHS(1) = BC0Val;
    end

    %Enforce boundary conditions at node 1
    switch BC1
        case 'VN'
            SourceVector(end) = SourceVector(end) + -BC1Val; % Need to add for VN
        case 'DL'
            GlobalMatrix(end,:) = 0;
            GlobalMatrix(end) = 1;
            CombinedRHS(end) = BC1Val;
    end

    Cnext = GlobalMatrix\CombinedRHS; % Solve equation for this timestep
    %write to output matrix for plotting, saving and analysis
    C(:,idxt) = Cnext;

    %Step time variant terms
    Ccurrent = Cnext;
    Fcurrent = Fnext;
    NBCcurrent = NBCnext;

end

%Output relevant domains for plotting, saving and analysis
Domain = linspace(xmin,xmax,NNodes);
TDomain = linspace(0,tmax,NTsteps);

end
```

B. Mesh generator

```
function [mesh] = OneDimLinearMeshGen(xmin,xmax,Ne)
%%This function generates a one dimensional, equispaced, linear finite
%%element mesh, with Ne number of elements, between the points at x
%%position xmin and xmax.
```

```

    mesh.ne = Ne; %set number of elements
    mesh.ngn = Ne+1; %set number of global nodes
    mesh.nvec = zeros(mesh.ngn,1); %allocate vector to store global node values
    dx = (xmax - xmin)/Ne; %calculate element size

    mesh.nvec = xmin:dx:xmax;

    %loop over elements & set the element properties
    for i=1:Ne

        %set spatial positions of nodes
        mesh.elem(i).x(1) = xmin + (i-1)*dx; %sets local elements
        mesh.elem(i).x(2) = xmin + i*dx ;

        %set global IDs of the nodes
        mesh.elem(i).n(1) = i;
        mesh.elem(i).n(2) = i+1;

        %set element Jacobian based on mapping to standard element
        mesh.elem(i).J = 0.5*dx; %this is assuming standard element of -1 to 1


    end

end
```

C. Global Stiffness matrix function

```
function StiffnessMatrix = GlobalStiffness(StiffnessMatrix,D,Lamda,Mesh)
%Function to assemble a global stiffness matrix using diffusion and
%reaction local element components

%Takes:
%StiffnessMatrix - Initialised to zeros, (N x N float)
%D - Diffusion coefficient (float)
%Lamda - Reaction coefficient (float)
%eID - Element number (int)
%msh - Mesh data structure, generated using OneDimLinearMeshGen.m or
%OneDimSimpleRefinedMeshGen.m (struct)

for idx = 1: length(StiffnessMatrix) -1

    % Generate diffusion local elements and populate global matrix
    LocalMatrix = LaplaceElemMatrix(D,idx,Mesh);
    StiffnessMatrix(idx:idx+1,idx:idx+1) = StiffnessMatrix(idx:idx+1,idx:idx+1) + LocalMatrix ;

    % Generate reaction local elements and populate global matrix
    LocalMatrix = ReactionMatrix(Lamda,idx,Mesh);
    StiffnessMatrix(idx:idx+1,idx:idx+1) = StiffnessMatrix(idx:idx+1,idx:idx+1) - LocalMatrix ;

end
```

D. Global Mass matrix function

```
%Function to assemble a global mass matrix

%Takes:
%MassMatrix - Initialised to zeros(N x N float)
%Mesh - Mesh data structure, , generated using OneDimLinearMeshGen.m or
%OneDimSimpleRefinedMeshGen.m (struct)

function MassMatrix = GlobalMass(MassMatrix,Mesh)


for idx = 1: length(MassMatrix) -1

    % Generate mass local elements and populate global matrix
    LocalMatrix = LocalMassMatrix(idx,Mesh);
    MassMatrix(idx:idx+1,idx:idx+1) = MassMatrix(idx:idx+1,idx:idx+1) + LocalMatrix ;

end
```

E. Global Source vector function

```
%Function to assemble a Global source vector from local source elements
%As demonstrated in the lecture notes
```

```

%Takes:
%SourceVector - Initialised to zeros(N x N float)
%Mesh - Mesh data structure, , generated using OneDimLinearMeshGen.m or
%OneDimSimpleRefinedMeshGen.m (struct)

function SourceVector = GlobalSource(SourceVector,Mesh)

for idx = 1 : length(SourceVector)
    if idx == 1
        SourceVector(idx) = SourceVector(idx) * Mesh.elem(idx).J;
    elseif idx == length(SourceVector)
            SourceVector(idx) = SourceVector(idx) * Mesh.elem(idx-1).J;
    else
        SourceVector(idx) = SourceVector(idx) * (Mesh.elem(idx-1).J + Mesh.elem(idx).J);
    end

 end
```

F. Local Element Mass matrix function

```
%Function to calculate the reaction local element Mass matrix for a given
%mesh element
%Built off of previous ReactionMatrix function
%Takes:
%eID - Element number (int)
%msh - Mesh data structure, generated using OneDimLinearMeshGen.m or
%Or OneDimSimpleRefinedMeshGen.m (struct)

function LocalElementmatrix = LocalMassMatrix(eID,msh)

%Extract J from msh structure
J = msh.elem(eID).J;

%Form local element matrix using equation derived in the notes
LocalElementmatrix = [ (2*J)/3 J/3  ;
                      J/3 (2*J)/3  ];
```

G. Local Element Diffusion matrix function

```
%Function to calculate the diffusion local element matrix for a given
%mesh element
%Takes:
%D - Diffusion coefficient (float)
%eID - Element number (int)
%msh - Mesh data structure, generated using OneDimLinearMeshGen.m or
%OneDimSimpleRefinedMeshGen.m (struct)

function LocalElementmatrix = LaplaceElemMatrix(D,eID,msh)

%Extract J from msh structure
J = msh.elem(eID).J;

%Form local element matrix using equation derived in the notes
LocalElementmatrix = [ D/(2*J) -D/(2*J) ;
                       -D/(2*J) D/(2*J)];
```

H. Local Element Reaction matrix function

```
%Function to calculate the reaction local element matrix for a given
%mesh element
%Takes:
%Lamda - Reaction coefficient (float)
%eID - Element number (int)
%msh - Mesh data structure, generated using OneDimLinearMeshGen.m or
%Or OneDimSimpleRefinedMeshGen.m (struct)

function LocalElementmatrix = ReactionMatrix(Lamda,eID,msh)

%Extract J from msh structure
J = msh.elem(eID).J;

%Form local element matrix using equation derived in the notes
LocalElementmatrix = [ (2*Lamda*J)/3 (Lamda*J)/3  ;
                       (Lamda*J)/3  (2*Lamda*J)/3 ];
```

I. Global Stiffness matrix unit test

```
%% Test 1 : Verify that stiffness matrix is correctly combining reaction
% and diffusion terms
% Note that these sub functions have already been verified separately
tol = 1e-14;
NElements = 10;
D = 2;
Lamda = -9;
eID = 1;
msh = OneDimLinearMeshGen(0,1,NElements);
StiffnessMat = zeros(NElements);

LaplaceMat = LaplaceElemMatrix(D,eID,msh);
ReactionMat = ReactionMatrix(Lamda,eID,msh);
StiffnessMat = GlobalStiffness(StiffnessMat,D,Lamda,msh);

assert(StiffnessMat(eID,eID) - (LaplaceMat(eID,eID) - ReactionMat(eID,eID)) <= tol)

