ME40064: Systems Modelling and Simulations

Summative Assignment: Transient MATLAB-Based FEM Modelling

6th December 2024

Candidate number: 11669

Word Count: 2990

UNIVERSITY OF BATH

Table of Contents

Artificial Intelligence Statement

Artificial Intelligence (AI) was used in proof-reading the final draft of this report and to partially assist in generating ideas. Ideas that have been generated with AI have been referenced as necessary. No Artificial Intelligence was used during the research and writing portions of this document, or in generating code. The student acknowledges that this work is their own, and that ChatGPT-4 was used to assist in proofreading (OpenAI, 2024).

Introduction

Equation 1: The transient diffusion-reaction-source equation (Cookson, 2024)

$$\frac{\partial C}{\partial t} = D\frac{\partial C^2}{\partial x^2} - \lambda\frac{\partial C}{\partial x} + f$$

This report, split into two parts, documents the verification and application of MATLAB code in modelling a system governed by the transient diffusion-reaction-source equation (Equation 1). For this project, MATLAB 2023a was used (The MathWorks, 2023).

'Part one: Base code' contains documentation relating to the description and verification of the core code structures used throughout this project. In 'Part two: Tissue Modelling', this code is adapted and used to model heat transfer within skin tissue, quantifying when second-and third-degree burns occur. The effect of applying insulation to the skin is considered, as well as the effect of including a layer air between insulation and skin.

PART 1: BASE CODE

Code Architecture and key features

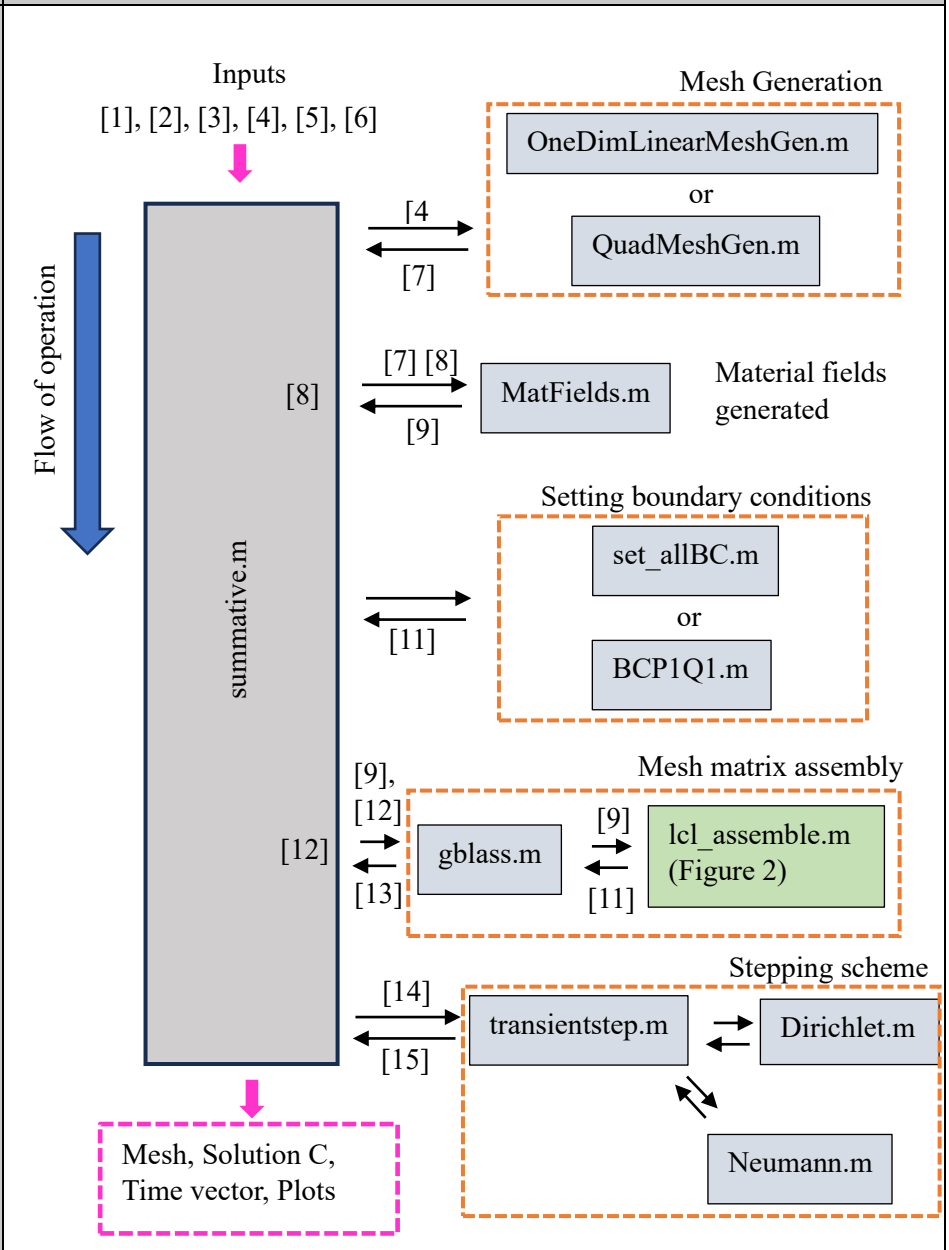
Flow DiagramVariables	Representative flow diagram[1]
[1] theta [2] solution order (linear or quadratic mesh?) [3] Default values: use PCP1Q1.m? [4] no. elements in mesh [5] time vector [6] plt? [7] msh [8] Material parameter functions in form f(x, t) [9]msh updated with material parameters [10] Boundary conditions [11] local matrices for all elements at given time [12] Initial conditions [13] Global matrices for mesh at given time (stiffness, mass, diffusion, etc. [14] Current solution C0 [15] Solution in next step in time C1	

Figure 1: Flow diagram detailing core code architecture.

[1] NB: This is a simplified diagram, and not an exact representation. For more detail, view source code

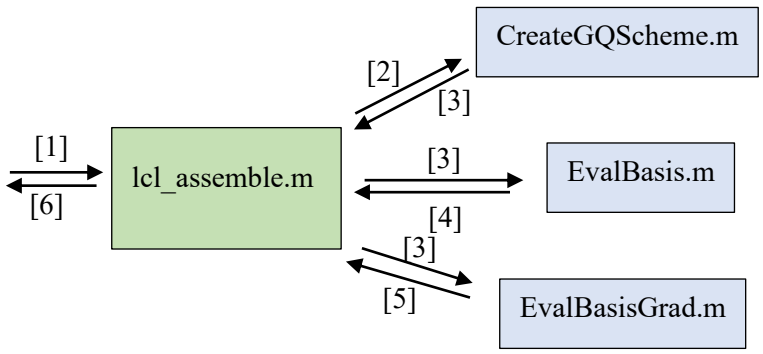
Variables	Representative flow diagram
[1] mesh updated with material parameters [2] Gauss Scheme order N [3] Gaussian Quadrature Scheme [4] psi values [5] gradient values [6] local matrices for a given element at given time	

Figure 2: Code architecture flow diagram continued

The flow diagrams in Figure 1 and 2 shows how the key functions within the MATLAB code interact with another, as well as where various variables are created and moved.

List of functions

Provided in Tables 1 and 2 is a full list of MATLAB files used in Part 1 of this assignment.

Table 1: List of functions associated with Part 1 of assignment, arranged in alphabetical order

Function (.m file)	Description	Name of test script (.m file)	Appendix
Core code			
BCP1Q1.m	Sets default conditions of P1Q1 of assignment	-	A.1
CreateGQScheme.m	Generates Gaussian Quadrature scheme	CreateGQSchemeTest.m	A.2
Dirichlet.m	Applies Dirichlet conditions	-	A.3
EvalBasis.m	Evaluates basis function values	-	A.4
EvalBasisGrad.m	Evaluates basis function gradient values	-	A.5
gblass.m	Assembles global matrices	-	A.6
generateL2Plot.m	For generating L2 convergence plot	-	A.7
L2.m	Calculates L2 error at a given timestep	-	A.8
lcl_assemble.m	Assembles local matrices	lcl_assembleUnitTest.m	A.9
MatFields.m	Generates material fields	MatFieldsUnitTest.m	A.10
Neumann.m	Applies Neumann boundary conditions	-	A.11
plot_C_atT.m	For generating figure plots	-	A.12
plot_C_atX.m	For generating figure plots	-	A.13
QuadMeshGen.m	Quadratic mesh generation	QuadMeshGenUnitTest.m	A.14
set_all_BC.m	UI interface for setting boundary conditions	-	A.15
setDBC.m		-	A.16
setNBC.m		-	A.17
summative.m	Primary interface with user when using model	-	A.18
transientstep.m	Calculates solution vector at next timestep	-	A.19
OneDimLinearMeshGen.m	Generates a linear mesh		A.27

Table 2: List of functions associated with Part 1 of assignment continued: core code verification functions.

Function	Description	Name of test script (.m file)	Appendix
Core code verification			
DecayReactionAnalyticSoln.m	Analytic solution: Test Case 4	-	A.20
DiffusionWithDecayAnalyticSoln.m	Analytic solution: Test case 2	-	A.21
DiffusionWithDecayingReacionAnalyticSoln	Analytic solution: Test case 3	-	A.22
EvalBCAnal.m	Stores various types of boundary conditions for test cases	-	A.23
materialfuncs.m	Stores material function definitions for test cases	-	A.24
PureSourceAnalyticSoln.m	Analytic solution: Test case 1	-	A.25
TransientAnalyticSoln.m	Analytic solution: Assignment task	-	A.26

P1Q1: Minimum Requirements

a. Q1a. and mesh resolution analysis

Q1a. Plot your solution c(x) vs. x, showing the solutions at t = 0.05, 0.1, 0.3, 1.0, in the format shown in Lecture 13, comparing it to the analytical solution.

In accordance with the assignment, Figure 3 through to Figure 5 were plotted using the function plotC_atX.m (Appendix A.13), showing the solution vector C across the mesh at the times 0.05s, 0.1s, 0.3s, and 1s. For these plots, quadratic meshes were used with a timestep dt of 0.01s.

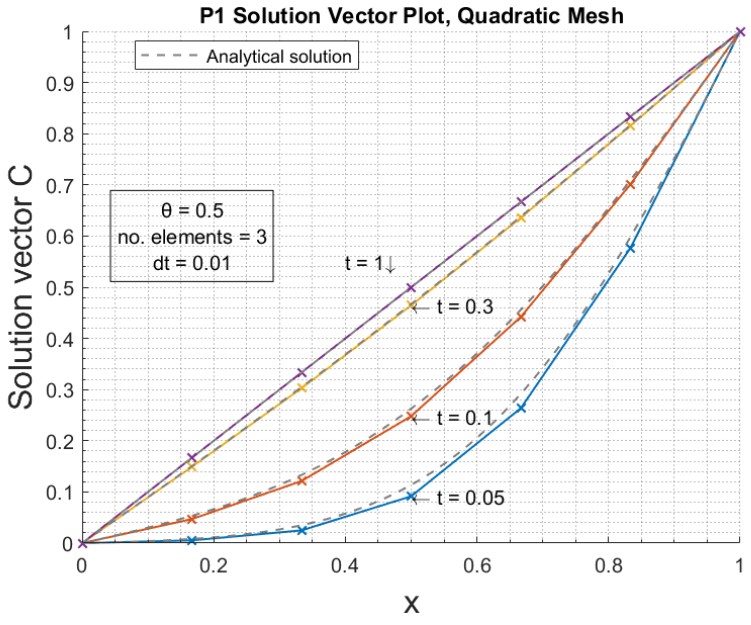


Figure 3: Plot of the solution vector produced using a 3-element quadratic-basis-function mesh (dt = 0.01s, Crank-Nicolson Scheme).

Initially, a 3-element mesh was used (Figure 3). Whilst the results produced were comparable to the analytical solution, they were noticeably discrete.

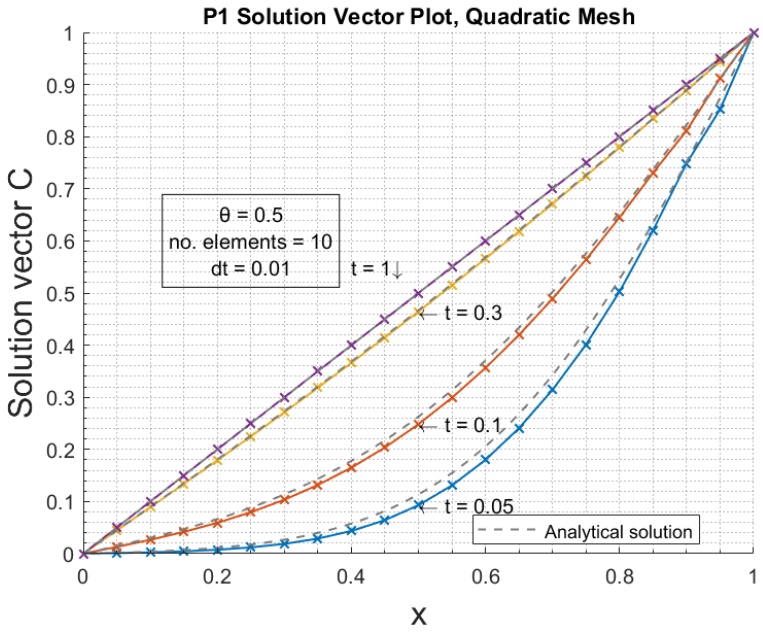


Figure 4: Plot of the solution vector produced using a 10-element quadratic-basis-function mesh (dt = 0.01s, Crank-Nicolson Scheme.

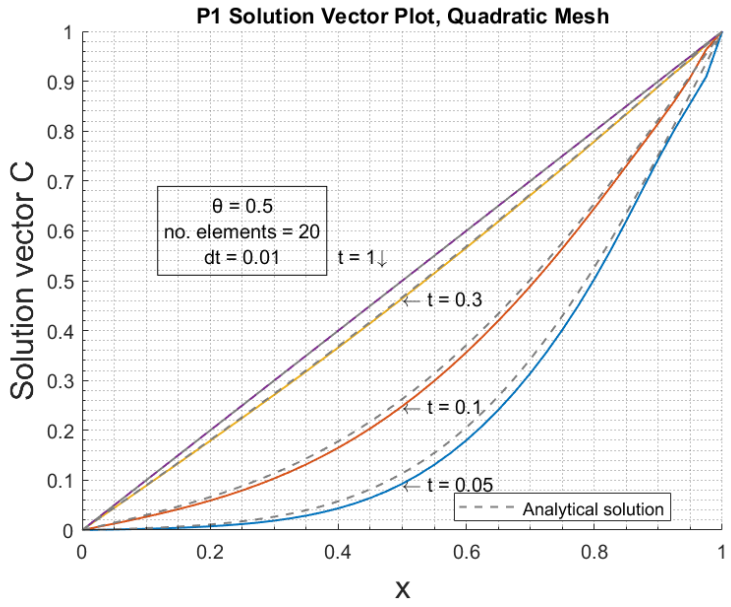


Figure 5: Plot of the solution vector produced using a 20-element quadratic-basis-function mesh (dt = 0.01s, Crank-Nicolson Scheme).

Increasing the number of elements reduces this. Figures 4 and 5, corresponding 10 and 20 element meshes respectively, show that as element size decreases, the solution vector converges towards the true analytical solution.

However, there remains a noticeable gap between the numerical and analytical solutions, most distinct at plots for t = 0.05 s and 0.1 s. This is a feature of the Crank-Nicolson time-stepping scheme that was used during simulation.

Table 3: Relationship between number of elements, element size dx, and computational time to run the simulation.

No. elements	Element size dx	Computational time using a quadratic-basis function mesh and dt of 0.01s (s)
3	0.33	0.11
6	0.17	0.17
10	0.1	0.32
20	0.05	0.48
40	0.025	0.80
80	0.0125	1.62

For mesh sizes of 20 elements and below, computational time remained relatively small (<0.5 s), where increasing mesh resolution having little computational cost.

Beyond this, further refinement resulted in an exponential increase in computational time. This is captured in Table 3, where past an element size dx of 0.05, halving element size roughly doubles computational time. For this reason, a ten-element mesh was used for the model in Part 1 of this assignment.

Q1b and timestep analysis

Q1b. Plot both the analytical & your numerical solutions at x=0.8, for t = 0 to 1.0.

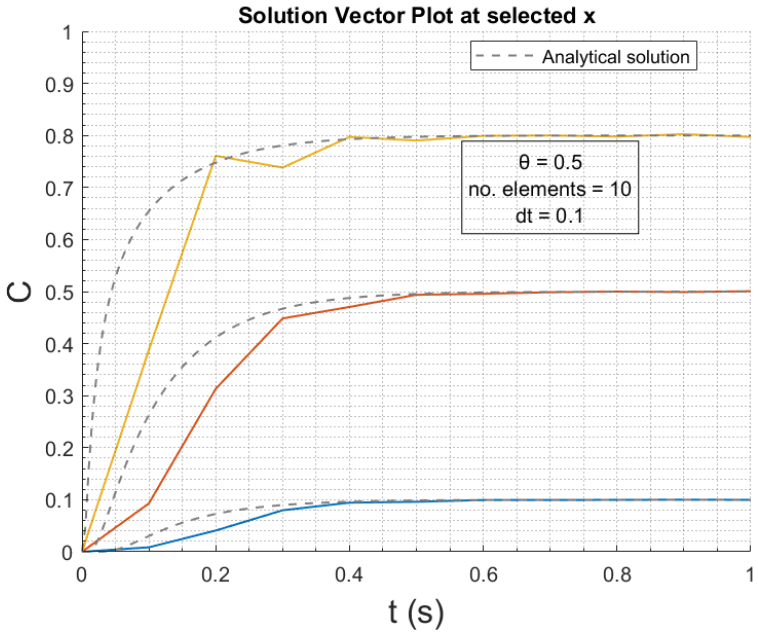


Figure 6: Plot of solution at x = 0.8, 0.5, and 0.1 in a five element quadratic-basis-function element mesh (dt = 0.1s, Crank Nicolson time-stepping scheme).

In accordance with the task, Figure 6 was generated using the function PlotC_atT.m (A.12), plotting the solution vector against time at x = 0.8, 0.5, and 0.1. Initially, a relatively large timestep dt = 0.1s and element size dx = 0.1 was used.

Whilst the model converged towards the analytical solution, it took an amount of time before they become comparable. Across the three x values, settling time ranged between 0.2-0.5s (2-5 timesteps respectively). With this taking 30-50% of the total simulation runtime, it was deemed significant.

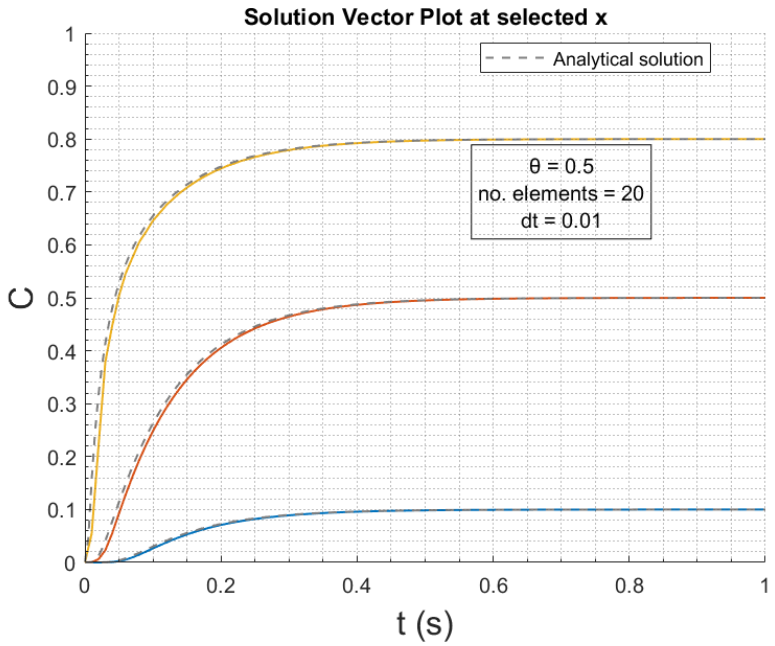


Figure 7: Plot of solution at x = 0.8, 0.5, 0.1 in a twenty-element quadratic element quadratic-basis-function element mesh (dt = 0.01s, Crank Nicolson time-stepping scheme).