%% Test 2 : Verify matrix symmetry
% Note that these sub functions have already been verified separately
tol = 1e-14;
NElements = 10;
D = 2;
Lamda = -9;
msh = OneDimLinearMeshGen(0,1,NElements);
StiffnessMat = zeros(NElements);

StiffnessMat = GlobalStiffness(StiffnessMat,D,Lamda,msh);

assert(StiffnessMat(1,1) - StiffnessMat(end,end) <= tol)

%% Test 3 : Verify matrix diagonal pattern as shown in lectures
% Note that these sub functions have already been verified separately
tol = 1e-14;
NElements = 10;
D = 2;
Lamda = -9;
eID = 1;
msh = OneDimLinearMeshGen(0,1,NElements);
StiffnessMat = zeros(NElements);

StiffnessMat = GlobalStiffness(StiffnessMat,D,Lamda,msh);

assert(StiffnessMat(eID,eID) - 2*StiffnessMat(1,1) <= tol)
```

J. Global Mass matrix unit test

```

%% Test 1: test symmetry of the matrix
% % Test that this matrix is symmetric
NElements = 10;
NNodes = NElements + 1;
msh = OneDimLinearMeshGen(0,1,NElements);
MassMatrix = zeros(NNodes,NNodes);
MassMatrix = GlobalMass(MassMatrix,msh);
tol = 1e-14;
msh = OneDimLinearMeshGen(0,1,10);
assert(abs(MassMatrix(1,1) - MassMatrix(end,end)) <= tol)

%% Test 2: test symmetry of the matrix
% % Test that diagonal summing is working as needed
NElements = 10;
NNodes = NElements + 1;
msh = OneDimLinearMeshGen(0,1,NElements);
MassMatrix = zeros(NNodes,NNodes);
MassMatrix = GlobalMass(MassMatrix,msh);
tol = 1e-14;
msh = OneDimLinearMeshGen(0,1,10);
assert(abs(MassMatrix(1,1) - 0.5* MassMatrix(2,2)) <= tol)
```

K. Global Source vector unit test

```
%% Test 1: Test Matrix Geometry
% % Test that this matrix is 1 Dimensional
NElements = 10;

msh = OneDimLinearMeshGen(0,1,NElements);
SourceVector = zeros(NElements,1);
```

```matlab
SourceVector = GlobalSource(SourceVector,msh);

assert(length(SourceVector(1,:)) == 1);

%% Test 2: Test Matrix Geometry
% % Test that the source vector follows pattern shown in lectures
tol = 1e-14;
NElements = 10;

msh = OneDimLinearMeshGen(0,1,NElements );
SourceVector = zeros(NElements);
SourceVector = GlobalSource(SourceVector,msh);

assert(SourceVector(1) - SourceVector(end) <= tol)

%% Test 3: Test Matrix Geometry
% % Test that the source vector follows pattern shown in lectures
tol = 1e-14;
NElements = 10;

msh = OneDimLinearMeshGen(0,1,NElements );
SourceVector = zeros(NElements);
SourceVector = GlobalSource(SourceVector,msh);

assert(SourceVector(2) - SourceVector(end-1) <= tol)
```

L. Local Element Mass matrix unit test

```matlab
%Similarities between diffusion and reaction local element matrices allow
%lots of the provided code to be reused here:

%% Test 1: test symmetry of the matrix
% % Test that this matrix is symmetric. This is the same verification as for the
% diffusion LEM's
tol = 1e-14;
eID=1; %element ID

msh = OneDimLinearMeshGen(0,1,10);
elemat = LocalMassMatrix(eID,msh);

assert(abs(elemat(1,2) - elemat(2,1)) <= tol)

%% Test 2: test 2 different elements of the same size produce same matrix
% % Test that for two elements of an equispaced mesh, as described in the
% % lectures, the element matrices calculated are the same. This is the
%   same verification as for the diffusion LEM's
tol = 1e-14;
eID=1; %element ID
msh = OneDimLinearMeshGen(0,1,10);

elemat1 = LocalMassMatrix(eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

eID=2; %element ID

elemat2 = LocalMassMatrix(eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

diff = elemat1 - elemat2;
diffnorm = sum(sum(diff.*diff));
assert(abs(diffnorm) <= tol)

%% Test 3: test that one matrix is evaluted correctly
% % Test that element 1 of the three element mesh problem described in the lectures
% % the element matrix is evaluated correctly. This uses the example shown
% % from tutorial sheet 3.
tol = 1e-14;
Lamda = 9; %Reaction coefficient
eID=1; %element ID
msh = OneDimLinearMeshGen(0,1,3);

elemat1 = LocalMassMatrix(eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

elemat2 = [ 1/9 0.5/9; 0.5/9 1/9];
diff = elemat1 - elemat2; %calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); %calculates the total squared error between the matrices
assert(abs(diffnorm) <= tol)

%% Test 4: test that different sized elements in a mesh are evaluted correctly - element 1
```

```
% % Test that elements in a non-equally spaced mesh are evaluated correctly
tol = 1e-14;
Lamda = 6; %Reaction coefficient
eID=1; %element ID
msh = OneDimSimpleRefinedMeshGen(0,1,5);

elemat1 = LocalMassMatrix(eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

elemat2 = [ 1/6 0.5/6; 0.5/6 1/6];
diff = elemat1 - elemat2; %calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); %calculates the total squared error between the matrices
assert(abs(diffnorm) <= tol)

%% Test 5: test that different sized elements in a mesh are evaluted correctly - element 4
% % Test that elements in a non-equally spaced mesh are evaluated correctly
tol = 1e-14;
Lamda = 48; %Reaction coefficient
eID=4; %element ID
msh = OneDimSimpleRefinedMeshGen(0,1,5);

elemat1 = LocalMassMatrix(eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

elemat2 = [ 1/48 0.5/48; 0.5/48 1/48];
diff = elemat1 - elemat2; %calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); %calculates the total squared error between the matrices
assert(abs(diffnorm) <= tol)
```

M. Local Element Diffusion matrix unit test

```
%% Test 1: test symmetry of the matrix
% % Test that this matrix is symmetric
tol = 1e-14;
D = 2; %diffusion coefficient
eID=1; %element ID
msh = OneDimLinearMeshGen(0,1,10);

elemat = LaplaceElemMatrix(D,eID,msh); %THIS IS THE FUNCTION YOU MUST WRITE

assert(abs(elemat(1,2) - elemat(2,1)) <= tol)

%% Test 2: test 2 different elements of the same size produce same matrix
% % Test that for two elements of an equispaced mesh, as described in the
% % lectures, the element matrices calculated are the same
tol = 1e-14;
D = 5; %diffusion coefficient
eID=1; %element ID
msh = OneDimLinearMeshGen(0,1,10);

elemat1 = LaplaceElemMatrix(D,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

eID=2; %element ID

elemat2 = LaplaceElemMatrix(D,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

diff = elemat1 - elemat2;
diffnorm = sum(sum(diff.*diff));
assert(abs(diffnorm) <= tol)

%% Test 3: test that one matrix is evaluted correctly
% % Test that element 1 of the three element mesh problem described in the lectures
% % the element matrix is evaluated correctly
tol = 1e-14;
D = 2.5; %diffusion coefficient
eID=1; %element ID
msh = OneDimLinearMeshGen(0,1,3);

elemat1 = LaplaceElemMatrix(D,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

elemat2 = [ 7.5 -7.5; -7.5 7.5];
diff = elemat1 - elemat2; %calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); %calculates the total squared error between the matrices
assert(abs(diffnorm) <= tol)

%% Test 4: test that different sized elements in a mesh are evaluted correctly - element 1
% % Test that elements in a non-equally spaced mesh are evaluated correctly
tol = 1e-14;
D = 1; %diffusion coefficient
```

```
eID=1; %element ID
msh = OneDimSimpleRefinedMeshGen(0,1,5);

elemat1 = LaplaceElemMatrix(D,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

elemat2 = [ 2 -2; -2 2];
diff = elemat1 - elemat2; %calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); %calculates the total squared error between the matrices
assert(abs(diffnorm) <= tol)

%% Test 5: test that different sized elements in a mesh are evaluted correctly - element 4
% % Test that elements in a non-equally spaced mesh are evaluated correctly
tol = 1e-14;
D = 1; %diffusion coefficient
eID=4; %element ID
msh = OneDimSimpleRefinedMeshGen(0,1,5);

elemat1 = LaplaceElemMatrix(D,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

elemat2 = [ 16 -16; -16 16];
diff = elemat1 - elemat2; %calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); %calculates the total squared error between the matrices
assert(abs(diffnorm) <= tol)
```

N. Local Element Reaction matrix unit test

```
%Similarities between diffusion and reaction local element matrices allow
%lots of the provided code to be reused here:

%% Test 1: test symmetry of the matrix
% % Test that this matrix is symmetric. This is the same verification as for the
% diffusion LEM's
tol = 1e-14;
Lamda = 2; %Reaction coefficient
eID=1; %element ID

msh = OneDimLinearMeshGen(0,1,10);
elemat = ReactionMatrix(Lamda,eID,msh); %THIS IS THE FUNCTION YOU MUST WRITE

assert(abs(elemat(1,2) - elemat(2,1)) <= tol)

%% Test 2: test 2 different elements of the same size produce same matrix
% % Test that for two elements of an equispaced mesh, as described in the
% % lectures, the element matrices calculated are the same. This is the
%   same verification as for the diffusion LEM's
tol = 1e-14;
Lamda = 5; %Reaction coefficient
eID=1; %element ID
msh = OneDimLinearMeshGen(0,1,10);

elemat1 = ReactionMatrix(Lamda,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

eID=2; %element ID

elemat2 = ReactionMatrix(Lamda,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

diff = elemat1 - elemat2;
diffnorm = sum(sum(diff.*diff));
assert(abs(diffnorm) <= tol)

%% Test 3: test that one matrix is evaluted correctly
% % Test that element 1 of the three element mesh problem described in the lectures
% % the element matrix is evaluated correctly. This uses the example shown
% % from tutorial sheet 3.
tol = 1e-14;
Lamda = 9; %Reaction coefficient
eID=1; %element ID
msh = OneDimLinearMeshGen(0,1,3);

elemat1 = ReactionMatrix(Lamda,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

elemat2 = [ 1 0.5; 0.5 1];
diff = elemat1 - elemat2; %calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); %calculates the total squared error between the matrices
assert(abs(diffnorm) <= tol)

%% Test 4: test that different sized elements in a mesh are evaluted correctly - element 1
```

```
% % Test that elements in a non-equally spaced mesh are evaluated correctly
tol = 1e-14;
Lamda = 6; %Reaction coefficient
eID=1; %element ID
msh = OneDimSimpleRefinedMeshGen(0,1,5);

elemat1 = ReactionMatrix(Lamda,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

elemat2 = [ 1 0.5; 0.5 1];
diff = elemat1 - elemat2; %calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); %calculates the total squared error between the matrices
assert(abs(diffnorm) <= tol)

%% Test 5: test that different sized elements in a mesh are evaluted correctly - element 4
% % Test that elements in a non-equally spaced mesh are evaluated correctly
tol = 1e-14;
Lamda = 48; %Reaction coefficient
eID=4; %element ID
msh = OneDimSimpleRefinedMeshGen(0,1,5);

elemat1 = ReactionMatrix(Lamda,eID,msh);%THIS IS THE FUNCTION YOU MUST WRITE

elemat2 = [ 1 0.5; 0.5 1];
diff = elemat1 - elemat2; %calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); %calculates the total squared error between the matrices
assert(abs(diffnorm) <= tol)
```

APPENDIX B
APPENDIX B - ADVANCED SOLVER CODE

Modified and new passages of code needed to implement the advanced functionality covered in section 1b. are presented here. Note that some functions presented earlier are reused without modification and so they have not been included again. Unit tests and details of verification are also included as before. Note that code provided as part of the lecture course has not been subjected to as stringent verification as self developed functions.

A. L2 Norm and convergence analysis

```
%Script to calculate the L2 Norm of the LaplaceSolver and assess its
%convergence for verification purposes

Gaussorder = 3;
Gauss = CreateGaussScheme(Gaussorder);

%Select arbitrary x position
Xpos = 0.4;
% Select time step, 0.5 = halfway through run
Tpos = 0.5;

%Set run parameters and initialise output values
NTsteps = 1000;
NNodesvec =  [4 8 16 32 64 128];
Nruns = length(NNodesvec);
Evec = zeros(Nruns,1);
h = zeros(Nruns,1);

for idxx = 1 : Nruns % For many different run lengths
    [C,Domain,TDomain] = SolveLaplaceTransient(2,0,NNodesvec(idxx),NTsteps,'DL',0,'DL',1,'CN'); %Using
    nonGQ,linear
    Msh = OneDimLinearMeshGen(0,1,NNodesvec(idxx)-1);                                              %Basis
    functions
    for eID = 1 : NNodesvec(idxx) - 1 % For all elements

        xlims = Msh.elem(eID).x;
        x0 = xlims(1);
        x1 = xlims(2);
        c0 = C(eID,NTsteps*Tpos);
        c1 = C(eID+1,NTsteps*Tpos);

        for idx = 1 : Gaussorder % For all gauss points in each element
            J = Msh.elem(eID).J;

            %Find C, xi points
            CXi = c0*((1-Gauss.xi(idx))/2) + c1*((1+Gauss.xi(idx))/2);
            xXi = x0*((1-Gauss.xi(idx))/2) + x1*((1+Gauss.xi(idx))/2);

            %Calculate L2 norm by GQ
            Evec(idxx) = Evec(idxx) + (Gauss.wt(idx)*J*(TransientAnalyticSoln(xXi,Tpos) - CXi))^2;

        end

    end

    %Calculate element size
    h(idxx) = (x1-x0);
end

%RMS value
Evec = ((Evec).^0.5);

%Plot L2 norm convergence
loglog(NNodesvec-1,Evec)
xlabel('Number of mesh elements')
ylabel('L2 Norm')

%Calculate gradient and output to terminal
grad = (log(h(1)) - log(h(end)))/(log(Evec(1)) - log(Evec(end)));
disp(['Gradient of line: ', num2str(grad)])
```

B. Gaussian Quadrature scheme function