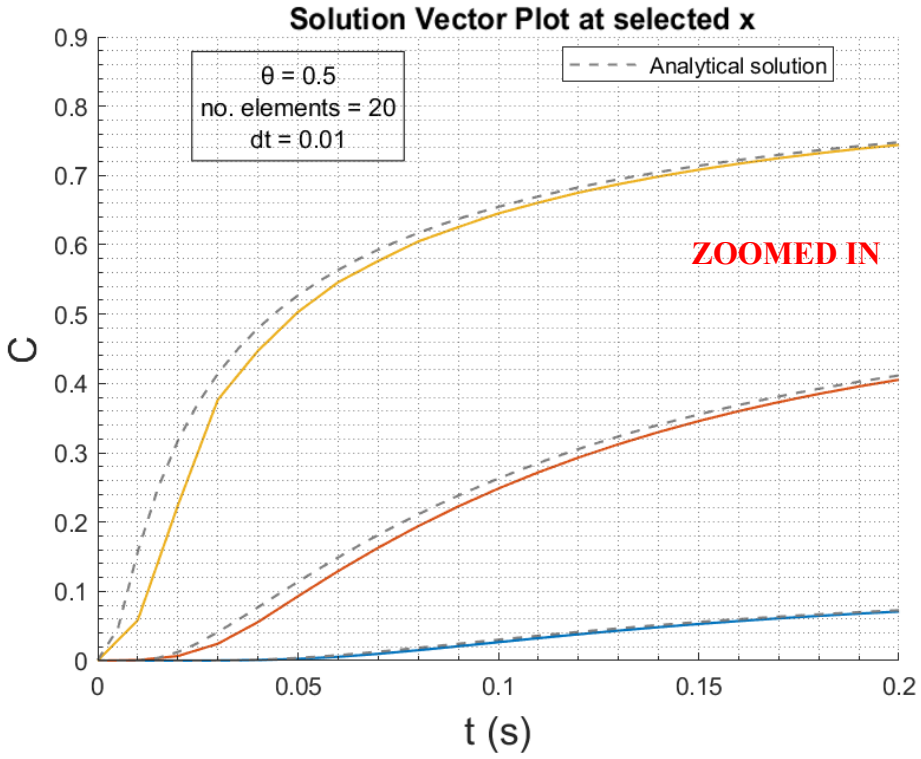


Figure 8: Zoomed in plot of previous Figure.

In an effort to minimise this, the simulation was repeated with a timestep one magnitude smaller, dt = 0.01s, and halved element size, dx = 0.05. This is shown in Figures 7 and 8.

Under these parameters, the model converged towards the analytical solution more smoothly and quickly. Settling time was estimated to range between 0.05-0.15s, or 5-15 timesteps. At 5-15% of total simulation runtime, this indicated better simulation quality.

Table 4: Computational time and est. settling time of simulation for P1Q1.

dt (s)	Computational time for a 5 element quadratic-basis function mesh (s)	Est. settling time		
		Abs. value (s)	No. timesteps	Percentage of simulation time (%)
0.1	0.10	0.30 - 0.50	3 - 5	30 - 50%
0.05	0.14	0.40 – 0.50	8 - 10	40 - 50%
0.01	0.43	0.05 - 0.15	5 - 15	5 - 15%
0.005	0.80	0.05 - 0.10	10 - 20	5 - 10%
0.001	3.76	0.01 – 0.05	10 - 50	1 - 5%

Reducing dt beyond this further reduced settling time, but at higher computational cost.

Viewing Table 4, decreasing dt from 0.01 s to 0.005 s resulted in an 86% increase in computational time with minimal reductions in settling time. Therefore, for simulations where settling time is not critical (e.g., steady-state analysis), a timestep of 0.01 s may be sufficient using this model.

For applications requiring further reduced settling times, decreasing dt to 0.001 s reduces settling time to approximately 0.05 s, 5% of total simulation runtime. This finer timestep may be more appropriate when investigating transient system behaviour immediately after applying boundary conditions, or for simulations lasting very short durations. However, it incurs a substantial computational cost, with a timestep reduction from 0.005 s to 0.001 s corresponding to a 373% increase in computational time.

P1Q2: Additional Model Verification

Table 5: List of governing equations used to assist model verification.

Case	Description	Governing equation	Analytical solution
1	Pure Source	$\frac{\partial C}{\partial t} = f$	$C(t) = ft$
2	Diffusion with Decay	$\frac{\partial C}{\partial t} = D\frac{\partial C^2}{\partial x^2} - \lambda C$	$C(t) = \sin(\frac{\pi x}{L})e^{-\left(\lambda + D\frac{\pi^2}{L^2}\right)t}$
3	Diffusion with exponentially decaying Reaction	$\frac{\partial C}{\partial t} = D\frac{\partial C^2}{\partial x^2} - \lambda e^{-\beta t}C$	$C(t) = C_0 e^{-\frac{\lambda}{\beta}(1-e^{-\beta t})}$
4	Decaying reaction	$\frac{\partial C}{\partial t} = -\lambda C$	$C(t) = C_0 e^{-\lambda t}$

Table 6: Function handles corresponding to analytic solutions.

Cases	Description	MATLAB function of analytic solution	Appendix
1	Pure Source	PureSourceAnalyticSoln.m	A.26
2	Diffusion with Decay	DiffusionWithDecayAnalyticSoln.m	A.22
3	Diffusion with exponentially decaying Reaction	DiffusionWithDecayingReactionAnalyticSoln.m	A.23
4	Decaying reaction	DecayReactionAnalyticSoln.m	A.21

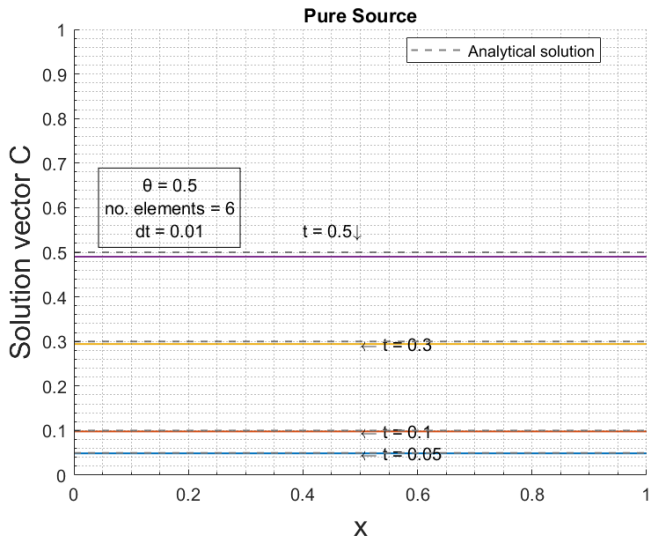
In addition to the governing equation defined in P1Q1, four additional test cases were modelled. Governing equations, shown in table 6, were ideated with the aid of artificial Intelligence ChatGPT -4 (OpenAI, 2024) and subsequently used to independently generate MATLAB code.

Each case was explored analogously to P1Q1, aiming to verify that material parameters and boundary conditions were integrated within the code correctly. For each case, a timestep of 0.01s and element size of 0.05 was used, as found to be optimal for computational time previously.

Where relevant, material parameter and boundary condition values of 1 and 0 were used to simplify analytic solutions. For example, the simulation capturing pure source (Case 1) had a source term f value of one and Neumann boundary conditions $\mathrm{x}(0,\mathrm{t}) = \mathrm{x}(1,\mathrm{t}) = 0$.

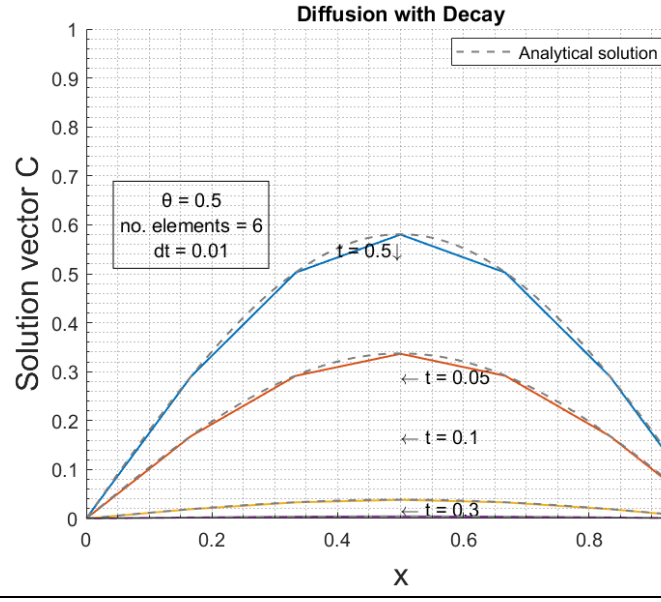
Case 1: Pure Source

Table 7: Equations, parameters, and plot relating to model verification Case 1: Pure Source.

Governing equation	Plot
$\frac{\partial C}{\partial t} = f$	
Analytical solution	
$C(t) = ft$	
Boundary conditions	
$\frac{\partial C}{\partial t}(0,t) = 0, \quad \frac{\partial C}{\partial t}(1,t) = 0$	
Equivalent Parameters	
$D = 0$	
$\lambda = 0$	
$f = 1$	

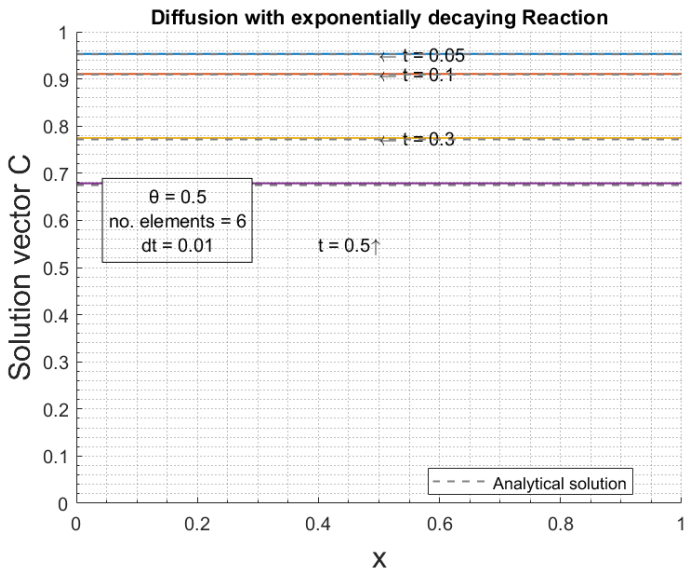
Case 2: Diffusion with Decay

Table 8: Equations, parameters, and plot relating to model verification Case 2: Diffusion with Decay.

Governing equation	Plot
$\frac{\partial C}{\partial t} = D\frac{\partial C^2}{\partial x^2} - \lambda C$	
Analytical solution	
$C(t) = \sin\left(\frac{\pi x}{L}\right)e^{-\left(\lambda + D\frac{\pi^2}{L^2}\right)t}$	
Boundary conditions	
$C(0,t) = 0, \quad C(1,t) = 0$	
Equivalent Parameters	
$D = 1$	
$\lambda = 1$	
$f = 0$	

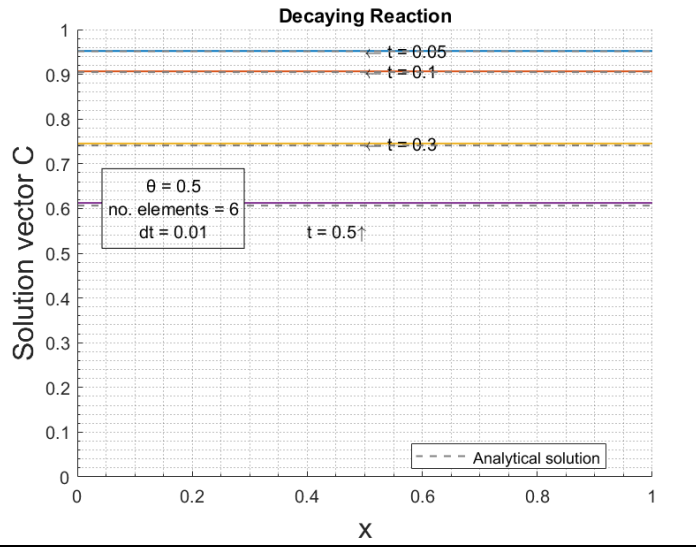
Case 3: Diffusion with Exponentially Decaying Reaction

Table 9: Equations, parameters, and plot relating to model verification Case 3: Diffusion with Exponentially Decaying Reaction.

Governing equation	Plot
$\frac{\partial C}{\partial t} = D\frac{\partial C^2}{\partial x^2} - \lambda e^{-\beta t} C$	
Analytical solution	
$C(t) = C_0 e^{-\frac{\lambda}{\beta}(1-e^{-\beta t})}$	
Boundary conditions	
$\frac{\partial C}{\partial t}(0,t) = 0, \quad \frac{\partial C}{\partial t}(1,t) = 0$	
Equivalent Parameters	
$D = 1$	
$\lambda = e^{-t}$	
$f = 0$	

Case 4: Decaying Reaction

Table 10: Equations, parameters, and plot relating to model verification Case 4: Decaying Reaction.

Governing equation	Plot
$\frac{\partial C}{\partial t} = -\lambda C$	
Analytical solution	
$C(t) = C_0 e^{-\lambda t}$	
Boundary conditions	
$\frac{\partial C}{\partial t}(0,t) = 0, \quad \frac{\partial C}{\partial t}(1,t) = 0$	
Equivalent Parameters	
$D = 0$	
$\lambda = 1$	
$f = 0$	

Validation

Within the tested cases, the model consistently produced results closely matching analytical solutions, validating the correct integration of diffusion, reaction, and source terms, as well as boundary conditions.

Extra Credit and key features

Analysis of Forward Euler, Backward Euler, and Crank-Nicolson time-stepping schemes

Within the time-stepping function transientstep.m (Appendix A.19), there is flexibility to use Forward Euler, Backward Euler, or Crank Nicolson stepping schemes. These correspond to input theta values of 0, 1, and 0.5 respectively.

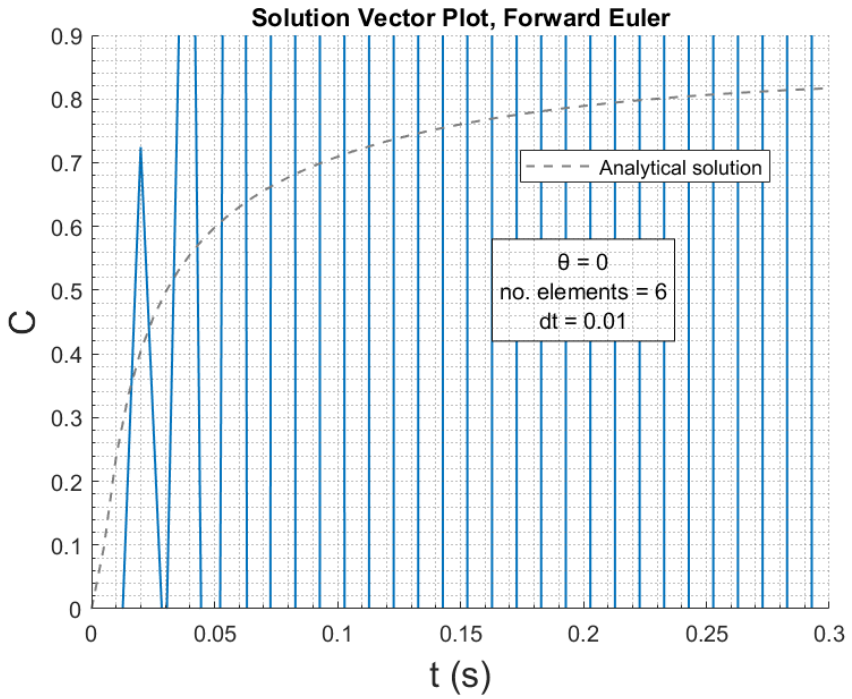


Figure 9: Solution vector plot at x = 0.8 depicting the Forward Euler stepping scheme, $dt = 0.01$.

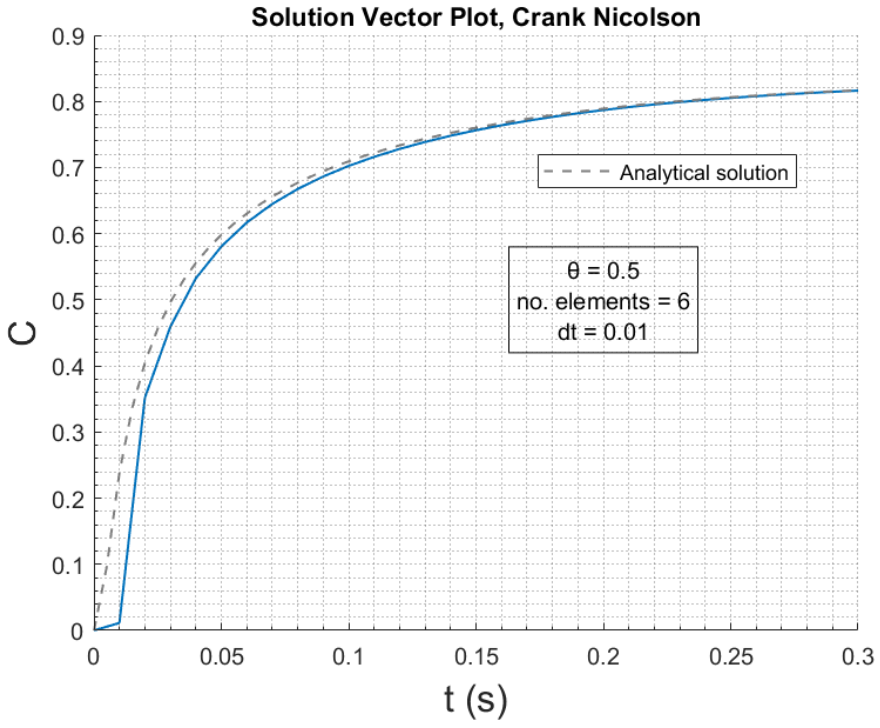


Figure 10: Solution vector plot at x = 0.8 depicting the Crank Nicolson stepping scheme, $dt = 0.01$.