```
%Creates Gauss-Legendre integration weights & points for npoints
%Can create a Gauss scheme for up to 3 gauss points.
%NB - This script employs exemplar code developed and showcased by Dr
%Cookson

function [gauss] = CreateGaussScheme(npoints)

if(npoints < 1) || (npoints > 3)
 error('Gauss:argChk','Scheme not implemented.')
end
gauss.np = npoints;
gauss.wt = zeros(npoints,1);
gauss.xi = zeros(npoints,1);
if(npoints==1)
 gauss.wt(1) = 2.0;
 gauss.xi(1) = 0.0;

elseif(npoints==2)

 gauss.wt(:) = 1.0;
 gauss.xi(1) = -sqrt(1/3);
 gauss.xi(2) = sqrt(1/3);

elseif(npoints==3)
 gauss.wt(1) = 8/9;
 gauss.wt(2) = 5/9;
 gauss.wt(3) = 5/9;
 gauss.xi(1) = 0.0;
 gauss.xi(2) = -sqrt(3/5);
 gauss.xi(3) = sqrt(3/5);

end

end
```

C. Solver "Top" function, using GQ and Quadratic basis functions

```
%Function to solve Laplace's equation for given parameters of diffusion and
%reaction coefficients. As a Laplacian problem the source terms are 0.
%
%Takes the following arguments:
%
%D - Diffusion Co-efficient (Float)
%Lamda - Reaction Co-efficient (Float)
%NNodes - Number of nodes in global mesh (NElements = NNodes - 1) (Int)
%BC0 - Type of node 0 boundary condition, 'DL' for Dirichlet or 'VN' for Von
%Nuemman (Str)
%BC0Val - Value of c or dc/dx for node 0 boundary condition (Float)
%BC1 - Node 1 boundary condition, same format as BC0
%BC1Val - Value of c or dx/dx for node 1 boundary condition (Float)
%DM - Differencing Method, can take 'CN','FE','BE' (String)

```

```
%Outputs:
%C - N x N solution matrix in space and time
%Domain - Values of x that map to C
%TDomain - Values of t that map to C

%This version uses GQ and quadratic basis functions to improve on the
%flexibility and performance of the previous solver

%The solution is plotted against a known analytical solution for:
%SolveLaplaceTransient_GQ(1,-9,4,100,'DL',0,'DL',1,'CN')
%
%Note that the domain is currently hardcoded from x = 0 to x = 1

function [C, Domain, TDomain] = SolveLaplaceTransient_GQ(D,Lamda,NElements,NTsteps,BC0,BC0Val,BC1,BC1Val,DM
    )

%Set domain
xmin = 0;
xmax = 1;
%Set time scheme
%LET
tmax = 1;
dt = tmax/NTsteps;

%Define theta dependent upon the difference method selected
if DM == 'CN'
    theta = 1;
elseif DM == 'FE'
    theta = 0;
elseif DM == 'BE'
    theta = 0.5;

end

% Initialise mesh for quadratic basis functions
NNodes = 2*NElements + 1;
Mesh = OneDimLinearMeshGenGQ(xmin,xmax,NElements); % Elements is N-1 ;
%Size of global mesh effects local element values due to varying J scaling

%Initialise matrices
StiffnessMatrix = zeros(NNodes,NNodes);
MassMatrix = zeros(NNodes,NNodes);
GlobalMatrix = zeros(NNodes,NNodes);
GlobalVector = zeros(NNodes,1);
SourceVector = zeros(NNodes,1); % Source term is all 0s here
C = zeros(NNodes,NTsteps);

% Need two solutionvectors to implement timestepping
Ccurrent = zeros(NNodes,1); %Define initial conditions here
Cnext = zeros(NNodes,1);

Fcurrent = zeros(NNodes,1); %Capability for timevariant source term
Fnext = Fcurrent;

NBCcurrent = zeros(NNodes,1); %Capability for timevariant Von-Neumman
NBCnext = NBCcurrent;         %Boundary conditions

% Populate global stiffness matrix
StiffnessMatrix = GlobalStiffnessGQ(StiffnessMatrix,D,Lamda,Mesh);

% Populate a global mass matrix
MassMatrix = GlobalMassGQ(MassMatrix,Mesh);

% Combine into an overall global matrix to constitute the LHS equation
GlobalMatrix = MassMatrix + theta*dt*StiffnessMatrix;

for idxt = 1 : NTsteps
    % Assemble the global source vector
    Fnext = GlobalSourceGQ(SourceVector,Mesh);
    % Construct Previous solution, source term and boundary RHS
    PrevSolution = (MassMatrix - (1-theta)*dt*StiffnessMatrix)*Ccurrent;
    SourceNew = dt*theta*(Fnext+NBCnext);
    SourceCurrent = dt*(1-theta)*(Fcurrent + NBCcurrent);
    CombinedRHS = PrevSolution + SourceNew + SourceCurrent;

    %Enforce boundary conditions at node 0
    switch BC0
```

```
        case 'VN'
            SourceVector(1) = SourceVector(1) + -BC0Val; % Need to add for VN
        case 'DL'
            GlobalMatrix(1,:) = 0;
            GlobalMatrix(1,1) = 1;
            CombinedRHS(1) = BC0Val;
    end

    %Enforce boundary conditions at node 1
    switch BC1
        case 'VN'
            SourceVector(end) = SourceVector(end) + -BC1Val; % Need to add for VN
        case 'DL'
            GlobalMatrix(end,:) = 0;
            GlobalMatrix(end) = 1;
            CombinedRHS(end) = BC1Val;
    end

    Cnext = GlobalMatrix\CombinedRHS; % Solve equation for this timestep
    %write to output matrix for plotting, saving and analysis
    C(:,idxt) = Cnext;

    %Step time variant terms
    Ccurrent = Cnext;
    Fcurrent = Fnext;
    NBCcurrent = NBCnext;

end

%Output relevant domains for plotting, saving and analysis
Domain = linspace(xmin,xmax,NNodes);
TDomain = linspace(0,tmax,NTsteps);

end
```

D. Mesh generator

```
%This function generates a one dimensional, equispaced, linear finite
%element mesh, with Ne number of elements, between the points at x
%position xmin and xmax.

%This version has been modified to be suitable for quadratic basis
%functions


function [mesh] = OneDimLinearMeshGenGQ(xmin,xmax,Ne)

    mesh.ne = Ne; %set number of elements
    mesh.ngn = 2*Ne+1; %set number of global nodes
    mesh.nvec = zeros(mesh.ngn,1); %allocate vector to store global node values
    dx = (xmax - xmin)/Ne; %calculate element size

    mesh.nvec = xmin:dx/2:xmax; %needs to halve for internal nodes

    %loop over elements & set the element properties
    for i=1:Ne

        %set spatial positions of nodes
        mesh.elem(i).x(1) = xmin + (i-1)*dx; %sets local elements
        mesh.elem(i).x(2) = xmin + i*dx ;

        %set global IDs of the nodes
        mesh.elem(i).n(1) = i;
        mesh.elem(i).n(2) = i+1;

        %set element Jacobian based on mapping to standard element
        mesh.elem(i).J = 0.5*dx; %this is assuming standard element of -1 to 1

    end

end
```

E. Global Stiffness matrix function

```
%Function to assemble a global stiffness matrix with diffusion and
%reaction local element components calculated using GQ.

```

```
%Takes:
%StiffnessMatrix - Initialised to zeros, (N x N float)
%D - Diffusion coefficient (float)
%Lamda - Reaction coefficient (float)
%eID - Element number (int)
%msh - Mesh data structure, generated using OneDimLinearMeshGen.m or
%OneDimSimpleRefinedMeshGen.m (struct)