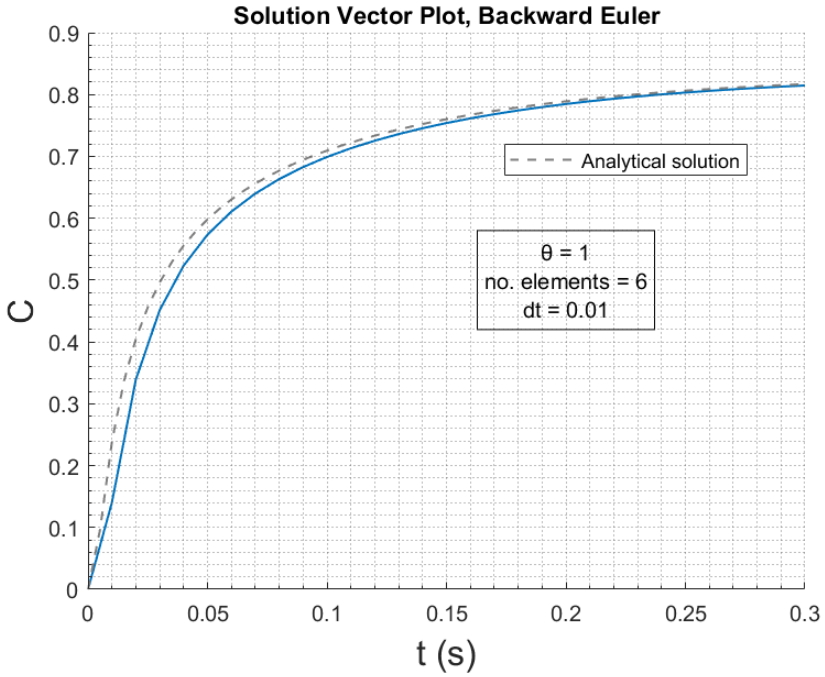


Figure 11: Solution vector plot at x = 0.8 depicting the Backward Euler stepping scheme, $dt = 0.01$.

All three stepping schemes were run using a quadratic mesh of 6 elements and a range of timesteps. In the plots shown above, a timestep dt of 0.01 s was used. Whilst both the Crank Nicolson and Backward Schemes were unconditionally stable, the Forward Euler stepping scheme displayed instability if dt was not sufficiently small. This made the Euler scheme the least reliable of the schemes, and consequently, least suitable for use within this project.

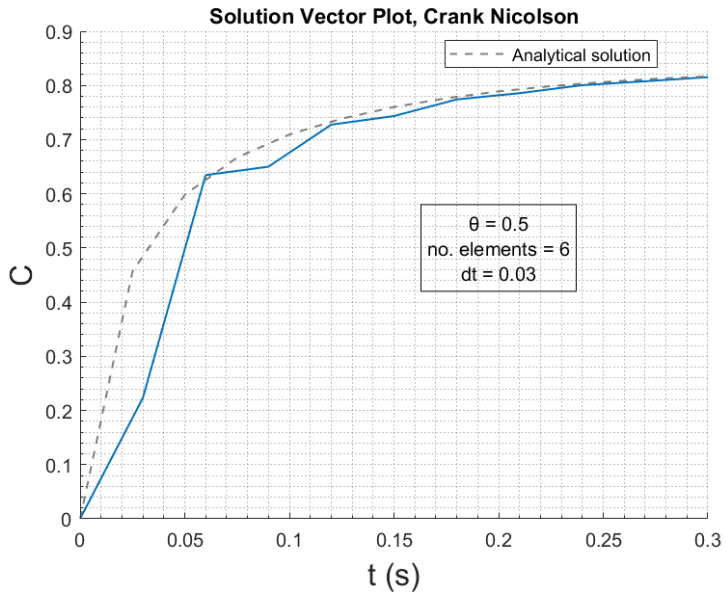


Figure 12: Solution vector plot at x = 0.8 depicting the Crank Nicolson stepping scheme, $dt = 0.03$.

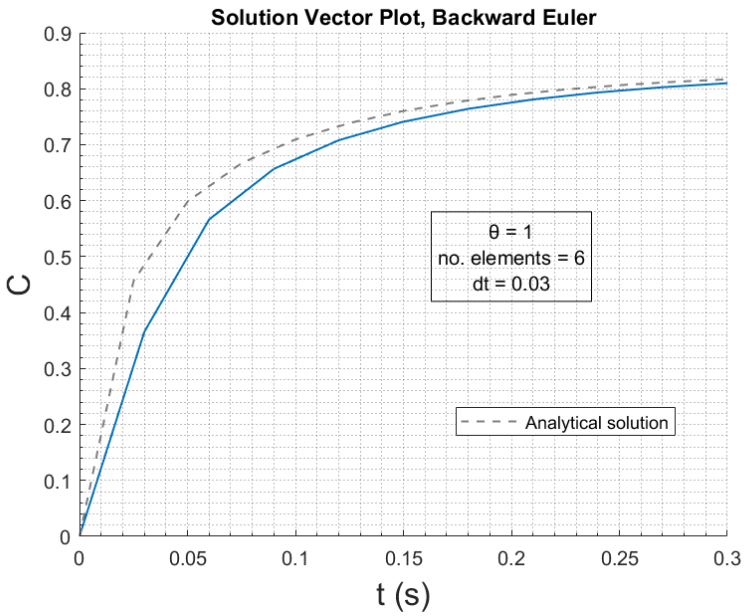


Figure 13: Solution vector plot at x = 0.8 depicting the Backward Euler stepping scheme, $dt = 0.03$.

Using a larger timestep dt = 0.03, both the Crank-Nicolson and Backward Euler schemes converged towards the analytical solution within 0.2 s of simulation time, but displayed noticeable discrepancies at the start. The Crank Nicolson Scheme displayed oscillatory behaviour before settling, while the Backward Euler Scheme displayed a smoother convergence to the analytical solution.

Between these two schemes, Crank-Nicolson is associated with a higher accuracy, with an error amplification rate in the order of dt^3 . In contrast, both the Backward and Forward Euler schemes display amplification rates in the order of dt^2(NCOE, n.d.). However, this comes at increased computational cost, and should be considered when selecting a scheme.

Gaussian Quadrature.

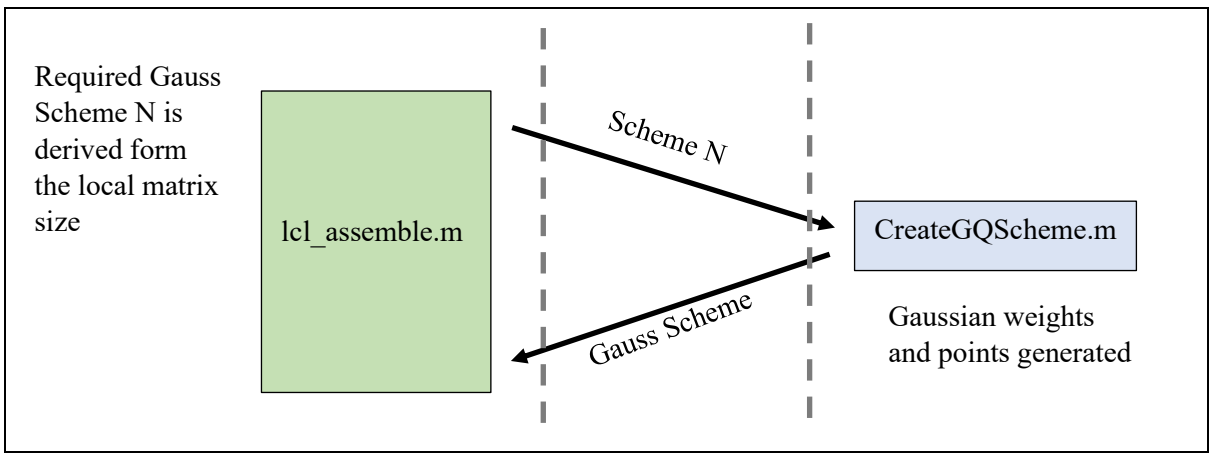


Figure 14: Schematic diagram explaining CreateGQScheme.m function.

Within the code architecture, Gaussian Quadrature was incorporated to evaluate general integrals. Function CreateGQScheme.m (Appendix.2) creates a Gaussian Quadrature Scheme that can then be passed through to other functions for use, such as lcl_assemble.m (Appendix A.9).

Within this assignment, Gauss Schemes of up to N = 4 were used.

Quadratic basis functions:

Within this assignment, code was generalised so both linear and quadratic meshes could be used. Function QuadMeshGen.m (Appendix A.14) generates a quadratic one-dimensional mesh in analogous manner to OneDimLinearMeshGen.m (Appendix A.27), giving the user an option of which mesh structure to use.

Time - varying and spatially - varying material fields

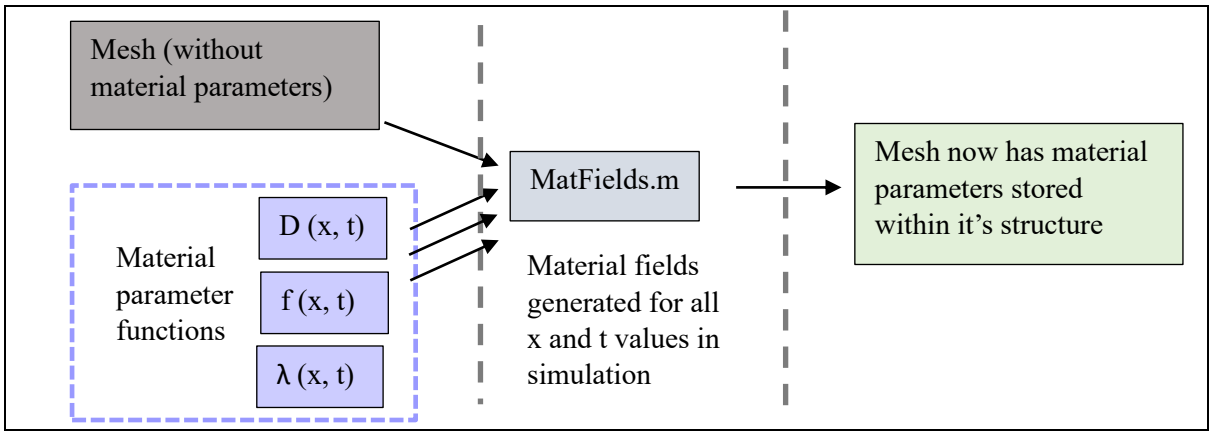


Figure 15: Flow Diagram describing how function MatFields.m produces mesh material parameters

To improve code flexibility, material fields were generated and stored in the mesh by a separate function file, MatFields.m (Appendix A.10). By altering function definitions, material parameters could be set to vary spatially and in time, as well as quadratically or linearly.

User-interface and visual simulation

Table 11: Files related to user-interactive features of model

File name	Description	Appendix
set_all_BC.m	Uses UI to prompt user for Boundary Conditions	A.15
Summative.m	Has an optional transient plot while simulation is running	A.18

Throughout code development, user interaction was built into the code architecture, controlled via inputs in the main summative.m (Appendix A.18) file. These features can be disabled for data-intensive studies to enhance efficiency, or left on during one-off simulations to observe transient behaviour.

L2 Norm analysis

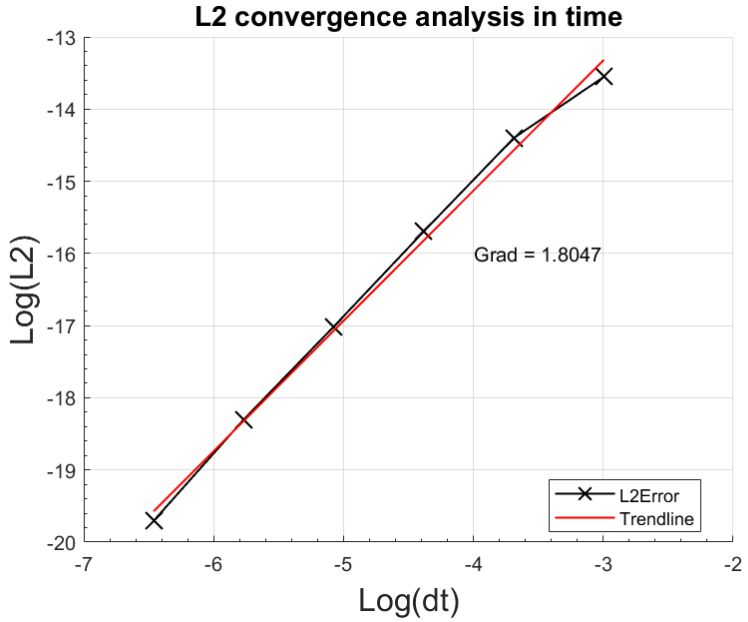


Figure 16: Log-log plot of L2norm convergence study in dt.

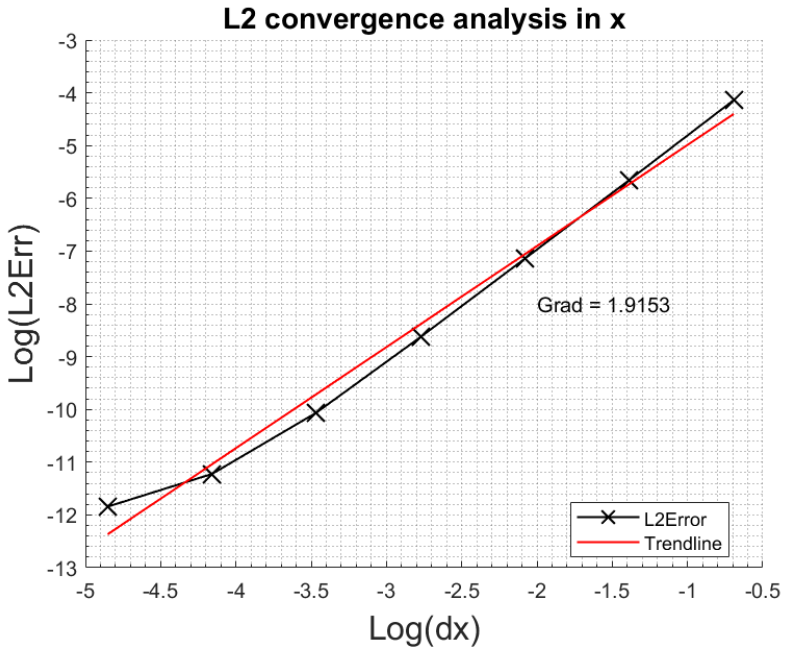


Figure 17: Log-log plot of L2norm convergence study in dx.

Conducting an L2norm convergence study, functions L2.m (Appendix A.8) and GenerateL2Plot.m (Appendix A.7) were used to generate log-log plots of L2 error against dx and dt (Figures 16 and 17 respectively). With a linear mesh and Backward-Euler time-stepping scheme, the rate of convergence was expected to produce a gradient of 2 in both plots.

Experimentally, the observed gradients were 1.8 and 1.92 in the time and spatial convergence studies respectively. With both close to the expected value of 2, these studies supported the validity of the numerical methods employed in this model.

In the plots, deviations from the trendlines occurred at larger dt and smaller dx values. This divergence arose as the parameters become comparable in magnitude, causing the error contributions from the other to become more significant. Finer meshes and smaller timesteps could have been used to produce a more rigorous study, but at the cost of significantly increasing computational time (view Tables 12 and 13). Given that this study served as a qualitative means of validating code, this was deemed unnecessary.

Table 12: Results of L2 convergence study in dx.

dx	L2 error	Computational time (s)
0.5	0.015933	2.265589
0.25	0.003512	4.163817
0.125	0.000791	7.533075
0.0625	0.000179	15.11509
0.03125	4.27E-05	30.08598
0.015625	1.34E-05	60.35237
0.007813	7.19E-06	121.3165

Table 13: Results of L2 convergence study in dt.

dt	L2 error	Computational time (s)
0.05	1.30E-06	0.371637
0.025	5.57E-07	0.460736
0.0125	1.53E-07	1.002952
0.00625	4.08E-08	1.632377
0.003125	1.13E-08	3.307224
0.001563	2.77E-09	6.580412

Current limitations of Code

Within the code used for part 1 of this assignment, key limitations were identified:

1. Currently, the code has not been adapted to account for time-varying boundary conditions.

2. When modelling a transient system, the code is currently very ‘brute-forced’ in extracting material parameters – There is potential to account for this by storing material fields under ‘classes’, such as time varying or spatially varying within the MatFields.m function. This would allow for more computational efficiency when modelling transient behaviour.

3. The current code is only equipped to handle one-dimensional meshes, which may not sufficient to represent more complex systems.

PART 2: Tissue modelling

Code Architecture and Key Features

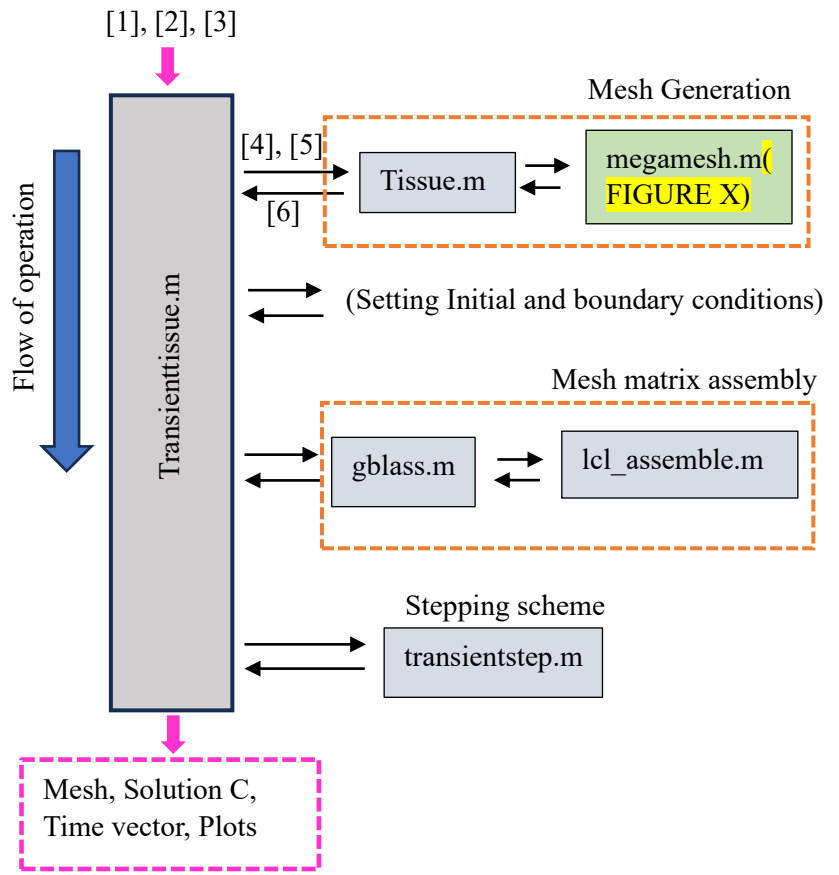
Parameters	Flow Chart
[1] theta scheme [2] surface temperature [3] plot: whether plots should be generated as simulation is run [4] mesh dimensions [5] number of elements in each mesh [6] Fully assembled tissue mesh with material parameters	

Figure 18: Schematic flow diagram showing code architecture for part 2 of assignment.

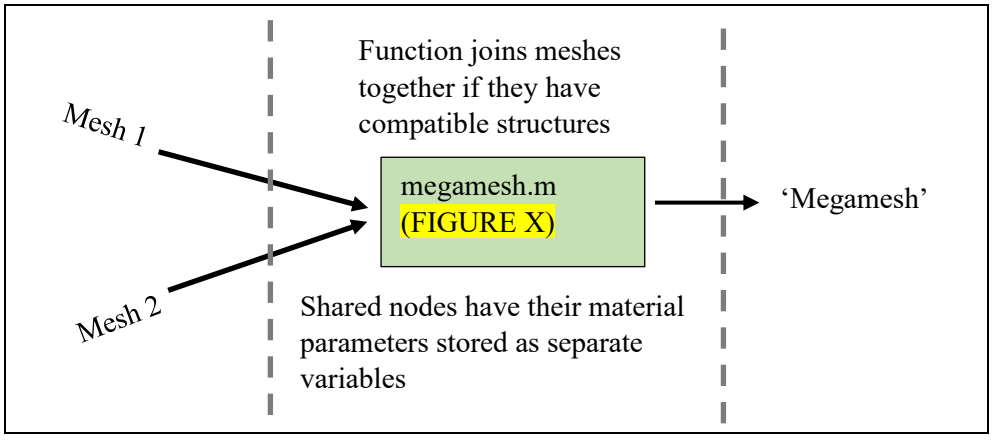


Figure 19: Schematic describing function megamesh.m (APPENDIX B.4)

At its core, the code architecture used for tissue modelling was largely similar to that outlined in Part 1. One notable addition is the function megamesh.m, which merges two meshes into a larger one. By generating separate meshes for the epidermis, dermis, and sub-cutaneous layers, a single 'tissue' mesh could be generated. This allowed for the ability to alter and optimise mesh resolution for each tissue layer independently.