%Outputs:
%StiffnessMatrix - (N x N float)

function StiffnessMatrix = GlobalStiffnessGQ(StiffnessMatrix,D,Lamda,Mesh)

ScalingMat = LaplaceElemMatrixGQ(D,1,Mesh);
Matrixscaling = length(ScalingMat)-1;
NElements = Mesh.ne;
NNodes = Mesh.ngn;
eID = 1;
for idx = 1:2:NNodes - Matrixscaling

    % Generate diffusion local elements and populate global matrix
    %eID = (idx-1)/2
    LocalMatrix = LaplaceElemMatrixGQ(D,eID,Mesh);

    Matrixscaling = length(LocalMatrix)-1; %Scale the distance the LEM is
                                           %added over based on its
                                           %dimensions

    StiffnessMatrix(idx:idx+Matrixscaling,idx:idx+Matrixscaling) =...
    StiffnessMatrix(idx:idx+Matrixscaling,idx:idx+Matrixscaling) + LocalMatrix ;

    % Generate reaction local elements and populate global matrix
    LocalMatrix = ReactionMatrixGQ(D,eID,Mesh);

    Matrixscaling = length(LocalMatrix)-1;

    StiffnessMatrix(idx:idx+Matrixscaling,idx:idx+Matrixscaling) =...
    StiffnessMatrix(idx:idx+Matrixscaling,idx:idx+Matrixscaling) - LocalMatrix ;

    eID = eID + 1; %Call correct element for material parameters etc.
end
```

F. Global Mass matrix function

```
%Function to assemble a global mass matrix using Gaussian quadrature
%And quadratic basis functions
%Takes:
%MassMatrix - Initialised to zeros(N x N float)
%Mesh - Mesh data structure, , generated using OneDimLinearMeshGen.m or
%OneDimSimpleRefinedMeshGen.m (struct)

%Outputs:
%MassMatrix - (N x N float)

function MassMatrix = GlobalMassGQ(MassMatrix,Mesh)

TestMat = LocalMassMatrixGQ(1,Mesh);
Matrixscaling = length(TestMat)-1; % Generic to LEM size/Basis functions
NElements = Mesh.ne;               % used
NNodes = Mesh.ngn;
eID = 1;

for idx = 1 : 2 : NNodes-Matrixscaling

    % Generate mass local elements and populate global matrix

    LocalMatrix = LocalMassMatrixGQ(eID,Mesh);

    MassMatrix(idx:idx + Matrixscaling,idx:idx + Matrixscaling) =...
    MassMatrix(idx:idx+ Matrixscaling,idx:idx + Matrixscaling) + LocalMatrix ;

    eID = eID + 1;

end
```

G. Global Source vector function

```matlab
%Function to assemble a Global source vector from local source elements
%As demonstrated in the lecture notes

%Takes:
%SourceVector - Initialised to zeros(N x N float)
%Mesh - Mesh data structure, , generated using OneDimLinearMeshGen.m or
%OneDimSimpleRefinedMeshGen.m (struct)

%Outputs:
%SourceVector - (N x N float)

function SourceVector = GlobalSourceGQ(SourceVector,Mesh)
NNodes = Mesh.ngn;
NElements = Mesh.ne;

%Construct local source vector based on
for idx = 1 : length(SourceVector)
    if idx == 1
        SourceVector(idx) = SourceVector(idx) * Mesh.elem(1).J;
    elseif idx == length(SourceVector)
            SourceVector(idx) = SourceVector(idx) * Mesh.elem(1).J;
    else
        SourceVector(idx) = SourceVector(idx) * (Mesh.elem(1).J + Mesh.elem(1).J);
    end

end
```

H. Local Element Mass matrix function

```matlab
function LocalElementmatrix = LocalMassMatrixGQ(eID,msh)
% Calculates the local mass element matrix for element eID in the
% mesh using a GQ and quadratic basis function realisation.

%Takes:

%eID - Element ID (int)
%msh - Mesh data structure (struct)

%Outputs:
%LocalElementmatrix - 3x3 LEM of the mass term (float)

% Initialise Gauss scheme
N=3;
Gauss = CreateGaussScheme(N);
LocalElementmatrix = zeros(3); %Initialise the LEM
J = msh.elem(eID).J;

%Compute a Guassian quadrature construction for the diffusion LEM using
%Gradient DPsi/DXi of quadratic basis functions

for n = 0:2
    for m = 0:2
        for i = 1:N
            LocalElementmatrix(n+1,m+1) = LocalElementmatrix(n+1,m+1) + Gauss.wt(i)...
                *(EvalQuadBasis(n,Gauss.xi(i)) * EvalQuadBasis(m,Gauss.xi(i)));
        end
    end
end
%Scale with by the jacobian
LocalElementmatrix = LocalElementmatrix*J;
end
```

I. Local Element Diffusion matrix function

```matlab
function LocalElementmatrix = LaplaceElemMatrixGQ(D,eID,msh)
% Calculates the local diffusion element matrix for element eID in the
% mesh using a GQ and quadratic basis function realisation.

%Takes:
%D - Diffusion coefficient (float)
%eID - Element ID (int)
%msh - Mesh data structure (struct)

%Outputs:
%LocalElementmatrix - 3x3 LEM of the diffusion term (float)

% Initiate Gauss scheme
```

```
N=2;
Gauss = CreateGaussScheme(N);
LocalElementmatrix = zeros(3); %Initialise the LEM
J = msh.elem(eID).J; % Note jacobian is same for whole matrix
%Compute a Guassian quadrature construction for the diffusion LEM using
%Gradient DPsi/DXi of quadratic basis functions

for n = 0:2
    for m = 0:2
        for i = 1:N
            LocalElementmatrix(n+1,m+1) = LocalElementmatrix(n+1,m+1) + Gauss.wt(i)...
                *(QuadBasisGradient(n,Gauss.xi(i)) * QuadBasisGradient(m,Gauss.xi(i))*(1/J));
        end
    end
end

%Scale with Diffusion coefficient and Jacobian
LocalElementmatrix = LocalElementmatrix*D;
end
```

J. *Local Element Reaction matrix function*

```
%Function to calulate LocalElementmatrix for reaction term

%Takes:
%Lamda - Reaction coefficient (float)
%eID - Element number (int)
%msh - Mesh data structure, generated using OneDimLinearMeshGen.m or
%Or OneDimSimpleRefinedMeshGen.m (struct)

%Outputs
%LocalElementmatrix - 2x2 LEM of the reaction term (float)

function LocalElementmatrix = ReactionMatrixGQ(Lamda,eID,msh)
%Easiest way to generation ReactionMatrix is by scaling the mass matrix
%by lamda as already have a function for this.

LocalElementmatrix = LocalMassMatrixGQ(eID,msh) * Lamda;
end
```

K. *Quadratic basis function evaluator*

```
%Function to calulate LocalElementmatrix for reaction term

%Takes:
%Lamda - Reaction coefficient (float)
%eID - Element number (int)
%msh - Mesh data structure, generated using OneDimLinearMeshGen.m or
%Or OneDimSimpleRefinedMeshGen.m (struct)

%Outputs
%LocalElementmatrix - 2x2 LEM of the reaction term (float)

function LocalElementmatrix = ReactionMatrixGQ(Lamda,eID,msh)
%Easiest way to generation ReactionMatrix is by scaling the mass matrix
%by lamda as already have a function for this.

LocalElementmatrix = LocalMassMatrixGQ(eID,msh) * Lamda;
end
```

L. *Quadratic basis function gradient evaluator*

```
%Function to calulate LocalElementmatrix for reaction term

%Takes:
%Lamda - Reaction coefficient (float)
%eID - Element number (int)
%msh - Mesh data structure, generated using OneDimLinearMeshGen.m or
%Or OneDimSimpleRefinedMeshGen.m (struct)

%Outputs
%LocalElementmatrix - 2x2 LEM of the reaction term (float)

function LocalElementmatrix = ReactionMatrixGQ(Lamda,eID,msh)
%Easiest way to generation ReactionMatrix is by scaling the mass matrix
%by lamda as already have a function for this.

```

```
LocalElementmatrix = LocalMassMatrixGQ(eID,msh) * Lamda;
end
```

M. Quadratic basis function gradient evaluator

```
%Function to calulate LocalElementmatrix for reaction term

%Takes:
%Lamda - Reaction coefficient (float)
%eID - Element number (int)
%msh - Mesh data structure, generated using OneDimLinearMeshGen.m or
%Or OneDimSimpleRefinedMeshGen.m (struct)

%Outputs
%LocalElementmatrix - 2x2 LEM of the reaction term (float)

function LocalElementmatrix = ReactionMatrixGQ(Lamda,eID,msh)
%Easiest way to generation ReactionMatrix is by scaling the mass matrix
%by lamda as already have a function for this.

LocalElementmatrix = LocalMassMatrixGQ(eID,msh) * Lamda;
end
```

N. Full nodal comparison unit test

```
%% Test 1: Compare the GQ version of the laplace solver to the directly solved version
clear all
tol = 0.01; %(0.01% tolerance in nodal values at each timestep)
addpath G:\SimAndMod\Coursework2\AppendixA

NTsteps = 1000;

%Run the two models
[C, Domain, TDomain] = SolveLaplaceTransient(1,1,101,NTsteps,'DL',0,'DL',1,'CN');
[CGQ, DomainGQ, TDomainGQ] = SolveLaplaceTransient_GQ(1,1,50,NTsteps,'DL',0,'DL',1,'CN');

%Sum nodal values
sumC = sum(sum(C));
sumCGQ = sum(sum(CGQ));

%Average difference in solutions per timestep
TotalDelta = (sumC-sumCGQ) ;
DeltaPerT = (TotalDelta/NTsteps);
PercentageChange = DeltaPerT/sumC * 100

assert(abs(PercentageChange) < tol)
```

O. Differencing method comaparison

```
%Script to compare the error in each differencing method at different
%timepoints

%Generate a Gauss scheme
Gaussorder = 3;
Gauss = CreateGaussScheme(Gaussorder);

%Define arbitrary X position in mesh
Xpos = 0.4;

%Define parameters and sample points
NElements = 10;
NTsteps = 1000;
TposArray = [0.1  0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1];

%Instantiate vectors
Nruns = length(TposArray);
Evec = zeros(Nruns,1);
h = zeros(Nruns,1);


%L2 Norm functional only for linear basis functions
addpath G:\SimAndMod\Coursework2\AppendixA

%For all possible differencing methods
for methodidx = ["CN" "FE" "BE"]
tic
    for idxx = 1 : length(TposArray)
```

```

        [C,Domain,TDomain] = SolveLaplaceTransient(2,0,NElements,NTsteps,'DL',0,'DL',1,methodidx); %Using nonGQ,linear
        Msh = OneDimLinearMeshGen(0,1,NElements-1);                                                %Basis functions

        for eID = 1 : NElements - 1 % For all elements

            xlims = Msh.elem(eID).x;
            x0 = xlims(1);
            x1 = xlims(2);
            c0 = C(eID,NTsteps*TposArray(idxx));
            c1 = C(eID+1,NTsteps*TposArray(idxx));

            for idx = 1 : Gaussorder % For all gauss points in each element
                J = Msh.elem(eID).J;

                %Find solution, xi points
                CXi = c0*((1-Gauss.xi(idx))/2) + c1*((1+Gauss.xi(idx))/2);
                xXi = x0*((1-Gauss.xi(idx))/2) + x1*((1+Gauss.xi(idx))/2);

                %Find L2 norm
                Evec(idxx) = Evec(idxx) + (Gauss.wt(idx)*J*(TransientAnalyticSoln(xXi,TposArray(idxx)) - CXi))^2;

            end

        end

        h(idxx) = (x1-x0);
    end

    Evec = ((Evec).^0.5);

    ErrorTimeVec.(methodidx) = [Evec,TposArray'];
toc
end


%Produce plot of L2 norm for each time point
figure
hold on
plot(ErrorTimeVec.CN(:,2),ErrorTimeVec.CN(:,1),'LineWidth',1)
plot(ErrorTimeVec.BE(:,2),ErrorTimeVec.BE(:,1),'LineWidth',1)
plot(ErrorTimeVec.FE(:,2),ErrorTimeVec.FE(:,1),'LineWidth',1)
xlabel('Time')
ylabel('L2 Norm error')
legend('Crank Nicholson','Backwards Euler','Forwards Euler')
```

APPENDIX C
APPENDIX C - APPLIED MODEL CODE

All code used to perform the applied simulation as discussed in section 2 is included here. Again, ancillary scripts for debugging, plots etc. have been omitted.

A. *Solver "Top" function, using GQ and Quadratic basis functions, applied to the use case*

```
%Function to solve Laplace's equation when applied to a specific hysical problem of
%modelling heat transfer through skin tissue. Here Lamda and D are
%calculated based on the material properties of the skin at each location
%in the mesh

%Takes the following arguments:
%D - Diffusion Co-efficient (Float)
%Lamda - Reaction Co-efficient (Float)
%NNodes - Number of nodes in global mesh (NElements = NNodes - 1) (Int)
%BC0 - Type of node 0 boundary condition, 'DL' for Dirichlet or 'VN' for Von
%Nuemman (Str)
%BC0Val - Value of c or dc/dx for node 0 boundary condition (Float)
%BC1 - Node 1 boundary condition, same format as BC0
%BC1Val - Value of c or dx/dx for node 1 boundary condition (Float)
%DM - Differencing Method, can take 'CN','FE','BE' (String)
%xloc - The location in the skin that the degree of burning is to be
%       evaluated, must be in range 0<xloc<0.01 (float)

```

```
%Outputs:
%C - N x N solution matrix in space and time (float)
%Domain - Values of x that map to C (float)
%TDomain - Values of t that map to C (float)
%GammaTotal - The total burn damage suffered (float)


%E.g.
%[C, Domain, TDomain] = SolveLaplaceTransient_GQ_p2(52,100,'DL',393.75,'DL',310.15,'CN',0.05)


function [C, Domain, TDomain, GammaTotal] = SolveLaplaceTransient_GQ_p2_1(NElements,NTsteps,BC0,BC0Val,BC1,
    BC1Val,DM,xloc)

%Set domain for range of distance in skin
xmin = 0;
xmax = 0.01;

%Set time scheme
tmax = 50;
dt = tmax/NTsteps;

%Set initial temperature condition to standard internal body temperature
Tstart = 310.15;