List of functions

Table 14: List of MATLAB files relevant to Part 2 of assignment, arranged in alphabetical order

Function	Description	Appendix reference
insulatedtissuebisection.m	Uses bisection method to find maximum exposure temperature with no burn of insulated tissue	B.1
insulationmsh.m	Generates mesh for insulation	B.2
insulationparameters.m	Stores material parameters of select insulation materials	B.3
megamesh.m	Takes two meshes and assembles into a bigger one	B.4
plot_C_at_TTissue.m	For graph plotting	B.5
plot_C_at_XTissue.m	For graph plotting	B.6
set_BC_Tissue.m	UI interface to set boundary conditions	B.7
tissue.m	Assembles the tissue mesh using megamesh.m	B.8
tissuebisection.m	Uses bisection method to find maximum exposure temperature with no burn of uninsulated tissue	B.9
tissueburn.m	Uses binary search to calculate the total time and exposure time before second-and third-degree burns occur	B.10
tissueparameters.m	Stores material parameters of the different layers of tissue	B.11
transienttissue.m	Main interface between user and code when simulation for uninsulated tissue is being run	B.12
transienttissueinsu.m	Main interface between user and code when simulation for insulated tissue is being run	B.13

P1Q1. General Performance and Initialisation of tissue simulation

Mesh size selection

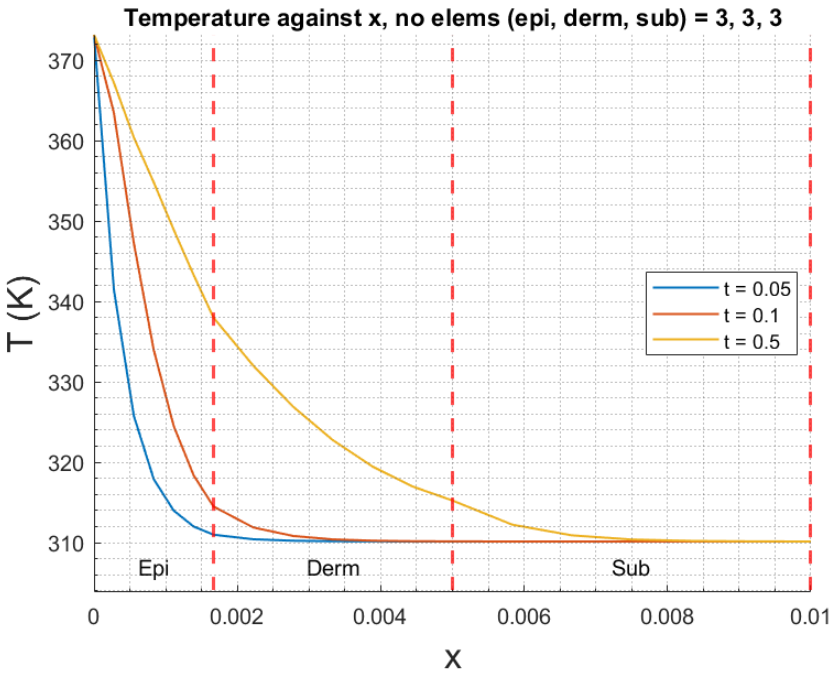


Figure 20: Temperature-x plot of tissue at select times, with three elements in each layer

In the initial set up of the transient tissue simulation, various element number combinations were tested for each tissue layer. A timestep dt of 0.01s was used, as found to be optimal in section P1Q1b.

Figure 22 shows the solution plots corresponding to a mesh with three elements in each tissue layer. Using a quadratic mesh, this corresponds to a full mesh of nine elements with 19 nodes. Whilst the simulation is stable, the solution plot is noticeably kinked, particularly in the first few timesteps.

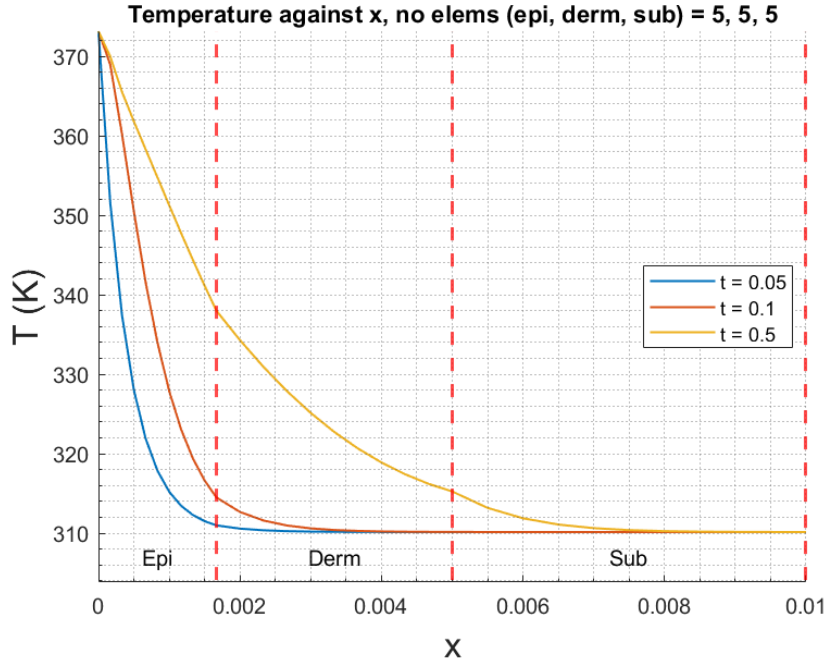


Figure 21: Temperature-x plot of tissue at select times, with five elements in each layer

Increasing the number to five elements in each 5 mitigates this considerably, shown in Figure23. Ten, fifteen, and twenty elements were also considered, but given the increased computational time (Table 15), five elements was deemed to be sufficient.

Table 15: Computational times for different tissue mesh configurations

No. elements in each layer	Computational time for a 50 s simulation with dt = 0.01 s (s)
3	0.5700
5	0.7960
10	1.3775
15	2.0827
20	2.6774

Timestep selection

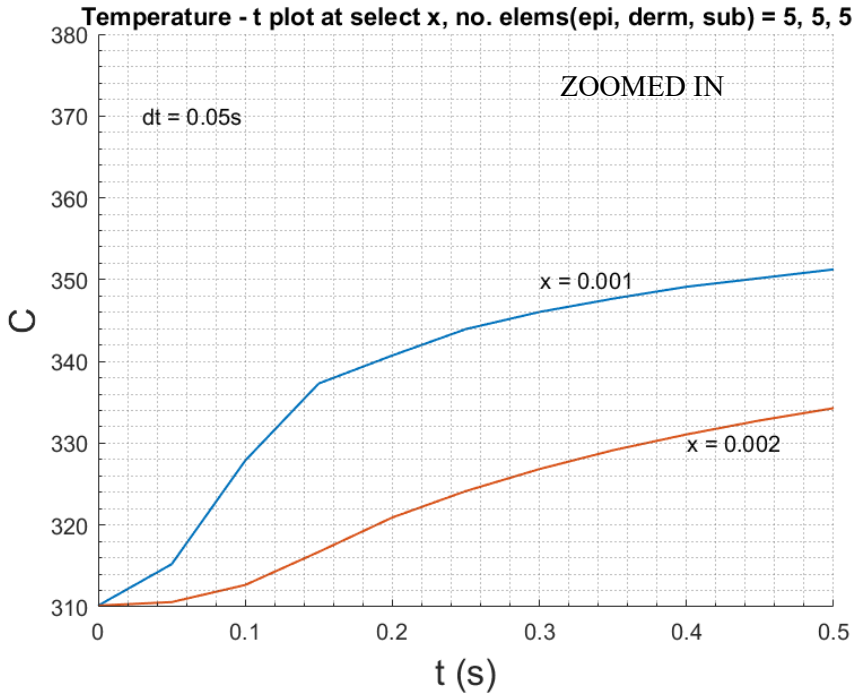


Figure 22: Plot of temperature over time at x = 0.001 using a 5-5-5 element tissue mesh configuration

Shown in Figure 24 is the simulation response at x = 0.001 and x = 0.002 using five elements in each tissue layer and timestep dt of 0.05s. Using this timestep, the solution is initially kinked, indicating that the model may not be accurately simulating the system's immediate behaviour upon applying boundary conditions.

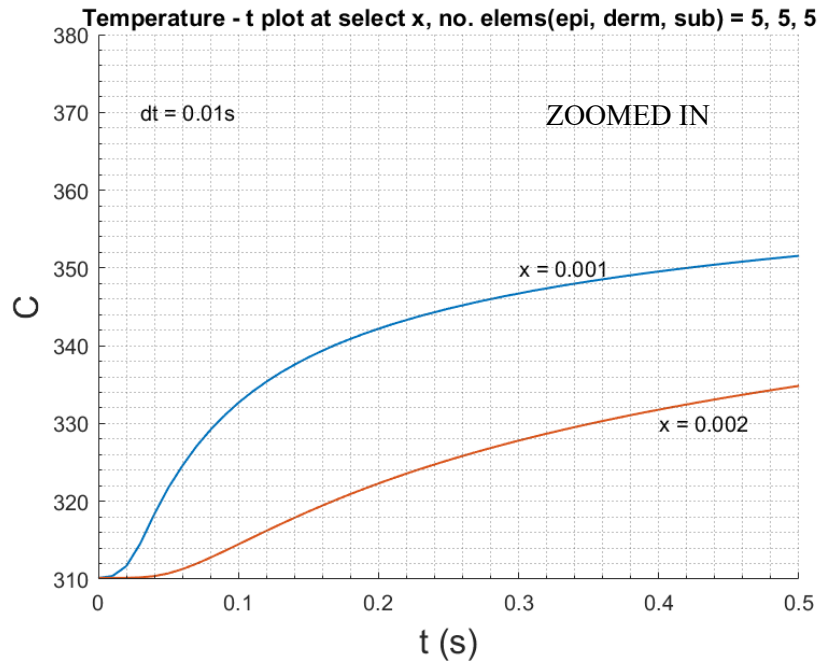


Figure 23: Plot of temperature over time at x = 0.001 and x = 0.005 using a 5-5-5 element tissue mesh configuration

Repeating this with a timestep dt of 0.01s in Figure 25, the plot displays no signs of oscillatory behavior or kinking, validating that a 0.01 s timestep is adequate.

General results and Effect of blood flow

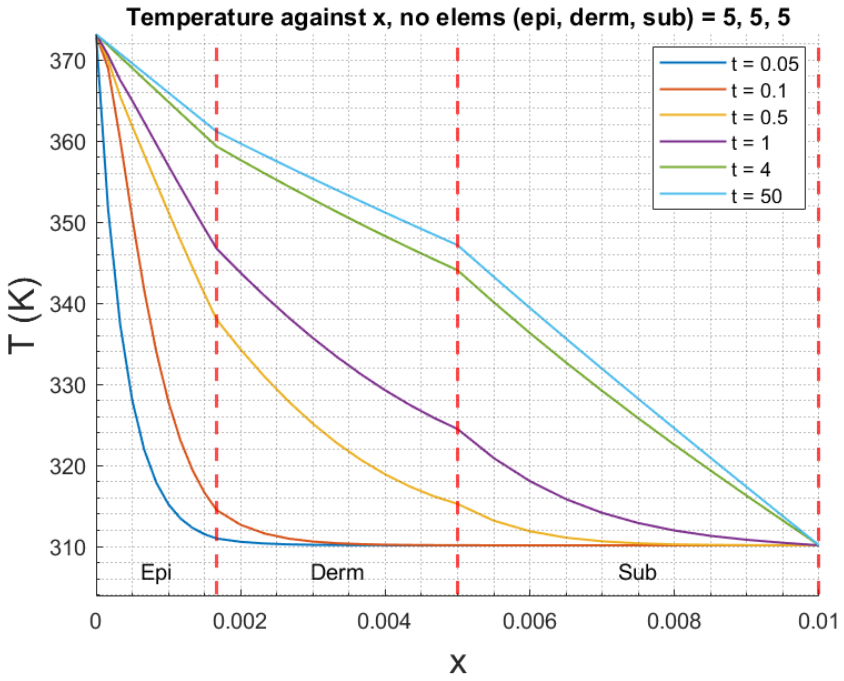


Figure 24: Temperature plot at select times, without the effect of blood flow

Running the model, the system undergoes rapid changes at the beginning of simulation. However, by the time t = 4 s is reached, the system has nearly reached steady state. This can be seen in Figure 26, where the plots for t = 4 s and t = 50 s are extremely similar.

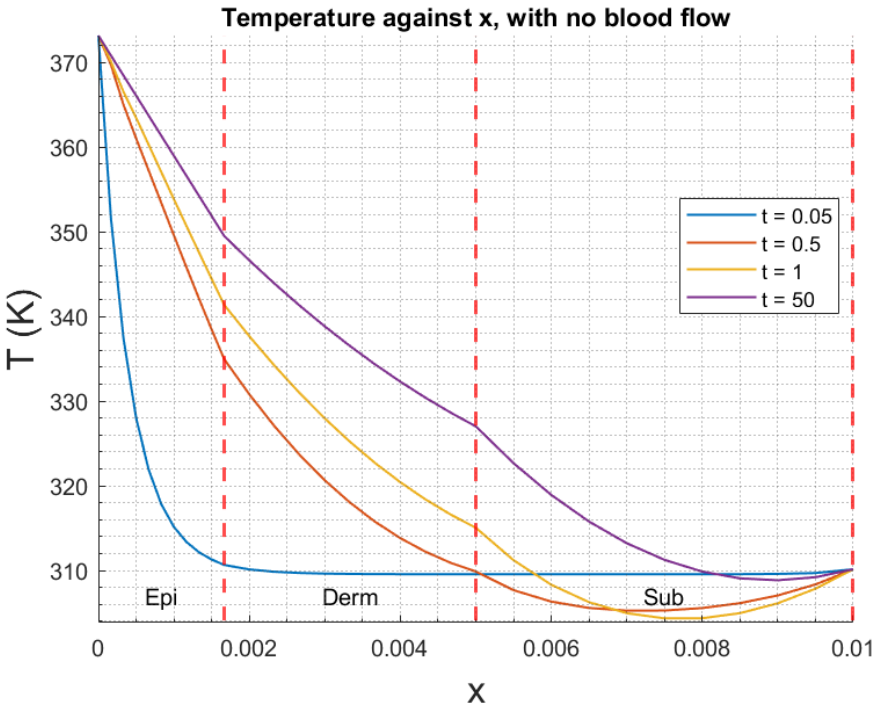


Figure 25: Temperature plot at select times, without the effect of blood flow

Modelling the effect of blood flow in this system is significant. In relation to the transient diffusion-reaction-source equation, omitting blood flow removes the source term, and consequently, alters system behaviour significantly.

This effect is captured in Figure 27, where temperatures initially drop, (most observable in plots corresponding to t = 0.05 s and t = 0.5 s), before rising again. By the end of the simulation (t = 50 s), maximum temperatures in the Dermis and subcutaneous layers are approximately 350 K and 327 K respectively, 3.0 and 5.7% lower compared to 361K and 347 K when blood flow is included.

P2Q2.Burns

Total Damage

Equation 2: Equation for calculating burn damage, where tburn is the time at which temperature first exceeds 317.15 K (Cookson, 2024)

$$\Gamma = \int_{t_{burn}}^{t=50} 2 \times 10^{98} \exp\left(-\frac{12017}{T - 273}\right) dt$$

Using the 'trapz' function and the following Equation 2, total burn damage was calculated at x = 0.005 and x = 0.00166667.

Table 16: Total burn damage after 50 s

	X = 0.005	**x = 0.00166667**
Γ after 50 s	1.61 x 10^{28}	3.79 x 10^{39}

Table 16 notes the calculated damage. As **Γ** > 1 in both cases, the simulation indicated that both second-and third-degree burns occurred.

Minimum Exposure Times

Approach

With the knowledge that both a second- and third-degree burns occurred, a binary search method was used to locate the time at which the burns were experienced (t_{burn}).

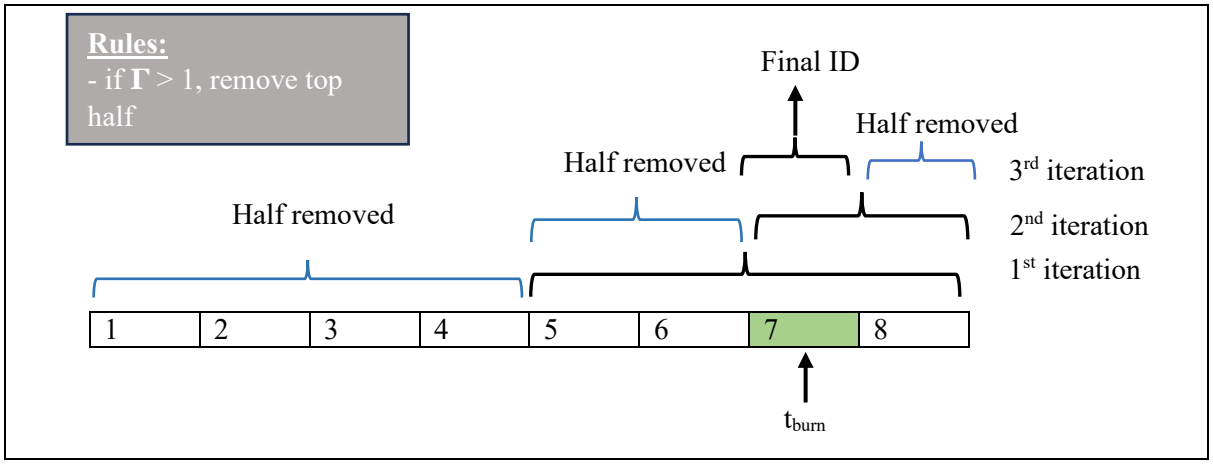


Figure 26: Schematic showing binary search algorithm, assuming t_{burn} is in the seventh cell in array

The binary search algorithm, also known as the half-interval search or logarithmic search method, is an algorithm that locates an item in a sorted array (Weisstein, n.d.-a).

Specifying a target, in this case $\boldsymbol{\Gamma} = 1$, the algorithm calculates the value of the integral $\boldsymbol{\Gamma}$ at the midpoint of the time array. If $\boldsymbol{\Gamma} > 1$, the upper half of the time array is omitted, and the remaining array is halved again. Conversely, if $\boldsymbol{\Gamma} < 1$, the array corresponding to the upper half is searched instead. This process is repeated until the remaining array is one unit long - in this context, this is first timestep in the array where $\boldsymbol{\Gamma} > 1$.

Results

Function tissueburn.m (Appendix B.10) encapsulates the bisection algorithm used to find the minimum exposure times for second and third degree burns.

Table 17: Simulation parameters used in exposure time analysis

No. elements in subcutaneous	No. elements in epidermis layer	No. elements in dermis layer	Timestep dt (s)
20	20	20	0.01

Table 18: Results of exposure time analysis

	Second Degree Burn (Γ = 1 at x = 0.005)	Third Degree Burn (Γ = 1 at x = 0.0016667)
Total time (s)	0.22	1.16
Exposure time (Total time – time taken to reach 317.15 K) (s)	0.12	0.58
Integral value Γ	1.0470	1.2111
Integral value at previous timestep Γ	0.0724	0.6841
Number of loops to find	14	15

Running the algorithm using using the parameters shown in Table 18, minimum exposure times were calculated. For a second-degree burn, t_{exp} was found to be 0.12 s, corresponding to 0.22 s into the simulation, and for a third-degree burn, t_{exp} was found to be 0.58 s, 1.16 s into the simulation.