%Define theta dependent upon the difference method selected
if DM == 'CN'
    theta = 1;
elseif DM == 'FE'
    theta = 0;
elseif DM == 'BE'
    theta = 0.5;

end



%Initialise mesh
NNodes = 2*NElements + 1;
Mesh = OneDimLinearMeshGenGQ(xmin,xmax,NElements);

%Add material parameters to mesh data structure
Mesh = EnhanceMeshData(Mesh,0,1);

%Run the model with the assumption of 0 blood flow if desired
%Mesh.G(:) = 0;

%Initialise neccesary matrices
StiffnessMatrix = zeros(NNodes,NNodes);
MassMatrix = zeros(NNodes,NNodes);
GlobalMatrix = zeros(NNodes); % Combination of the two
GlobalVector = zeros(NNodes,1);
SourceVector = zeros(NNodes,1);
GammaT = zeros(NTsteps,1);
C = zeros(NNodes,NTsteps);

% Need two solutionvectors to implement timestepping
Ccurrent = zeros(NNodes,1) + Tstart; %Define initial conditions here
Cnext = zeros(NNodes,1);

Fcurrent = zeros(NNodes,1); % Initialise source term
Fnext = Fcurrent;

NBCcurrent = zeros(NNodes,1); %Capability for timevariant Von-Neumman
NBCnext = NBCcurrent;         %Boundary conditions

%Step through time
for idxt = 1 : NTsteps

    %Re-initialise matrices
    StiffnessMatrix = zeros(NNodes);
    MassMatrix = zeros(NNodes);
    GlobalMatrix = zeros(NNodes);
    GlobalVector = zeros(NNodes,1);

    % Populate global stiffness matrix
    StiffnessMatrix = GlobalStiffnessGQ_p2(StiffnessMatrix,Mesh);
    % Populate a global mass matrix
    MassMatrix = GlobalMassGQ(MassMatrix,Mesh);

    % Combine into an overall global matrix to constitute the LHS equation
    GlobalMatrix = MassMatrix + theta*dt*StiffnessMatrix;

    %Assemble the global source vector
    Fnext = GlobalSourceGQ_p2(SourceVector,Mesh);

    % Construct Previous solution, source term and boundary RHS
    PrevSolution = (MassMatrix - (1-theta)*dt*StiffnessMatrix)*Ccurrent;
    SourceNew = dt*theta*(Fnext+NBCnext);
    SourceCurrent = dt*(1-theta)*(Fcurrent + NBCcurrent);
    CombinedRHS = PrevSolution + SourceNew + SourceCurrent;

    %Enforce boundary conditions at node 0
    switch BC0
        case 'VN'
            SourceVector(1) = SourceVector(1) + -BC0Val; % Need to add for VN
        case 'DL'
            GlobalMatrix(1,:) = 0;
            GlobalMatrix(1,1) = 1;
            CombinedRHS(1) = BC0Val;
    end

    %Enforce boundary conditions at node 1
    switch BC1
        case 'VN'
            SourceVector(end) = SourceVector(end) + -BC1Val; % Need to add for VN
        case 'DL'
            GlobalMatrix(end,:) = 0;
            GlobalMatrix(end) = 1;
            CombinedRHS(end) = BC1Val;
    end

    Cnext = GlobalMatrix\CombinedRHS;
    C(:,idxt) = Cnext;

    %Assess if a burn may occur
    %Need to interpolate within nodes for high accuracy assessment
    Cinterp = QuadInterpolate(Cnext,Mesh,xloc);

    %If a burn may occur add its damage contribution to the integral
    if Cinterp > 317.15
        GammaT(idxt) = 2*10^98*exp(-12017/(Cinterp - 273.15));
    end

    %Step time variant terms
    Ccurrent = Cnext;
    Fcurrent = Fnext;

end

%Use inbuilt function to integrate GammaT burn damage
%and output to terminal
GammaTotal = trapz(GammaT) * dt;
disp(['Burn damage of ' num2str(GammaTotal)])

%Output relevant domains for plotting, saving and analysis
Domain = linspace(xmin,xmax,NNodes);
TDomain = linspace(0,tmax,NTsteps);

end
```

B. Function to add material properties to mesh structure

```
%Script to enhance the mesh data structure to include material parameters
%This will allow discontinuities to be handled such as in part 2 of the
%problem
%Needs to map k,G,rho,c,rhob,cb,Tb
%Assumes that xmin is defined as 0

%Takes:
%msh - Mesh data structure (struct)
%xmin - Lower limit of x domain (float)
%xmax - Upper limit of x domain (float)

```

```
%Outputs:
%msh - Expanded upon datastructure containing material properties of a
%       specific skin model (struct)

function msh = EnhanceMeshData(msh,xmin,xmax)

%Shift scale into positive region
xmax = xmax - xmin
xmin = 0

%Define location of layers of skin
Epos = 0.00166667;
Dpos = 0.005;
Bpos = 0.01;

%Need to convert these values to be scaled up between 0 and 1 to operate
%over simple domain

Epos = Epos * xmax/Bpos;
Dpos = Dpos * xmax/Bpos;
Bpos = Bpos * xmax/Bpos;

%Define parameter values from table

k = [25 40 20];
G = [0 0.0375 0.0375];
rho = [1200 1200 1200];
c = [3300 3300 3300];
rhob = [0 1060 1060];
cb = [0 3770 3770];
Tb = [0 310.15 310.15];

%Load values into the mesh data structure by their element id

%Node positions
Enode = ceil(msh.ne*Epos) ;
Dnode = ceil(msh.ne*Dpos);
Bnode = ceil(msh.ne*Bpos);

%Populate Epidermis segmemt of the mesh
for idx = 1 : Enode
    msh.k(idx) = k(1);
    msh.G(idx) = G(1);
    msh.rho(idx) = rho(1);
    msh.c(idx) = c(1);
    msh.rhob(idx) = rhob(1);
    msh.cb(idx) = cb(1);
    msh.Tb(idx) = Tb(1);
end

%Populate Dermis segment of the mesh
for idx = Enode : Dnode
    msh.k(idx) = k(2);
    msh.G(idx) = G(2);
    msh.rho(idx) = rho(2);
    msh.c(idx) = c(2);
    msh.rhob(idx) = rhob(2);
    msh.cb(idx) = cb(2);
    msh.Tb(idx) = Tb(2);
end

%Populate remaining segment of the mesh (To the body)
for idx = Dnode : Bnode
    msh.k(idx) = k(3);
    msh.G(idx) = G(3);
    msh.rho(idx) = rho(3);
    msh.c(idx) = c(3);
    msh.rhob(idx) = rhob(3);
    msh.cb(idx) = cb(3);
    msh.Tb(idx) = Tb(3);
end
```

C. *Script to calculate the amount of temperature reduction needed to avoid burns*

```
%Script to calculate the neccesary temperature reduction to avoid a burn
%Set stepping increment for search algorithm

```

```
%Define a temperature delta to step through.
%This is the precision this script is searching to
TempStep = 0.1;

% at the Epidermis, Gamma = 1 for second degree burn
GammaTotal = 1;
TempReduce = 50;
xloc = 0.00166667;

%Run until gamma is reduced below 1
while GammaTotal >= 1
    [Cplot, Domain, TDomain, GammaTotal] = SolveLaplaceTransient_GQ_p2_1(52,100,'DL',393.75-TempReduce,'DL'
    ,310.15,'CN',xloc);
    TempReduce = TempReduce + TempStep;
end
Epidermisburn = TempReduce

% at the Dermis, Gamma = 1 for third degree burn
GammaTotal = 1;
TempReduce = 40;
xloc = 0.005;

%Run until gamma is reduced below 1
while GammaTotal >= 1
    [Cplot, Domain, TDomain,GammaTotal] = SolveLaplaceTransient_GQ_p2_1(52,100,'DL',393.75-TempReduce,'DL'
    ,310.15,'CN',xloc);
    TempReduce = TempReduce + TempStep;
end
Dermisburn = TempReduce
```

D. *Function to Interpolate between nodes with quadratic basis functions*

```
% Function to interpolate a solution between nodal values using quadratic
% basis functions

%Takes:
%C - Full solution at that time (1 x N float)
%msh - Enhanced mesh data structure (struct)
%xloc - Location within the skin/x domain (float)

function InterpC = QuadInterpolate(C,msh,xloc)

NNodes = msh.ngn;

% Need to use basis functions to accurately interpolate between nodes
xmax = msh.nvec(end);
xpos = xloc/xmax; %The exact position on the mesh

%Find which nodes this is between
xlower = floor(NNodes*xpos);
xupper = ceil(NNodes*xpos);

if xlower == xupper
   InterpC = C(xlower);
   return
end

%Find x value at these nodes
x0 = msh.nvec(xlower);
x1 = msh.nvec(xupper);

%Find xi in this element
xi = 2*(xloc-x0)/(x1-x0) -1;

%Recall Cs at this position
c0 = C(xlower);
c2 = C(xupper);
c1 = (c0 + c2)/2;

%Use quadratic basis functions to interpolate
InterpC = c0 *(xi*(xi-1))/2 + c1 * (1-xi^2) + c2 * (xi*(xi+1))/2;
```

E. *Function to construct global source vector*

```
%Function to assemble a Global source vector from local source elements
%As demonstrated in the lecture notes

```

```
%Takes:
%SourceVector - Initialised to zeros(N x N float)
%Mesh - Mesh data structure, , generated using OneDimLinearMeshGen.m or
%OneDimSimpleRefinedMeshGen.m (struct)

%Outputs:
%MassMatrix - (N x N float)

function SourceVector = GlobalSourceGQ_p2(SourceVector,Mesh)
NNodes = Mesh.ngn;
NElements = Mesh.ne;
eID = 1;
for idx = 1 : 2 : NNodes - 2
    SourceVector(idx:idx+2) = SourceVector(idx:idx+2) - LocalSourceGQ(eID,Mesh);
    eID = eID + 1;
 end
```

F. Function to calculate local source vector

```
%Function to calculate the local source vector for each element based on
%material properties in the skin

%Takes:
%eID - Element ID (int)
%msh - Enhanced mesh data structure (struct)

%Outputs:
%LocalElementmatrix - 3x1 local source vector (float)

function localsource = LocalSourceGQ(eID,msh)

%Initialise a 3x1 vector
localsource = zeros(3,1);

%Determine source terms based on location within Mesh and populate 1D vector
F = (msh.rhob(eID) * msh.cb(eID) * msh.G(eID))/(msh.rho(eID)*msh.c(eID)) * msh.Tb(eID);
J = msh.elem(eID).J;
localsource(:) = F*J;
```

G. Function to construct global stiffness matrix

```
%Function to assemble a global stiffness matrix with diffusion and
%reaction local element components calculated using GQ. Modified for
%application to the skin problem

%Takes:
%StiffnessMatrix - Initialised to zeros, (N x N float)
%eID - Element number (int)
%msh - Enhanced mesh data structure, generated using OneDimLinearMeshGen.m
% and enhanced with EnhanceMeshData(struct)

%Outputs:
%StiffnessMatrix - (N x N float)

function StiffnessMatrix = GlobalStiffnessGQ_p2(StiffnessMatrix,Mesh)

%Determine the size of the local elements being operated upon
ScalingMat = LaplaceElemMatrixGQ_p2(1,Mesh);
Matrixscaling = length(ScalingMat)-1;
%Extract mesh characteristics
NElements = Mesh.ne;
NNodes = Mesh.ngn;
eID = 1;
for idx = 1:2:NNodes - Matrixscaling

    % Generate diffusion local elements and populate global matrix
    LocalMatrix = LaplaceElemMatrixGQ_p2(eID,Mesh);

    Matrixscaling = length(LocalMatrix)-1; %Scale the distance the LEM is
                                           %added over based on its
                                           %dimensions

    StiffnessMatrix(idx:idx+Matrixscaling,idx:idx+Matrixscaling) =...
    StiffnessMatrix(idx:idx+Matrixscaling,idx:idx+Matrixscaling) + LocalMatrix ;

    % Generate reaction local elements and populate global matrix
    LocalMatrix = ReactionMatrixGQ_p2(eID,Mesh);

    Matrixscaling = length(LocalMatrix)-1;

    StiffnessMatrix(idx:idx+Matrixscaling,idx:idx+Matrixscaling) =...
    StiffnessMatrix(idx:idx+Matrixscaling,idx:idx+Matrixscaling) - LocalMatrix ;

    eID = eID + 1; %Call correct element for material parameters etc.
end
```

H. Function to calculate local element diffusion matrix

```
% Calculates the local diffusion element matrix for element eID in the
% mesh using a GQ and quadratic basis function realisation.

%Takes:
%D - Diffusion coefficient (float)
%eID - Element ID (int)
%msh - Mesh data structure (struct)

%Outputs:
%LocalElementmatrix - 3x3 LEM of the diffusion term (float)
function LocalElementmatrix = LaplaceElemMatrixGQ_p2(eID,msh)

% Initiate variables and Gauss scheme
N=2;
Gauss = CreateGaussScheme(N);
LocalElementmatrix = zeros(3);
J = msh.elem(eID).J;

%Calculate D based on the location within the Mesh
D = msh.k(eID)/(msh.rho(eID)*msh.c(eID));

%Compute a Guassian quadrature construction for the diffusion LEM using
%Gradient DPsi/DXi of quadratic basis functions
for n = 0:2
    for m = 0:2
        for i = 1:N
            LocalElementmatrix(n+1,m+1) = LocalElementmatrix(n+1,m+1) + Gauss.wt(i)...
                *(QuadBasisGradient(n,Gauss.xi(i)) * QuadBasisGradient(m,Gauss.xi(i))*(1/J));
        end
    end
end

%Scale with Diffusion coefficient and Jacobian
LocalElementmatrix = LocalElementmatrix*D;
end
```

I. Function to calculate local element reaction matrix

```
%Function to calculate the local reaction matrices for each element
%This version scales by a lamda that is dynamically calculated along the
%mesh

%Takes:
%eID - Element ID (int)
%msh - Mesh data structure (struct)

%Outputs:
%LocalElementmatrix - 3x3 LEM of the reaction term (float)

function LocalElementmatrix = ReactionMatrixGQ_p2(eID,msh)

%Easiest way to generation ReactionMatrix is by scaling the mass matrix
%by lamda as already have a function for this.

%Extract mesh material properties to calculate lamda
Lamda = (msh.G(eID)*msh.rhob(eID)*msh.cb(eID))/(msh.rho(eID)*msh.c(eID));
LocalElementmatrix = LocalMassMatrixGQ(eID,msh) * Lamda;
end
```