P2Q3.Minimum temperature reduction

Approach: Bisection Method

To calculate minimum temperature reductions to ensure no second and third degree burns within 50 seconds, the bisection method was used between the temperatures of 310.15 K and 373.15 K.

Pseudocode for the bisection method

1. T1 is initialized to be 310.15 K
2. T2 is initialized to be 373.15 K
3. T3 is initialized as the midpoint of T1 and T2
4. Specify acceptable temperature accuracy Ttol, e.g. $\pm$1 K

5. Loop: While T3 -T2 > (Ttol)
    a. **Γ** is calculated at 50s for T1 applied to the skin, this is **Damage(1)**
    b. **Γ** is calculated at 50s for T2 applied to the skin, this is **Damage(2)**
    c. **Γ** is calculated at 50s for T3 applied to the skin, this is **Damage(3)**
    d. If T3 -T2 > (Ttol)
        i. Depending on whether **Damage(3) is more or less than 1,** Tnext is either the midpoint of T1 and T2, or T2 or T3
        ii. Values are reassigned: T1 = T2, T2 = T3, T3 = Tnext

6. Loop finishes when T2 and T3 are within an tolerance Ttol

Figure 27: Pseudocode for tissuebisection.m (Appendix B.9).

The bisection method is analogous to the binary search algorithm, except instead of discontinuous, sorted arrays, it deals with continuous functions (Weisstein, n.d.-b). As a result, rather than iterating until a single value remains in an array, the algorithm iterates until a certain threshold is met. In this assignment, this threshold was defined as *"when two consecutive test exposure temperatures fall within a certain tolerance"*. The function being bisected was skin exposure temperature.

Notably, the bisection method is only valid under the condition that there is exactly one solution within the range of values being tested. As such, for more complex functions with multiple roots, the method may not be sufficient.

Table 19: Simulation parameters used in exposure time analysis

No. elems in subcutaneaous	No. elems in epidermis layer	No. elems in dermis layer	Timestep dt (s)
5	5	5	0.05

For this study, a temperature tolerance $T_{tol} \pm$ 0.5 K was used. Simulation input parameters remained unchanged to that used previously in calculating burn damage, except for an increase in timestep to 0.05 s to reduce computational time.

Results

Table 20: Minimum temperature reductions to prevent tissue burn calculated using bisection method

	Second Degree Burn (Γ = 1 at x = 0.005)	**Third Degree Burn (Γ = 1 at x = 0.0016667)**
Temperature Upper Bound (K)	329.69	337.19
Temperature Lower Bound (K)	329.29	336.72
Max temperature for no burn over 50 s (K)	329.29	336.72
Temperature reduction from 373.15 K (K)	43.93	36.43
Number of iterations to calculate (no. loops)	7	7
Total Computing time (s)	10.42	

Using the bisection method, a minimum temperature reduction of 44 K was needed to prevent a second-degree burn, and a 37 K reduction to prevent a third-degree burn. These correspond to exposure temperatures of 330 K and 337 K respectively.

P2Q4.Insulation

To model insulation, the code architecture was adapted to provide include insulation as an additional layer in the tissue mesh using the function insulationmsh.m (Appendix B.2).

Material parameters for Kevlar, Wool, and air were stored within the function insulationparameters.m (Appendix B.3) for use during simulation. Kevlar was chosen for its wide use in heat-resistant clothing (DupontUK, n.d.), while wool was selected as a common material in everyday clothing. Material parameters were obtained using Ansys Granta EduPack (Ansys®, 2022).

Table 21: Insulation material parameter values used in study

Material	**Kevlar**		**Wool**		**Air** [2]	
Material thickness (relative to skin)	0.005		0.005		0.02	
Density (kg/m3)	1450		1300		1.293	
Cp (J/(kg.K))	1400		1360		1005	
k (W/(m.K))	0.25	25	0.25	25	0.025	2.5
Nom.	Scaled					

For this study, simulations were run both with and without a layer of air included between the skin and insulative material in the mesh. A mesh thickness $x_{insu} = 0.005$ was used for the insulative layer, and $x_{air} = 0.02$ was used for air[3]. All layers were quadratic with five elements each.

[2] At standard room temperature and pressure

[3] These values were derived by scaling the dimensions of the epidermis layer, using the assumption that the epidermis layer is 1-4 mm thick LECHLER, T. 2019. Growth and Differentiation of the Epidermis. *In:* KANG, S., AMAGAI, M., BRUCKNER, A. L., ENK, A. H., MARGOLIS, D. J., MCMICHAEL, A. J. & ORRINGER, J. S. (eds.) *Fitzpatrick's Dermatology, 9e.* New York, NY: McGraw-Hill Education, ibid., insulative material layer 1-5 mm thick, and air 5-10mm thick.

Results: Direct-contact assumption

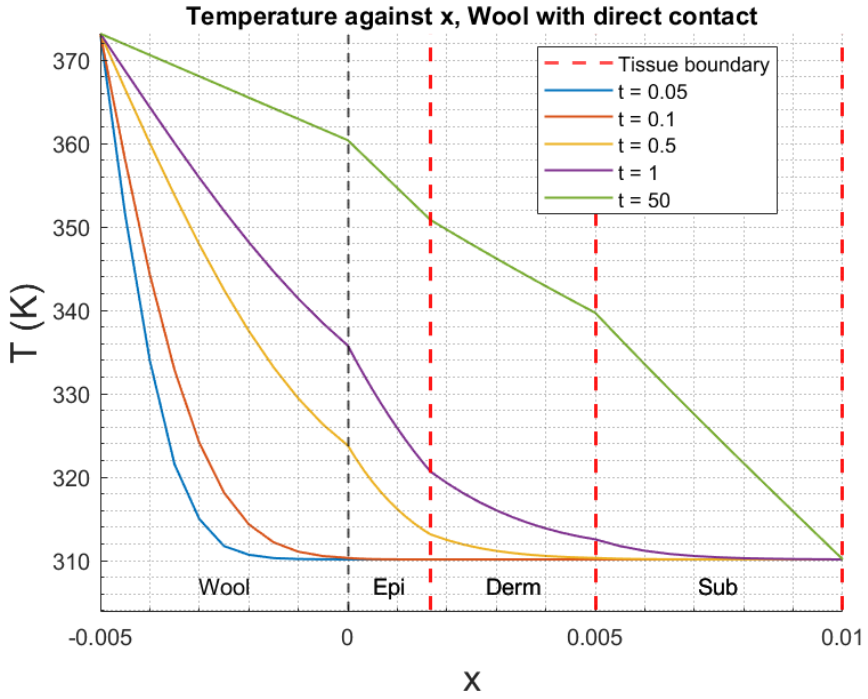


Figure 28: Temperature - x plot of tissue insulated with Wool, assuming direct contact with skin.

Table 22: Effect of insulation on preventing burns

	No insulation		Kevlar		Wool	
Type of burn	2nd	3rd	2nd	3rd	2nd	3rd
Total time until burn (s)	0.22	1.16	1.69	3.42	1.52	3.17
Exp. Time (s)	0.12	0.58	0.86	1.69	0.75	1.54
Max T for no burn over 50 s (K)	329	336	334	343	333	342
T reduction from 373.15 K (K)	43	37	39	30	40	31

Overall, both Kevlar and Wool helped to protected tissue to a similar level. The total time taken for second degree burns to occur increased by over 500% when either material was used for insulation, at +1.47 s and +1.30 s for Kevlar and wool respectively. For third degree burns, time until burn increased by over 150% (+2.26 s and +2.01 s, Kevlar and wool respectively). This is significant compared to non-insulated burn time of 0.22 s, and supports the notion that insulative clothing protects from tissue burn.

However, implementing insulation had a less substantial impact over a full 50-second simulation period. For both materials, maximum exposure temperatures without burn increased by 4-5 K (+0.91-1.22%) for a second-degree burn, and 6-7 K (+1.82 -2.13%) for a third-degree burn. Compared to the temperatures observed without insulation, this increase is minimal, suggesting that whilst insulation helps mitigate heat damage during the transient phase of the simulation, it is less effective once steady-state conditions are reached.

Discussion: Effect of including a layer of air

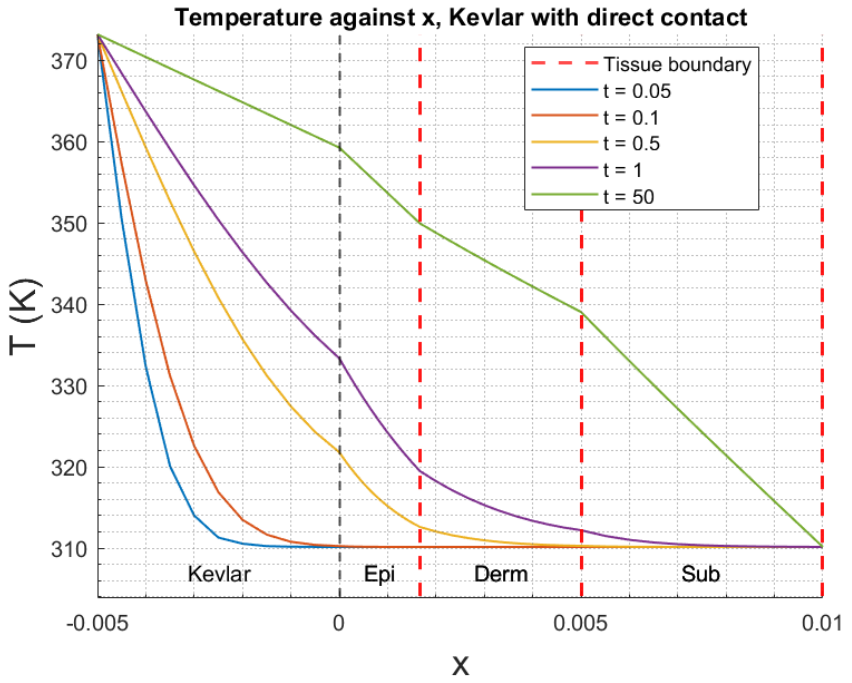


Figure 29: Temperature - x plot of tissue insulated with Kevlar, assuming direct contact with skin.

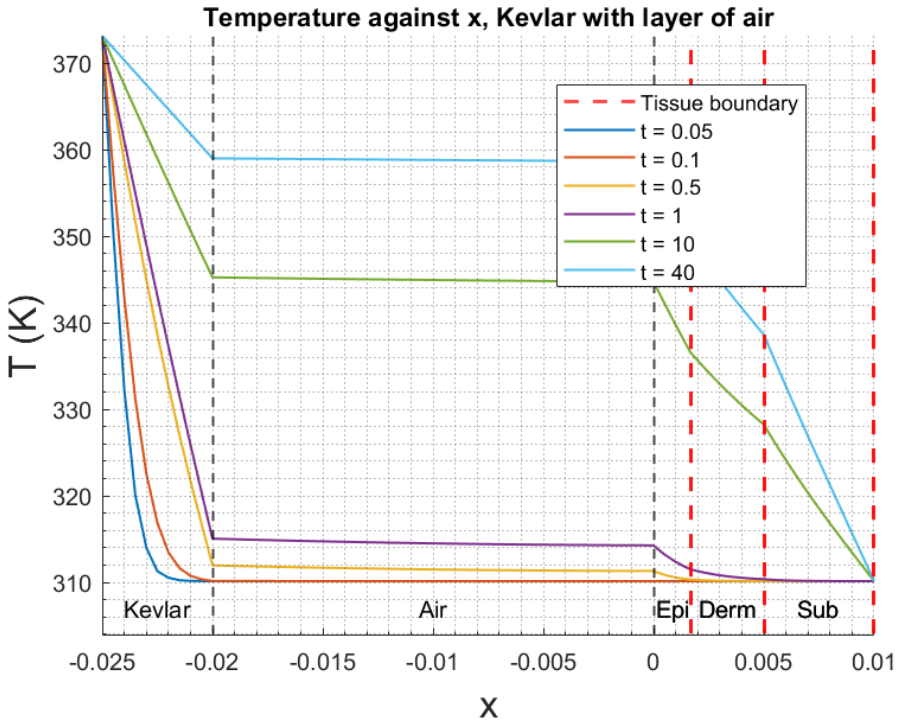


Figure 30: Temperature-x plot of tissue with Kevlar insulation, assuming a layer of air in between

Table 23: Tabulated results when insulative layer is simulated, both with and without a layer of air

	No insulation		**Kevlar**				**Wool**			
	-		**Direct contact**		**With air**		**Direct contact**		**With air**	
Type of burn	2nd	3rd	2nd	3rd	2nd	3rd	2nd	3rd	2nd	3rd
Total time until burn (s)	0.22	1.16	1.69	3.42	5.54	8.92	1.52	3.17	5.03	8.14
Exp. Time (s)	0.12	0.58	0.86	1.69	2.96	4.77	0.75	1.54	2.66	4.28
Max T for no burn over 50 s (K)	329	336	334	343	334	344	333	342	334	343
T reduction from 373.15 K (K)	43	37	39	30	39	29	40	31	39	30

When an additional layer of air was introduced between the insulation and tissue in the mesh, protection against burn was improved further. Compared to the direct-contact assumption, total time until burn increased further by 157-161% for a second degree burn and 228-231% for third-degree burn. This is significant, and substantially indicates that when present in the real case (e.g. a gap of air between protective clothing and skin), air must be accounted for in a simulation model.

However, similar to that observed previously, introducing air had little effect on maximum exposure temperatures over a full 50 seconds, with increases of 1 K (+0.3%) observed for both wool and Kevlar. This reinforced the earlier findings that whilst effective in protecting tissue in the transient phase, this of insulation may be insufficient for prolonged heat exposure.

Discussion: Limitations of the tissue model

In modelling insulation as a form of tissue protection, the current model poses various limitations.

First, it does not consider continuously varying material parameters across tissue. Given the approach to mesh assembly, material parameters are discontinuous at the interfaces where smaller each tissue layer intersects, which may not be reflective of reality.

Secondly, material parameters are assumed to be constant with respect to time, and consequently may not capture certain processes. For example, blood flow may increase due to vasodilation, altering the diffusive effects of the system (Ramanlal and Gupta, 2024). Similiarly, material parameters may change with temperature, e.g. the specific heat capacities of burnt and healthy skin may differ substantially. These are not considered in this model, and pose limitations to real life application.

Finally, in utilising a one-dimensional mesh, transverse effects are not considered. As such, whilst this model may be suitable for simulating uniform heat distribution across a skin's surface (i.e. assuming that the net-transverse heat-transfer is zero), it may be inadequate to model point sources of heat, such as a localized and/or intense burn.

References

ANSYS® 2022. Granta EduPack: Softare for Materials Education. Ansys Inc.

COOKSON, A. 2024. Assignment: Transient MATLAB-Based FEM Modelling. University of Bath.

DUPONTUK. n.d. *Firefighter gear & turnout gear* [Online]. Available: https://www.dupont.co.uk/personal-protection/firefighter-turnout-gear.html#:~:text=Nomex%C2%AE%20and%20Kevlar%C2%AE,comfort%20from%20the%20inside%20out. [Accessed 24th December 2024].

LECHLER, T. 2019. Growth and Differentiation of the Epidermis. *In:* KANG, S., AMAGAI, M., BRUCKNER, A. L., ENK, A. H., MARGOLIS, D. J., MCMICHAEL, A. J. & ORRINGER, J. S. (eds.) *Fitzpatrick's Dermatology, 9e.* New York, NY: McGraw-Hill Education.

NCOE. n.d. *Analysis of finite difference equations¶* [Online]. Norweigian Centre of Excellence. Available: https://hplgit.github.io/INF5620/doc/pub/sphinx-decay/._main_decay006.html [Accessed 6th December 2024].

OPENAI. 2024. *ChatGPT-4* [Online]. OpenAI. Available: https://openai.com/ [Accessed 2024].

RAMANLAL, R. & GUPTA, V. 2024. Physiology, Vasodilation. *StatPearls.* Treasure Island (FL): StatPearls Publishing

Copyright © 2024, StatPearls Publishing LLC.

WEISSTEIN, E. W. n.d.-a. *Binary Search* [Online]. MathWorld--A Wolfram Web Resource. Available: https://mathworld.wolfram.com/BinarySearch.html [Accessed 10th December 2024].

WEISSTEIN, E. W. n.d.-b. *Bisection* [Online]. MathWorld--A Wolfram Web Resource. Available: https://mathworld.wolfram.com/Bisection.html [Accessed 20249th December 2024 2024].

Appendix

Appendix A

A.1. BCP1Q1.m

```
function [NBC, DBC] = BCP1Q1()
%Function that initialises the Boundary conditions corresponding to the
%coursework assignment Part 1, Question 1, namely:

% x = [0, 1], c(x, 0) = 0, c(0, t) = 0, c(1, t) = 1, t>0

NBC = 'no NBCs';
DBC = [1, 0; 2, 1];

end
```

A.2. CreateGQScheme.m

```
function [ gq ] = CreateGQScheme(N)
%CreateGQScheme Creates GQ Scheme of order N
% Creates and initialises a data structure
gq.npts = N;

if (N > 0) && (N < 5)
%order of quadrature scheme i.e. %number of Gauss points
    gq.gsw = zeros(N,1); %array of Gauss weights
    gq.xipts = zeros(N,1); %array of Gauss points
switch N
    case 1 %GQ for N = 1
        gq.gsw(1) = 2;
        gq.xipts(1) = 0;

    case 2 %GQ for N = 2
        gq.gsw(1) = 1;
        gq.gsw(2) = 1;

        gq.xipts(1) = -((1/3)^0.5);
        gq.xipts(2) = (1/3)^0.5;

    case 3 %GQ for N = 3
        gq.gsw(1) = 5/9;
        gq.gsw(2) = 8/9;
        gq.gsw(3) = 5/9;

        gq.xipts(1) = -((3/5)^0.5);
        gq.xipts(2) = 0;
        gq.xipts(3) = (3/5)^0.5;

    case 4 %GQ for N = 4
        gq.gsw(1) = (18-(30)^0.5)/36;
        gq.gsw(2) = (18+(30)^0.5)/36;
        gq.gsw(3) = (18+(30)^0.5)/36;
        gq.gsw(4) = (18-(30)^0.5)/36;

        gq.xipts(1) = (3/7+2/7*(6/5)^0.5)^0.5;
```

```
        gq.xipts(2) = (3/7-2/7*(6/5)^0.5)^0.5;
        gq.xipts(3) = -(3/7-2/7*(6/5)^0.5)^0.5;
        gq.xipts(4) = -(3/7+2/7*(6/5)^0.5)^0.5;
end

else
    error('N must be between 1 and 3 for this quadrature scheme.');

end
```

A.3. Dirichlet.m

```
function DirGlobals = Dirichlet(global_matrix, global_vector, DBC)
%This is a function that enforces Dirichlet conditions into the global
%matrix and vector.

%DBC = [1, (Dirichlet condition at xmin), 2, (Dirichlet condition at xmax)]
%DirGlobal is of form {GlobalMatrix, Global_vector}



%Alters both the global vector and global matrix


%Enforcing onto global vector where relevant
checkxmin = isnan(DBC(1, 2));%Check if there is Dirichlet enforced at xmin

if checkxmin == 0 %Enforcing Dirichlet at xmin
    global_matrix(1, :) = 0; %Setting top row to zero
    global_matrix(1, 1) = 1; %Setting the top left to one
    global_vector(1) = DBC(1, 2); %Setting value to global vector
end

checkxmax = isnan(DBC(2, 2));%Check if there is Dirichlet enforced at xmax
if checkxmax == 0 %Enforcing Dirichlet at xmax

    global_matrix(end, :) = 0;      %Setting bottom row to zero
    global_matrix(end, end) = 1;    %Setting bottom right to 1
    global_vector(end) = DBC(2, 2); %Setting value to global vector

end

DirGlobals = {global_matrix, global_vector};
```

A.4. EvalBasis.m

```
function [ psi ] = EvalBasis(lnid, xipt, order)
%EvalBasis Evaluates linear OR QUADRATIC Lagrange basis functions
% Given the local node id (lnid) for a linear Lagrange element, and a xi
% coordinate (between [-1,1]), returns the correpsonding value of the
% basis function for that local node, at that xi point

switch order
```

```
    %for linear basis functions
    case 1
        sign = (-1)^(lnid+1);
        psi = 0.5*(1 + ((sign*xipt)));
    %for quadratic basis functions
    case 2
        switch lnid
            case 0
                psi = xipt*(xipt-1)/2;
            case 1
                psi = 1-xipt^2;
            case 2
                psi = xipt*(xipt+1)/2;
        end
end

end
```

A.5. EvalBasisGrad.m

```
function [ dpsidxi ] = EvalBasisGrad(lnid, xipt, order)
%   EvalBasisGrad Returns gradient of basis functions
%   Returns the gradients of the linear OR quadratic  Lagrange basis functions
for a
%   specificed local node id (lnid = 0 or 1 for linear, 0, 1, or 2 for
quadratic) and xipt

% order = 2 is quadratic, 1 is linear

switch order
    case 1
        %Use the node id to generate the sign of the basis gradient - ie.
        %either + or -. when lnid=0, sign is -ve, when lnid=1, sign is +ve.
        sign = (-1)^(lnid+1);
        dpsidxi = 0.5 * sign;

    case 2 %Gradients for quadratic basis functions
        switch lnid
            case 0
                dpsidxi = xipt - 0.5;
            case 1
                dpsidxi = -2*xipt;
            case 2
                dpsidxi = xipt + 0.5;
        end

end


end
```

A.6. gblass.m

```
function gblmats = gblass(msh, gbl, gblvec, tID)

%Function that assembles all the global matrices of a mesh at a given time
%indicated by tID

%Initialising all of the respective matrices
%May not be the most computationally efficient, but robust
gblmats.d = gbl; %DIffusion matrix
gblmats.lin = gbl; %Linear reaction matrix
gblmats.stiff = gbl; %Stiffness Matrix
gblmats.mass = gbl; %Mass matrix
gblmats.f = gblvec; %Source term vector
gblmats.BC = gblvec; %Boundary condition vector


%Assemble local matrices and source terms for all elements
lclmats = cell(1, msh.ne); %Preallocating the size of lclmats
for i = 1 : msh.ne
    lclmats{i} = lcl_assemble(i, msh, tID); %Use function lcl_assemble to
assemble all local matrices
end

%Assembling global diffusion matrix
for i = 1: msh.ne %Loop for every element in msh, add to global
    nstrt = msh.elem(i).n(1);
    nend = msh.elem(i).n(end);
    gblmats.d(nstrt:nend, nstrt:nend) = gblmats.d(nstrt:nend, nstrt:nend) +
lclmats{i}.Diff;
end


%Assembling linear reaction matrix
for i = 1: msh.ne %Loop for every element in msh, add to global
    nstrt = msh.elem(i).n(1);
    nend = msh.elem(i).n(end);
    gblmats.lin(nstrt:nend, nstrt:nend) = gblmats.lin(nstrt:nend, nstrt:nend) +
lclmats{i}.Lin;
end


%Assembling stiffness matrix
for i = 1: msh.ne %Loop for every element in msh, add to global
    nstrt = msh.elem(i).n(1);
    nend = msh.elem(i).n(end);
    gblmats.stiff(nstrt:nend, nstrt:nend) = gblmats.stiff(nstrt:nend,
nstrt:nend) + lclmats{i}.Stiffness;
end

%Assembling mass matrix
for i = 1: msh.ne %Loop for every element in msh, add to global
    nstrt = msh.elem(i).n(1);
    nend = msh.elem(i).n(end);
    gblmats.mass(nstrt:nend, nstrt:nend) = gblmats.mass(nstrt:nend, nstrt:nend)
+ lclmats{i}.Mass;
end

%Assembling global vector
```

```
for i = 1: msh.ne %Loop for every element in msh, add to global
    nstrt = msh.elem(i).n(1);
    nend = msh.elem(i).n(end);
    gblmats.f(nstrt:nend, :) = gblmats.f(nstrt:nend, :) + lclmats{i}.f;
end


end
```

A.7. generateL2Plot.m

```
function Study = generateL2Plot(no_dt, no_dx, analyticsol)
%A function to generate the L2norm plots for teh convergence studies
% no_dt is the number of dt values tested, no_dx for dx values. Each
% step tested is half the size of the previous one


%Generating and saving plot of L2 error against element size
nt = zeros(no_dt, 1); %Initialise vector corresponding to the number of
timesteps
dtvec = zeros(no_dt, 1); %Initialise vector corresponding to timestep size
L2terr = zeros(no_dt, 1); %This is the L2error vector corresponding to L2
convergence in time
ne = 100; %choosing an large number of elements
tID = 2; %This is the tID that error is going to be calculated at

%% Loop corresponding to dt convergence

for i = 1: no_dt %Loop generating L2 error of time domain
    tic %time how long each loop takes
    nt(i) = 20*(2^(i-1));
    dtvec(i) = 1/(nt(i));
    t = linspace(0, 1, nt(i)+1); %Generates an increasingly higher resolution
time vector
    rawdata = summative(1, 1, 1, ne, t, 0, analyticsol); %Generate the data
needed to calculate the L2Error
    C = rawdata{1};
    t = rawdata{2};
    msh = rawdata{3};

    L2terr(i) = L2(C, msh, analyticsol, t, tID); %Saves the L2error
    tprocessingtime(i) = toc; %Time how long it took to execute

end

disp('time done') %So user has an idea of where code is at


%% loop corresponding to dx convergence

%Generating and saving plot of L2 error against element size
ne = zeros(no_dx, 1); %Initialise vector corresponding to the number of elements
in mesh that we are going to test
dxvec = zeros(no_dx, 1); %Initialise vector vontaining the dx value
L2xerr = zeros(no_dx, 1); %This is the L2error vector corresponding to L2
convergence in time
t = linspace(0, 1, 10001); %Create an arbitiarily huge timestep

tID = 15;
```

```
for i = 1: no_dx %Loop generating L2 error of x domain
    tic %time how long each thing takes
    ne(i) = 2*(2^(i-1)); %Starting with a dx val of 0.5, creates an increasingly
high resolution mesh size
    dxvec(i) = 1/(ne(i));
    rawdata = summative(0.5, 1, 1, ne(i), t, 0, analyticsol); %Generate the data
needed to calculate the L2Error
    C = rawdata{1};
    t = rawdata{2};
    msh = rawdata{3};

    L2xerr(i) = L2(C, msh, analyticsol, t, tID);
    xprocessingtime(i) = toc; %Time how long it took to execute
end



%%
%Plotting L2 errors on graph
%L2 analysis in respect to timestep

%Using polyfit to find line of best fit, extract gradient
tcoeffs = polyfit(log(dtvec), log(L2terr), 1);
tgrad = tcoeffs(1);

%For plotting line of best fit
txFit = linspace(min(log(dtvec)), max(log(dtvec)), 20);
tyFit = polyval(tcoeffs , txFit);

figure(1) % for time convergence
hold on
    plot(log(dtvec), log(L2terr), Color='k',LineStyle='-',LineWidth=1,
Marker='x', MarkerSize= 12)
    plot(txFit, tyFit, Color = 'r', LineStyle = '-', LineWidth=1)
    text(-4, -16, append('Grad = ', num2str(tgrad)))
    legend('L2Error', 'Trendline', 'Location','best')
    xlabel('Log(dt)', 'FontSize',16)
    ylabel('Log(L2)', 'FontSize',16)
    grid on
grid minor;
    ax = gca;
    ax.XMinorTick = 'on';
    ax.YMinorTick = 'on';
    ax.MinorGridLineStyle = ':';
    ax.MinorGridColor = [0.3, 0.3, 0.3];
    ax.MinorGridAlpha = 0.7;
    title('L2 convergence analysis in time', 'FontSize',14)

hold off
%%
%L2 analysis in respect to dx

%Using polyfit to find line of best fit, extract gradient
xcoeffs = polyfit(log(dxvec), log(L2xerr), 1);
xgrad = xcoeffs(1);

%For plotting line of best fit
xxFit = linspace(min(log(dxvec)), max(log(dxvec)), 20);
xyFit = polyval(xcoeffs , xxFit);
```

```
%%
figure(2) %for x convergence
hold on
    plot(log(dxvec), log(L2xerr), Color='k',LineStyle='-',LineWidth = 1,
Marker='x', MarkerSize= 12)
    plot(xxFit, xyFit, Color = 'r', LineStyle = '-', LineWidth= 1)
    text(-2, (max(log(L2xerr))+min(log(L2xerr)))/2, append('Grad = ',
num2str(xgrad)))

    legend('L2Error', 'Trendline', 'Location','best')
    xlabel('Log(dx)', 'FontSize',16)
    ylabel('Log(L2Err)', 'FontSize',16)
    grid on
grid minor;
    ax = gca;
    ax.XMinorTick = 'on';
    ax.YMinorTick = 'on';
    ax.MinorGridLineStyle = ':';
    ax.MinorGridColor = [0.3, 0.3, 0.3];
    ax.MinorGridAlpha = 0.7;
    title('L2 convergence analysis in x', 'FontSize',14)

hold off

%Values of study
Study.L2values.t = L2terr;
Study.L2values.x = L2xerr;

Study.dtvec = dtvec;
Study.dxvec = dxvec;

Study.grads.t = tgrad;
Study.grads.x = xgrad;

Study.times.t = tprocessingtime;
Study.times.x = xprocessingtime;


end
```

A.8. L2.m

```
function  L2normdomain = L2(C, msh, analyticsol, t ,tID)
%A script to calculate the total accumulated L2 error at a given point in time
for the
%entire 1D mesh

%analytic is the function handle for the analytical solution, e.g.
%@TransientAnalyticSoln
N = 4; %Using gauss scheme of 4 because why not

gqL2 = CreateGQScheme(N); %Generate gauss scheme

for i = 1: msh.ne %calculate error for each element in msh
    ngn = length(msh.elem(i).n); %number of nodes in the element
    J = msh.elem(i).J; %Extracting the Jacobian
    order = ngn-1; %basis function describing the element

    C_anal = zeros(gqL2.npts, 1); %analytic
```

```
    C_num = zeros(gqL2.npts, 1); %numerical

    for j = 1: gqL2.npts %for each gauss point


        %Initialise
        C_lclnode = zeros(ngn, 1);
        psi = zeros(ngn, 1);
        num_contribution = zeros(ngn, 1);

        for k = 1: ngn %Calculate C using basis funcitons and nodal C values
            C_lclnode(k) = C(msh.elem(i).n(k), tID ); %C nodal value
            psi(k) = EvalBasis(k-1, gqL2.xipts(j), order); %psi value
            num_contribution(k) = psi(k)*C_lclnode(k); %Contribution to
interpolated C
            xnode(k) = msh.elem(i).x(k)*psi(k); %Calculate x by summing node x
and psi values using EvalBasis()
        end
%%
        gqx = sum(xnode(k));%Calculate the x value by converting out of local
scale
        C_anal(j) = analyticsol(gqx, t(tID));%Calculate analytic solution

        C_num(j) = sum(num_contribution); %Inerpolated C
        Err_xipt(j) = C_anal(j)-C_num(j); %Error by definition
        Err2_xipt(j) = Err_xipt(j)^2; %Error squared
        gwt(j) = gqL2.gsw(j); %gauss weight
        Err2_xipt_weighted(j) = gqL2.gsw(j)*Err2_xipt(j)*J; %multiply by gauss
weight and jacbobian
    end
err2el(i) = sum(Err2_xipt_weighted); %Error for the element

end

L2domain = sum(err2el); %sum for the whole domain
L2normdomain = L2domain^0.5;%normalise
```

A.9. lcl_assemble.m

```
function lclMat = lcl_assemble(eID, msh, tID)
%This is a function that creates and stores ALL LOCAL MATRICES OF THE
%the element eID within a mesh msh, at a given tID

%This has been adapted to account for nodes with double parameters(when
%view megamesh.m)
ngn = length(msh.elem(eID).n); %number of nodes in the element
J = msh.elem(eID).J; %Extracting the Jacobian

order = ngn-1; %basis function describing the element
N = ceil((order+1)/2)+1;
lclMat.Mass = zeros(ngn); %Local Mass Matrix M
lclMat.Lin = zeros(ngn); %Local linear reaction matrix
lclMat.Diff = zeros(ngn); %Local Diffusion Matrix
lclMat.Stiffness = zeros(ngn); %Local Stiffness Matrix K
lclMat.f = zeros(ngn, 1); %local source vector

gq = CreateGQScheme(N); %generate gq scheme
```

```
%loop that creates the source term vector
for n = 1:ngn
    contribution = zeros(length(gq.xipts), 1);
    for i = 1: length(gq.xipts)

        %Using GQ for the general integral
        contribution(i) = gq.gsw(i)*EvalBasis((n-1), gq.xipts(i), order);
        f = msh.matpar.ffield(msh.elem(eID).n(i), tID);

        %This if statement corresponds to double material parameter values
        if isnan(f) %At mesh boundaries, extract values from shared value
structure
            %extract correct parameter from mesh
            if i == ngn
                index = find(msh.sharednodeid == msh.elem(eID).n(i));
                f = msh.doublepars.ffield{index}(1, tID);
            elseif i == 1
                index = find(msh.sharednodeid == msh.elem(eID).n(i));
                f = msh.doublepars.ffield{index}(2, tID);
            else
                disp('shit did not execute')
            end
        end
        lclMat.f(n, :) = sum(contribution)*J*f; %multiply by Jacobian
    end
end


%loop that creates the Mass matrix
for n = 1:ngn
    for m = 1:ngn
        contribution = zeros(length(gq.xipts), 1);
        for i = 1: length(gq.xipts)
            %Calling Gauss
            contribution(i) = gq.gsw(i)*EvalBasis((n-1), gq.xipts(i),
order)*EvalBasis((m-1), gq.xipts(i), order);
            lclMat.Mass(n, m) = sum(contribution)*J;
        end
    end
end

%loop that creates the Linear Reaction matrix
for n = 1:ngn
    for m = 1:ngn
        %Calling Gauss
        contribution = zeros(length(gq.xipts), 1);

        for i = 1: length(gq.xipts)
            contribution(i) = gq.gsw(i)*EvalBasis((n-1), gq.xipts(i),
order)*EvalBasis((m-1), gq.xipts(i), order);
            lambda = msh.matpar.linfield(msh.elem(eID).n(i), tID);

            %if statment corresponding to double values at node
            if isnan(lambda) %At mesh boundaries, extract values from shared
value structure
                if i == ngn
                    index = find(msh.sharednodeid == msh.elem(eID).n(i));
                    lambda = msh.doublepars.linfield{index}(1, tID);
                elseif i == 1
```

```
                    index = find(msh.sharednodeid == msh.elem(eID).n(i));
                    lambda = msh.doublepars.linfield{index}(2, tID);
                else
                    disp('problem in lclassemble')
                end
            end
            lclMat.Lin(n, m) = -sum(contribution)*J*lambda;
        end
    end
end


%loop that creates the Diffusion matrix
dsquiggledx = 2/(msh.elem(eID).x(end) - msh.elem(eID).x(1)); %Effectively
recalculating the gradient for calculating the diffusion matrix
for n = 1:ngn
    for m = 1:ngn
        %find GQ
        contribution = zeros(length(gq.xipts), 1);

        for i = 1: length(gq.xipts)
            contribution(i) = gq.gsw(i)*EvalBasisGrad((n-1), gq.xipts(i),
order)*EvalBasisGrad((m-1), gq.xipts(i), order)*dsquiggledx^2; %FINISH THIS
            D = msh.matpar.dfield(msh.elem(eID).n(i), tID);

            %If statement for double parameters at node
            if isnan(D) %At mesh boundaries, extract values from shared value
structure
                if i == ngn
                    index = find(msh.sharednodeid == msh.elem(eID).n(i));
                    D = msh.doublepars.dfield{index}(1, tID);
                elseif i == 1
                    index = find(msh.sharednodeid == msh.elem(eID).n(i));
                    D = msh.doublepars.dfield{index}(2, tID);
                else
                    disp('problem in lcl assemble')
                end
            end
            lclMat.Diff(n, m) = sum(contribution)*J*D;
        end
    end
end



lclMat.Stiffness = lclMat.Diff-lclMat.Lin; %stiffness matrix
```

A.10. MatFields.m

```
function msh = MatFields(msh, t, D_func, Lin_func, f_func)
%A function that creates and stores material parameters into the mesh,
%where the user inputs defines the functions as inputs in the form of, for
%example:
% D_func = @(t, x) exp(t).*x.*0.5; for a D parameter that varies linearly
% in x, and exponentially decays over time


%These are stored in a matrix that is sized((no. nodes in mesh), (no. cells
```

```
%in time vector)

%Loop to create a diffusion field in form of (node position, time)
for i = 1: msh.ngn
    for j = 1: length(t)
        msh.matpar.dfield(i, j) = D_func(t(j), msh.nvec(i));
    end
end

%Loop to create a lambdafield in form of (node position, time)
for i = 1: msh.ngn
    for j = 1: length(t)
        msh.matpar.linfield(i, j) = Lin_func(t(j), msh.nvec(i));
    end
end

%Loop to create a sourcefield in form of (node position, time)
for i = 1: msh.ngn
    for j = 1: length(t)
        msh.matpar.ffield(i, j) = f_func(t(j), msh.nvec(i));
    end
end

end
```

A.11. Neumann.m

```
function NeuGlobals =  Neumann(global_matrix, BC_vector, NBC)
%A functions that enforces Neumann boundaries
%NBC is in the form [1, (NBC at xmin); 2, (NBC at xmax)]. the 1 and 2 are
%IDs

%Enforcing onto global vector where relevant
checkxmin = isnan(NBC(1, 2));%Check if there is NBC enforced at xmin
if checkxmin == 0 %Enforcing Dirichlet at xmin
    BC_vector(1) = -NBC(1, 2); %NOTE THE NEGATIVE SIGN HERE
end

checkxmax = isnan(NBC(2, 2));%Check if there is NBC enforced at xmax
if checkxmax == 0 %Enforcing Dirichlet at xmax
    BC_vector(end) = NBC(2, 2);
end

NeuGlobals = {global_matrix, BC_vector};
```

A.12. plot_C_atT.m

```
function  plot_C_atT(msh, C, t, tpltvals, analyticsoln)
%Function that was used to generate plots of C against x at a certain time,
%comparing it against an analytic solution using results produced with the
%main summative.m file

% Example input: plot_C_atT(msh, C, t, [0.03, 2, 5, 10, 50],
@TransientAnalyticSoln)

% msh = mesh
% t = time vector
```

```
% tpltvals = times at which plot is wanted. Can be multiple, e.g. t = [0.05,
0.1, 4, 10]
% analyticsoln = handle for the analytic solution function. e..
% @PureSourceAnalyticSoln


%A function that plots the solution vector C against x

tol = 1e-12; % Set a small tolerance for floating-point comparison
x = msh.nvec; %Extract xvector

figure(1)
hold on
ylim([0 1]) %Set y axis limits

colours = lines(length(tpltvals)); % Use the 'lines' colormap for distinct
colors
greyLegendAdded = false; % so all analytical solutions only have one label on
the legend

for i = 1: length(tpltvals) %for all time values

    CID = find(abs(t-tpltvals(i)) < tol, 1);
    if numel(size(C)) ~= 1 % Check if C is not a number
        disp('Exact time value does not exist in solution matrix')
        disp('Calculating for next closest t value')
        [~, CID] = min(abs(t - tpltvals(i))); %If does not exist, find the index
of closest value
    else
        disp('Exact time value available')
    end

    %For graph plotting
    y = C(:, CID);

    % %Making x vector for plotting analytic solution
    xsol = linspace(min(msh.nvec), max(msh.nvec), 200);
    csol = analyticsoln(xsol, t(CID));
    plot(x, y, 'LineStyle', "-", 'Color', colours(i, :), 'LineWidth', 1,
'HandleVisibility', 'off');

        if i == 4
            timename = append('t = ', num2str(t(CID)), '\uparrow' );
            text(0.4,0.55,timename);
        else
            timename = append('\leftarrow t = ', num2str(t(CID)));
            text(median(x),median(y),timename);
        end
         % Add the analytical plot with a single legend label
    if ~greyLegendAdded
        plot(xsol, csol, 'Color', [0.5 0.5 0.5], 'LineStyle', '--', 'LineWidth',
1, 'DisplayName', 'Analytical solution');
        greyLegendAdded = true; % Ensure the legend is added only once
    else
        plot(xsol, csol, 'Color', [0.5 0.5 0.5], 'LineStyle', '--', 'LineWidth',
1, 'HandleVisibility', 'off');
    end

end
```

```
% Add the legend after plotting
legend('show', 'Location', 'best');


titlename = append('Decaying Reaction' );

thetascheme = append('θ = ', num2str(0.5));
noems = append('no. elements = ', num2str(msh.ne));
dt = append('dt = ', num2str( (t(end)-t(1)) / (length(t)-1) ) );
parameters = {thetascheme, noems, dt};
% text(x(ceil(end/4)),y(ceil(3*end/4)), parameters);
text(x(ceil(end/4)), 0.6, parameters, ...
    'FontSize', 10, 'Color', 'black', ...
    'BackgroundColor', 'white', 'EdgeColor', 'black', ...
    'HorizontalAlignment', 'center', 'VerticalAlignment', 'middle');

title(titlename)
xlabel('x', 'FontSize', 16);
ylabel('Solution vector C', 'FontSize', 16);
 % Ensure grid settings are applied
    grid minor;
    ax = gca;
    ax.XMinorTick = 'on';
    ax.YMinorTick = 'on';
    ax.MinorGridLineStyle = ':';
    ax.MinorGridColor = [0.3, 0.3, 0.3];
    ax.MinorGridAlpha = 0.7;
    ax.Layer = 'top'; % Bring grid to the top
% %Save in form "Formativeplot_lambdaval_DVal_Fval
graphname = append('P1Plot_t =', num2str(t(CID)), '.png');
saveas(gcf,graphname)

end
```

A.13. plot_C_atX.m

```
function  plot_C_atX(msh, C, t, xvals)

%A function that plots the solution vector C against a set of xvalues
% xvals = vector of xvals for plot, e.g. x = [0.1, 0.8, 1]

tol = 1e-12; % Set a small tolerance for floating-point comparison
xplot = t; %x axis

figure(1)
hold on
ylim([0 1])
colours = lines(length(xvals)); % Use the 'lines' colormap for distinct colors
greyLegendAdded = false; % so all analytical solutions only have one label on
the legend

%for all xvalues input by user
for i = 1: length(xvals)

    CID = find(abs(msh.nvec-xvals(i)) < tol, 1);
    if numel(size(CID)) ~= 1  % Check if C is not a number
        disp('Exact x value does not exist in solution matrix')
        disp('Calculating for next closest t value')
```

```
        [~, CID] = min(abs(msh.nvec - xvals(i))); %If does not exist, find the
index of nearest val
    else
        disp('Exact x value available')
    end

    %For graph plotting
    y = C(CID, :);

    % %Making x vector for plotting exact solution
    tsol = linspace(min(t), max(t), 200);
    csol = TransientAnalyticSoln(msh.nvec(CID), tsol);
    xname = append('x = ', num2str(msh.nvec(CID)));
    %text(median(xplot),(median(y)-0.025),xname);
        % plot(xplot, y, 'LineStyle', "-", 'Marker', "x", 'MarkerSize', 8,
'Color', colours(i, :), 'LineWidth', 1, 'DisplayName', xname);
        % plot(xplot, y, 'LineStyle', "-", 'Color', colours(i, :), 'LineWidth',
1, 'DisplayName', xname);
        plot(xplot, y, 'LineStyle', "-", 'Color', colours(i, :), 'LineWidth', 1,
'HandleVisibility', 'off');
       % text(0.4, 0.3, 'dt = 0.01')

    if ~greyLegendAdded
        plot(tsol, csol, 'Color', [0.5 0.5 0.5], 'LineStyle', '--', 'LineWidth',
1, 'DisplayName', 'Analytical solution');
        greyLegendAdded = true; % Ensure the legend is added only once
    else
        plot(tsol, csol, 'Color', [0.5 0.5 0.5], 'LineStyle', '--', 'LineWidth',
1, 'HandleVisibility', 'off');
    end


end

% Add the legend after plotting
legend('show', 'Location', 'best');
% ylim([0 0.9]);
% xlim([0 0.3])

titlename = append('Solution Vector Plot at selected x');

thetascheme = append('θ = ', num2str(0.5));
noems = append('no. elements = ', num2str(msh.ne));
dt = append('dt = ', num2str( (t(end)-t(1)) / (length(t)-1) ) );
parameters = {thetascheme, noems, dt};
% text(x(ceil(end/4)),y(ceil(3*end/4)), parameters);
text(0.7, 0.7, parameters, ...
    'FontSize', 10, 'Color', 'black', ...
    'BackgroundColor', 'white', 'EdgeColor', 'black', ...
    'HorizontalAlignment', 'center', 'VerticalAlignment', 'middle');
% text(0.05, 0.8, parameters, ...
%     'FontSize', 10, 'Color', 'black', ...
%     'BackgroundColor', 'white', 'EdgeColor', 'black', ...
%     'HorizontalAlignment', 'center', 'VerticalAlignment', 'middle');



title(titlename)
xlabel('t (s)', 'FontSize', 16);
```

```
ylabel('C', 'FontSize', 16);
 % Ensure grid settings are applied
    grid minor;
    ax = gca;
    ax.XMinorTick = 'on';
    ax.YMinorTick = 'on';
    ax.MinorGridLineStyle = ':';
    ax.MinorGridColor = [0.3, 0.3, 0.3];
    ax.MinorGridAlpha = 0.7;
    ax.Layer = 'top'; % Bring grid to the top
% %Save in form "Formativeplot_lambdaval_DVal_Fval
graphname = append('P1_X_Plot dt = ', num2str(dt), '.png');
saveas(gcf,graphname)

end
```

A.14. QuadMeshGen.m

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

mesh.ne = Ne; %set number of elements
dx = (xmax - xmin)/Ne; %calculate element size

%% These are for the quadratic basis functions
    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
    dngn = (xmax - xmin)/(mesh.ngn-1); %the amount of space between each
quadratic node

    mesh.nvec = xmin:dngn:xmax; %x values of each QUADRATIC node

%% loop over elements & set the element properties
    for i=1:Ne
%% Quadratic
        %set spatial positions of QUADRATIC nodes
        mesh.elem(i).x(1) = xmin + (i-1)*dx;
        mesh.elem(i).x(2) = xmin + (i-0.5)*dx;
        mesh.elem(i).x(3) = xmin + i*dx ;

        %set global IDs of the QUADRATIC nodes
        mesh.elem(i).n(1) = 2*i - 1;
        mesh.elem(i).n(2) = 2*i;
        mesh.elem(i).n(3) = 2*i + 1;

%% Jacobian is the same for both types of basis functions
        %set element Jacobian based on mapping to standard element
        mesh.elem(i).J = abs(0.5*dx); %note the abs here, it's so that meshes
can be made in negative direction

    end

end
```

A.15. set_all_BC.m

```
function [NBC, DBC] = set_all_BC()
%UI for setting boundary conditions. A bit outdated, but still good

%Applying boundary conditionss
%Neumann BC section of code
answer = questdlg('Enforce Neumann conditions? (D(dc/dx) known at given xmin
and/or xmax)','Yes','No');
% Handle response
switch answer
    case 'Yes'
        calc_Neu = 1;
    case 'No'
        calc_Neu = 2;
end

if calc_Neu ==1
    NBC = setNBC();
else
    NBC = 'no NBCs'; %overwrites previously stored NBC's
end

% %Dirichlet BC section of code
answer = questdlg('Enforce dirichlet? (C known at given xmin and/or
xmax)','Yes','No');
% Handle response
switch answer
    case 'Yes'
        calc_Dir = 1;
    case 'No'
        calc_Dir = 2;
end

%%
if calc_Dir ==1
    DBC = setDBC();
else
    DBC = 'No DBCs';
end

end
```

A.16. setDBC.m

```
function DirCons = setDBC()
% Funciton that used UI to ask if you want to set dirichlet conditions, and
% if so, what value

answer = questdlg('Enforce Dirichlet at xmin?','Yes','No');
% Handle response
switch answer
    case 'Yes'
        prompt = {'Enter concentration value at xmin:'};
        dlgtitle = 'C selection';
        fieldsize = [1 45];
        definput = {'0'};
        valuexmin = str2double(inputdlg(prompt,dlgtitle,fieldsize,definput));
    case 'No'
        valuexmin = NaN;
end
```

```
answer = questdlg('Enforce Dirichlet at xmax?','Yes','No');
% Handle response
switch answer
    case 'Yes'
        prompt = {'Enter concentration value at xmin:'};
        dlgtitle = 'C selection';
        fieldsize = [1 45];
        definput = {'1'};
        valuexmax = str2double(inputdlg(prompt,dlgtitle,fieldsize,definput));
    case 'No'
        valuexmax = NaN;
end

%Producing the output, if no Dirichlet, there will be NaNs in those spaces
xmin = 1;
xmax = 2;

DirCons = [xmin, valuexmin; xmax, valuexmax]; %Housekeeping for personal use
end
```

A.17. setNBC.m

```
function NeuCons = setNBC()
% Funciton that used UI to ask if you want to set Neumann boundary conditions,
and
% if so, what value

answer = questdlg('Enforce Neumann at xmin?','Yes','No');
% Handle response
switch answer
    case 'Yes'
        prompt = {'Enter D(dc/dx) value at xmin:'};
        dlgtitle = 'D(dc/dx) selection';
        fieldsize = [1 45];
        definput = {'2'};
        valuexmin = str2double(inputdlg(prompt,dlgtitle,fieldsize,definput));
    case 'No'
        valuexmin = NaN;
end

answer = questdlg('Enforce Neumann at xmax?','Yes','No');
% Handle response
switch answer
    case 'Yes'
        prompt = {'Enter D(dc/dx) value at xmin:'};
        dlgtitle = 'D(dc/dx) selection';
        fieldsize = [1 45];
        definput = {'0'};
        valuexmax = str2double(inputdlg(prompt,dlgtitle,fieldsize,definput));
    case 'No'
        valuexmax = NaN;
end

%Producing the output, if no BC applied, there will be NaNs in those spaces
xmin = 1;
xmax = 2;
```

```
NeuCons = [xmin, valuexmin; xmax, valuexmax]; %Housekeeping for personal use

end
```

A.18. summative.m

```
%Script encapsulating Summative assignment

%%

function outputs = summative(theta, sol_order, defaultvals, ne, t, plt,
analyticsoln)
% Example input for P1Q1:
% outputs = summative(0.5, 1, 1, 10, [1:0.05:1], 1, @TransientAnalyticSoln)

% theta = dictates the time stepping scheme. 0.5 for Crank, 0 for forward euler,
1 for backward euler2dc
% sol_order = 1 (linear) or 2 (quadratic basis mesh)
% If defaultvals = 1, simulation will run using BC's specified in P1Q1 of
% the assignment
% ne is number of desired elements in mesh.
% t is the time vector
% If plt = 1, a figure with a transient plot will run during simulation
% analyticsoln is a function handle for the analytic solution, e.g.
% @TransientAnalyticSoln

%default is a switch. If default is one, the boundary conditions that are
%specified in coursework assignment Part 1 Q1 will be apply
close all

dt = max(t)/numel(t); %find dt

%Generate a mesh, either linear or quadratic
if sol_order == 1
    msh = OneDimLinearMeshGen(0, 1, ne);
elseif sol_order == 2
    msh = QuadMeshGen(0, 1, ne);
else
    error('Pick a valid mesh order');
end

%Depending on the analytic solution handle, material parameter definitions
%are extracted from materialfuncs.m
[D_func,  Lin_func, f_func] = materialfuncs(analyticsoln);
%Use MatFields to generate and add material parameters D, lambda, etc. to
%the mesh
msh = MatFields(msh, t, D_func, Lin_func, f_func); %This mesh now has the
following stored: dfield, lambdafield, Local Diffusion integral, Local Lambda
Integral

%% Getting into the loop
%Initialise global matrices and vectors using number of nodes in mesh
gbl = zeros(msh.ngn);
gblvec = zeros(msh.ngn, 1);

C = zeros(msh.ngn, length(t)); %Initialise solution for all timesteps

%Initial conditions selected here
if strcmp(func2str(analyticsoln), 'TransientAnalyticSoln')
```

```
    C0 = zeros(msh.ngn, 1); %for P1Q1 of coursework
else
    [C0, ~, ~] = evalBCAnal(analyticsoln, msh); %Boundary conditions of the test
cases mentioned in the report
end

C(:, 1) = C0; %placing solution IC into transient matrix

%%
%Initialising figure for plotting
figure;
hold on;
% Initialize plot handles
h_cur = []; % Handle for current
h_analytic = []; % Handle for the analytic timstep
h_test = [];
%ylim([-1 1])

% Ensure grid settings are applied
grid minor;
ax = gca;
ax.XMinorTick = 'on';
ax.YMinorTick = 'on';
ax.MinorGridLineStyle = ':';
ax.MinorGridColor = [0.3, 0.3, 0.3];
ax.MinorGridAlpha = 0.7;
ax.Layer = 'top'; % Bring grid to the top
%%
counter = 0; %counter for graph plotting
for i = 1: length(t)-1

    if i == 1 %Applying boundary conditions
        if defaultvals == 1
            [NBC, DBC] = BCP1Q1();
        else
            [NBC, DBC] = set_all_BC();
        end

        if ~strcmp(func2str(analyticsoln), 'TransientAnalyticSoln')
            [~, NBC, DBC] = evalBCAnal(analyticsoln, msh); %Finding analytic
solution relevant BC's
        end
    end

   %% Calculating the initial condition
   gblmats = gblass(msh, gbl, gblvec, i); %assembling global arrays

   C1 = transientstep(gblmats, dt, C0, theta, DBC, NBC, i, plt); %finding next
solution vector
   C(:, i+1) = C1; %Adding solution of next timestep into matrix
   C0 = C1; %Reassigning C0

       % Plot C(i+1) every couple loops
       if mod(i, 1) == 0  && counter < 10 && plt == 1
           xsol = linspace(min(msh.nvec), max(msh.nvec), 20);
           csol = analyticsoln(xsol, t(i));

           %Delete the previous plots for neatness
```

```
            if ~isempty(h_cur), delete(h_cur); end %goddamn never realised you
could have a whole if statement on just one line
            if ~isempty(h_analytic), delete(h_analytic); end
            if ~isempty(h_test), delete(h_test); end


            %plot(msh.nvec, C(:, i), Color = 'b', DisplayName= 'Previous!');
            h_cur = plot(msh.nvec, C(:, i+1), Color = 'k', DisplayName=
'Numerical', HandleVisibility='on');
            h_analytic = plot(xsol, csol, 'Color', 'k', 'LineStyle', '--',
'Marker', 'x', 'LineWidth', 1, 'DisplayName', 'Analytic Solution',
HandleVisibility='off');
            h_test = text(0.2, 0.2, append('t = ', num2str(t(i))),
"FontSize",10);

            legend('show')
            pause(0.2) %pause for one second
            counter = counter + 1;
        end

    %Reinitialising global parameters
    gbl = zeros(msh.ngn);
    gblvec = zeros(msh.ngn, 1);
end

outputs = {C, t, msh};

end
```

A.19. transientstep.m

```
function C1 = transientstep(gblmats, dt, C0, theta, DBC, NBC, i, plt)
% gblmats = global matrices created using gblass.m
% dt = timestep size
% C0 = current solution vector
% theta = dictates the time stepping scheme. 0.5 for Crank, 0 for forward euler,
1 for backward euler2dc
%  DBC and NBC are Neumann boundary conditions respectively
% plt is an input to assist in debugging
%i is time ID


%A function that creates the next step solution vector using the current
%step and the theta scheme

GBLVEC = (gblmats.mass - (1-theta).*dt.* gblmats.stiff)*C0; %by definition
fterms = dt.*(theta.*gblmats.f + (1-theta).*gblmats.f); %by defition

%Apply Neumann boundary conditions
if ~any([isstring(NBC), ischar(NBC)])
    NeuGlobals = Neumann(gblmats, gblmats.BC, NBC);
    gblmats.BC = NeuGlobals{2};
    %Says NBC was executed ONCE
    if i ==1 && plt == 1
        disp('NBC was executed') %tells user if executed
    end
else
    if i ==1 && plt ==1
```

```
        disp('NBC not executed') %tells user if executed
    end

end

%By definition
NBCterms = dt.*(theta.*gblmats.BC + (1-theta).*gblmats.BC);
GBLVEC = GBLVEC + fterms +NBCterms;

GM = gblmats.mass + theta.*dt.*gblmats.stiff;

%Applying Dirichlets
if ~any([isstring(DBC), ischar(DBC)])
    DirGlobals = Dirichlet(GM, GBLVEC, DBC);
    GM = DirGlobals{1};
    GBLVEC = DirGlobals{2};

    %Says DBC was executed or not ONCE
    if i ==1 && plt == 1
        disp('DBC was executed') %tells user if executed
    end
else
    if i ==1 && plt == 1
        disp('DBC not executed') %tells user if executed
    end

end

C1 = GM\GBLVEC; %Cnext calculated!!

end
```

A.20. DecayReactionAnalyticSoln.m

```
function c = DecayReactionAnalyticSoln(x, t)
%Analytic solution assuming
% D = 0
% f = 0
% lambda = 1

% IC of c(x, 0) = 0
%
% BC's
% NBC of 0 at both x = 0, 1
% L = 1
    c = exp(-t).*ones(length(x), 1);
end
```

A.21. DiffusionWithDecayAnalyticSoln.m

```
function c = DiffusionWithDecayAnalyticSoln(x, t)
%Analytic solution assuming
% D = 1
% f = 0
% lambda = 1

% IC of c(x, 0) = sin(pix/L)
```

```
%
% BC's
% DBC of 0 at both x = 0, 1
%
% L = 1

c = sin(pi.*x)*exp(-(1 + pi^2)*t);

end
```

A.22. DecayReactionAnalyticSoln.m

```
function c = DiffusionWithDecayingReactionAnalyticSoln(x, t)
%Analytic solution assuming
% D = 1
% f = 0
% lambda = e^-t, ie beta and lambda0 = 1

% IC of c(x, 0) = 0
%
% BC's
% DBC of 0 at both x = 0, 1
%
% L = 1
    c = exp(-(1-exp(-t))).*ones(length(x), 1);
end
```

A.23. EvalBCAnal.m

```
function [C0, NBC, DBC] = evalBCAnal(analyticsoln, msh)
%This is a function that produces the initial and boundary conditions
%corresponding to various governing equations. analyticsoln is a funciton
%handle, e.g. @TransientAnalyticSoln

%switch to set boundary conditions
switch func2str(analyticsoln)

    case  'PureSourceAnalyticSoln'
        NBC = [1, 0; 2, 0];
        DBC = 'No Dir';

    case'DiffusionWithDecayingReactionAnalyticSoln'
        NBC = [1, 0; 2, 0];
        DBC = 'No Dir';

    case 'DecayReactionAnalyticSoln'
        NBC = [1, 0; 2, 0];
        DBC = 'no dir';%[1, 0; 2, 0];

    case 'DiffusionWithDecayAnalyticSoln'
        NBC = 'No Neu';
        DBC = [1, 0; 2, 0];

end
```

```
%Switch to set initial conditions
switch func2str(analyticsoln)
    case 'TransientAnlyticSoln'
        C0 = zeros(msh.ngn, 1);
    case'DecayReactionAnalyticSoln'
        C0 = ones(msh.ngn, 1);
    case 'PureSourceAnalyticSoln'
        C0 = zeros(msh.ngn, 1);
    case 'DiffusionWithDecayingReactionAnalyticSoln'
        C0 = ones(msh.ngn, 1);
    case 'DiffusionWithDecayAnalyticSoln'
        C0 = (sin(pi.*(msh.nvec./(max(msh.nvec)-min(msh.nvec)))))';
end
end
```

A.24. materialfuncs.m

```
function [D_func,  Lin_func, f_func] = materialfuncs(analyticsoln)
%This function stores the function definitions corresponding to various
%governing ewuations. View report for more info

switch func2str(analyticsoln)
    case 'TransientAnalyticSoln'
        %Defining governing material parameter eqns and generating material
fields
        %term constants
        D = 1;
        lambda = 0;
        f = 0;

        %function handles to pass through to MatFields
        D_func = @(t, x) D;
        Lin_func = @(t, x) lambda;
        f_func = @(t, x) f;

    case 'PureSourceAnalyticSoln'
        %Defining governing material parameter eqns and generating material
fields
        %term constants
        D = 0;
        lambda = 0;
        f = 1;

        %function handles to pass through to MatFields
        D_func = @(t, x) D;
        Lin_func = @(t, x) lambda;
        f_func = @(t, x) f;

    case 'DecayReactionAnalyticSoln'
        %Defining governing material parameter eqns and generating material
fields
        %term constants
        D = 0;
        lambda = 1;
        f = 0;

        %function handles to pass through to MatFields
        D_func = @(t, x) D;
```

```
        Lin_func = @(t, x) lambda;
        f_func = @(t, x) f;

    case 'DiffusionWithDecayAnalyticSoln'
        %Defining governing material parameter eqns and generating material
fields
        %term constants
        D = 1;
        lambda = 1;
        f = 0;

        %function handles to pass through to MatFields
        D_func = @(t, x) D;
        Lin_func = @(t, x) lambda;
        f_func = @(t, x) f;

    case 'DiffusionWithDecayingReactionAnalyticSoln'
        D = 1;
        beta = 1;
        lambda = 1;
        f = 0;

        %function handles to pass through to MatFields
        D_func = @(t, x) D;
        Lin_func = @(t, x) lambda*exp(-beta*t);
        f_func = @(t, x) f;

end

end
```

A.25. PureSourceAnalyticSoln.m

```
function c = PureSourceAnalyticSoln(x, t)
%Analytic solution assuming
% D = 0
% f = 1
% lambda = 0
%
% BC's
% NBC of 0 at both x = 0, 1
% L = 1
    c = t.*ones(length(x), 1);
end
```

A.26. TransientAnalyticSoln.m

```
function [ c ] = TransientAnalyticSoln(x,t)
%TransientAnalyticSonl Analytical solution to transient diffusion equation
%   Computes the analytical solution to the transient diffusion equation for
%   the domain x=[0,1], subject to initial condition: c(x,0) = 0, and Dirichlet
%   boundary conditions: c(0,t) = 0, and c(1,t) = 1.
%   Input Arguments:
%   x is the point in space to evaluate the solution at
%   t is the point in time to evaluate the solution at
%   Output Argument:
%   c is the value of concentration at point x and time t, i.e. c(x,t)
```

```
trans = 0.0;

for k=1:1000
    trans = trans + ((((-1)^k)/k) * exp(-k^2*pi^2*t)*sin(k*pi*x));
end

c = x + (2/pi)*trans;

end
```

A.27. OneDimLinearMeshGen.m

```
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;
        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
```

Appendix B

B.1. insulatedtissuebisection.m

```
function Results = insulatedtissuebisection(matchoice, air)

%A script that uses the bisection method/binary search method to find the
%max temperature that the outside of insulation layer can be to prevent second
and
%third degree burns in 50 seconds
```

```
%This has been adapted from tissuebisection.m

%Mat choice is a string, either 'Kevlar' or 'Wool'

% 'air' input indicates whether or not air is placed as a layer in the mesh. If
air =
% 1, there is a layer of air between the insulation material and skin.
% Otherwise, the mesh will assume direct contact with skin

tic %time how long this whole thing takes

tol = 10e-12; %to account for floating point
tolT = 0.5; %Temperature tolerance

T0 = 310; % close enough to 310.15, I just don't like decimals
T1 = 800; %Arbitiarily high number to start bisection

Trange = T1-T0;

second = 1/600; %for second degree burns
third = 0.005; %for third degree burns

%For simulation
theta = 0.5; %crank N scheme
IC = 0; %Not using initial conditions!
plt = 0; %No plotting, we're short on time
xvals = [second, third];

%initialise counters
Results.noloops.second = 0;
Results.noloops.third = 0;

%Loop for both 2nd and 3rd degree burns
for i = 1:2

    counter = 0; %initialise counter

    while T1 - T0 > tolT %repeat until temperatures lay between a certain
tolerance

        surftemp0 = (T1+T0)/2; %BISECTTTTT

        %run simulation and get values
        outputs = transienttissueinsu(theta, IC, surftemp0, plt, matchoice,
air);
        msh = outputs.msh;
        C = outputs.C;
        t = outputs.t;
        dt = (t(end)-t(1))/(length(t)-1);

        CXID = find(abs(msh.nvec-xvals(i)) < tol, 1);%Find the ID of the x value

        %If it does not exist, find the index of the nearest x value in the
        %mesh
        if numel(size(CXID)) ~= 1  % Check if C is not a number
            disp('Exact x value does not exist in solution matrix')
            disp('Calculating for next closest t value')
            [~, CXID] = min(abs(msh.nvec - xvals(i)));
        else
```

```
          disp('Exact x value available')
        end

        CX = C(CXID, :); %Remove all values where the temperature is below Tcrit
        for k = 1: length(CX)
            if CX(k) < 317.15
                CX(k) = NaN;
            end
        end

        %% What's the damage?
        trapCZ.vals = (2*10^98).*exp(-(12017./(CX-273.15))); %For use in the
integral
        TX = trapCZ.vals;
        nonNaNTX = TX(~isnan(TX)); % Exclude NaNs
        %%
        int50 = dt*trapz(nonNaNTX); %damage after 50s
        %% to here

        %Figure out where to bisect next
        if int50 > 1
            Trange = Trange/2;
            T1 = T0+Trange;

        elseif int50 < 1
            Trange = Trange/2;
            T0 = T1-Trange;

        end

        counter = counter + 1;
    end

    if i == 1
        Results.TempCrit.second = [T0, T1];
        T0 = 310;
        T1 = 370;
        Trange = T1-T0;

        Results.noloops.second = counter; %store no. loops

    elseif i == 2
        Results.TempCrit.third = [T0, T1];
        T0 = 310;
        T1 = 370;
        Trange = T1-T0;

        Results.noloops.third = counter; %store no. loops


    end

end

Results.time = toc; %andddd time

close all

end
```

B.2. insulationmsh.m

```
function insulation = insulationmsh(t, xrangelayer, noem, matchoice)
% Example: Kevlarmsh = insulationmsh(t, [0, 2], 10, 'Kevlar')

%This function generates a mesh for the insulation layer, with material
%parameters stored

% t is a vestigial input, isn't hurting anyone right?
% srange layer is is the range of the mesh, MUST BE IN ASCENDING ORDER
% noem is the number of elements in the layer
%Mat choice is a string: either 'Kevlar' or 'Wool'

%NOTE that this generates a quadratic mesh

%Material parameters extracted
materials = insulationparameters(); %Made a separate function because it was
getting way too clunky

%% extracts parameters depending on material string
switch matchoice
    case 'Kevlar'
        par = materials.Kevlar;
    case 'Wool'
        par = materials.Wool;
    case 'Air'
        par = materials.Air;
end

%Calculate material parameters
D_func_insu = @(t, x) par.k/(par.rho*par.cp); %Using heat equivalent of
diffusion coefficient
Lin_func_insu = @(t, x) 0; %Assuming no reaction term absorbing or creating heat
via reaction
f_func_insu = @(t, x) 0; %Assuming no heat energy being produced

%%


insulation = QuadMeshGen(xrangelayer(1), xrangelayer(2), noem); %Genrating a
naked mesh
insulation = MatFields(insulation, t, D_func_insu, Lin_func_insu, f_func_insu);
%Mesh now has material parameters assigned to it

end
```

B.3. insulationparameters.m

```
function materials = insulationparameters()

%parameters are stored in form of [subcutaeneous, dermis, epidermis]
% A script that stores all the parameters needed for the coursework

%Kevlar
```

```
Kevlar.rho = 1450; %Density in kg/m3
Kevlar.cp = 1360; %Specific heat capacity
Kevlar.k = 25; %thermal conductivity SCALED TO BLOOD VALUES IN ASSIGNMENT

%Wool
Wool.rho = 1300; %Density in kg/m3
Wool.cp = 1360; %Specific heat capacity
Wool.k = 25; %thermal conductivity SCALED TO BLOOD VALUES IN ASSIGNMENT

%Air
Air.rho = 1.293; %Density in kg/m3
Air.cp = 1005; %Specific heat capacity
Air.k = 0.025*100; %thermal conductivity, SCALED TO BLOOD VALUES IN ASSIGNMENT

%Collecting for export
materials.Kevlar = Kevlar;
materials.Wool = Wool;
materials.Air = Air;


end
```

B.4. megamesh.m

```
function jointmesh = megamesh(mesh1, mesh2)

%This ia a function that stitches two meshes together.
% At shared nodes, function assigns NaN to material parameters so they can
% be identified in lcl_assemble.m. The removed material parameters are
% stored separately under a .doublepars child variable that can be called
% according to the element being called for

% Mesh structures need to be compatible for the script to work
% 1. msh1 must end EXACTLY where mesh2 ends. i.e. mesh1.nvec(end) =
mesh2.nvec(1);
% 2. both meshes must either have or not have material parameters assigned. YOu
can't attach a dressed mesh to an undressed one
% 3. Only two meshes at a time!!

%If loop kills script if the two meshes are not compatible for concatenate
if mesh1.nvec(end) ~= mesh2.nvec(1)
    error('Condition not met. The script will terminate.');
end
%%
%the easy stuff
jointmesh.ne = mesh1.ne + mesh2.ne; %number elements is both meshes put together
jointmesh.ngn = mesh1.ngn + mesh2.ngn - 1;  %number of nodes is both put
together, but minus one because of the ovelapping node
jointmesh.nvec = [mesh1.nvec, mesh2.nvec(2:end)]; %This is the x vector of the
two meshes put together

%%Join the material fields if they exist
if isfield(mesh1, 'matpar') && isfield(mesh2, 'matpar')
   % disp('matpars available') %for user

    %obtain the shared material parameters
```

```
    %Top row in each variable coincides with last node in mesh1, bottom row with
    %first node in mesh2
    jointmesh.doublepars.dfield = [mesh1.matpar.dfield(end, :);
mesh2.matpar.dfield(1, :)];
    jointmesh.doublepars.linfield = [mesh1.matpar.linfield(end, :);
mesh2.matpar.linfield(1, :)];
    jointmesh.doublepars.ffield = [mesh1.matpar.ffield(end, :);
mesh2.matpar.ffield(1, :)];


     % Generalized processing for `doublepars`
    if isfield(mesh1, 'doublepars') || isfield(mesh2, 'doublepars')
        doubleparsFields = {'dfield', 'linfield', 'ffield'}; % List of fields to
process

        % Ensure jointmesh.doublepars exists
        if ~isfield(jointmesh, 'doublepars')
            jointmesh.doublepars = struct();
        end

        for i = 1:length(doubleparsFields)
            field = doubleparsFields{i};

            % Initialize field in jointmesh if necessary
            if ~isfield(jointmesh.doublepars, field)
                jointmesh.doublepars.(field) = [];
            end

            % Add mesh1's `doublepars` field if it exists
            if isfield(mesh1, 'doublepars') && isfield(mesh1.doublepars, field)
                jointmesh.doublepars.(field) = [mesh1.doublepars.(field);
jointmesh.doublepars.(field)];
            end

            % Add mesh2's `doublepars` field if it exists
            if isfield(mesh2, 'doublepars') && isfield(mesh2.doublepars, field)
                jointmesh.doublepars.(field) = [jointmesh.doublepars.(field);
mesh2.doublepars.(field)];
            end
        end
    end

    % This ensures that doublepars are stored in the right form
    if ~isfield(mesh1, 'doublepars') && ~isfield(mesh2, 'doublepars')
        disp('executed');
        jointmesh.doublepars.dfield = {jointmesh.doublepars.dfield};
        jointmesh.doublepars.linfield = {jointmesh.doublepars.linfield};
        jointmesh.doublepars.ffield = {jointmesh.doublepars.ffield};
    end

else
    error('matpars not joined')
end


%%
%need to rewrite the node values for mesh2
for i = 1: mesh2.ne
    mesh2.elem(i).n = mesh2.elem(i).n + mesh1.ngn - 1;
```

```
end

jointmesh.elem = [mesh1.elem, mesh2.elem]; %concatenate the elements together
%%
% Loop that notes the node ID's coinciding with double material parameters
if isfield(mesh1, 'sharednodeid')
    jointmesh.sharednodeid = [mesh1.sharednodeid,
jointmesh.elem(mesh1.ne).n(end)];
else
    jointmesh.sharednodeid = jointmesh.elem(mesh1.ne).n(end);
end

if isfield(mesh2, 'sharednodeid')
    mesh2.sharednodeid = mesh2.sharednodeid + jointmesh.ngn-1;
    jointmesh.sharednodeid = [jointmesh.sharednodeid, mesh2.sharednodeid];
end

%loop that concatenates the material matrices, and then REMOVES the parameters
%coinciding with shared nodes.

jointmesh.matpar.dfield = [mesh1.matpar.dfield; mesh2.matpar.dfield( 2:end, :)];
jointmesh.matpar.linfield = [mesh1.matpar.linfield; mesh2.matpar.linfield(
2:end, :)];
jointmesh.matpar.ffield = [mesh1.matpar.ffield; mesh2.matpar.ffield( 2:end, :)];

for i = 1: length(jointmesh.sharednodeid)
    jointmesh.matpar.dfield(jointmesh.sharednodeid(i),:) = NaN;
    jointmesh.matpar.linfield(jointmesh.sharednodeid(i),:) = NaN;
    jointmesh.matpar.ffield(jointmesh.sharednodeid(i),:) = NaN;
end



%%
% Loop that notes how many matrices have been concatenated. Ugly but hey -
% it works
if isfield(mesh1, 'nomats')
    if isfield(mesh2, 'nomats')
        jointmesh.nomats = mesh1.nomats + mesh2.nomats;
    else
        jointmesh.nomats = mesh1.nomats + 1;
    end
elseif isfield(mesh2, 'nomats')
    jointmesh.nomats = mesh2.nomats + 1;
else
    jointmesh.nomats = 2;
end
```

B.5. plot_C_at_TTissue.m

```
function  plot_C_atTtissue(msh, C, t, tpltvals)

%A function that plots the solution vector C against x at a given set of
%times noted by tpltvals

%msh is mesh
```

```
% C is the solution matric
%t is the time vector
% tpltvals is a vector of time values the plots are wanted for , e.g. [0.1, 0.5,
8, 10 ,50]

tol = 1e-12; % Set a small tolerance for floating-point comparison
x = msh.nvec; %x axis of the plot

figure(1)
hold on
%housekeeping for a neat graph
%The boundary x values of the tissue
B = 0.01;
D = 0.005;
E = 1/600;
ymin = 310.15;
ymax = 373.15;

yrange = ymax-ymin;

ylim(sort([ymin-0.1*(yrange), ymax]))


xline (B, LineStyle="--", Color="r", Linewidth = 1.5,
HandleVisibility="off")%make just one of the tissue lines visible on legend
xline (D, LineStyle="--", Color="r", Linewidth = 1.5, HandleVisibility="off")
xline (E, LineStyle="--", Color="r", Linewidth = 1.5, HandleVisibility="off")

text((B+D)/2, ymin-0.05*yrange, 'Sub', 'HorizontalAlignment', 'center')
text((E+D)/2, ymin-0.05*yrange, 'Derm', 'HorizontalAlignment', 'center')
text((E)/2, ymin-0.05*yrange, 'Epi', 'HorizontalAlignment', 'center')

xline (B, LineStyle="--", Color="r", Linewidth = 1.5, DisplayName= 'Tissue
boundary')%make just one of the tissue lines visible on legend
xline (D, LineStyle="--", Color="r", Linewidth = 1.5, HandleVisibility="off")
xline (E, LineStyle="--", Color="r", Linewidth = 1.5, HandleVisibility="off")
%yline (ymin, LineStyle = "-", Color = "r", LineWidth = 2, DisplayName =
'Internal skin temp')

text((B+D)/2, ymin-0.05*yrange, 'Sub', 'HorizontalAlignment', 'center')
text((E+D)/2, ymin-0.05*yrange, 'Derm', 'HorizontalAlignment', 'center')
text((E)/2, ymin-0.05*yrange, 'Epi', 'HorizontalAlignment', 'center')

%This is for housekeeping the labels and titles and stuff
air = 0;
matchoice = 'Wool';

if air == 1
    thickair = 0.02;
else
    thickair = 0; %Scaled in proportion to skin
end
thickmat = 0.005;% Scaled in proportion to skin
xair = 0 - thickair;
xmat = xair-thickmat;

if thickair ~= 0 %Drawing the insulation boundary layers on
   xline(0, LineStyle="--", Color="k", Linewidth = 1, DisplayName= 'air-skin',
HandleVisibility="off");
```

```
   xline(xair, LineStyle="--", Color="k", Linewidth = 1, DisplayName= 'air-
insulation', HandleVisibility="off")

   text(xair/2, ymin-0.05*yrange, 'Air','HorizontalAlignment', 'center')
   text((xmat+xair)/2, ymin-0.05*yrange, matchoice, 'HorizontalAlignment',
'center')

else
    xline(xair, LineStyle="--", Color="k", Linewidth = 1, DisplayName= 'skin-
insulation', HandleVisibility="off")
    text((xmat)/2, ymin-0.05*yrange, matchoice, 'HorizontalAlignment', 'center')

end


colours = lines(length(tpltvals)); % Use the 'lines' colormap for distinct
colors
greyLegendAdded = false; % so all analytical solutions only have one label on
the legend


for i = 1: length(tpltvals) %plot for each time value in the tpltvals vector

    CID = find(abs(t-tpltvals(i)) < tol, 1);
    if numel(size(C)) ~= 1 % Check if C is not a number
        disp('Exact time value does not exist in solution matrix')
        disp('Calculating for next closest t value')
        [~, CID] = min(abs(t - tpltvals(i))); %If does not exist, find the next
closest value
    else
        disp('Exact time value available')
    end

    %For graph plotting
    y = C(:, CID);

    % %Making x vector for plotting exact solution
        plot(x, y, 'LineStyle', "-", 'Color', colours(i, :), 'LineWidth', 1,
'DisplayName',append('t = ', num2str(tpltvals(i))));

end

% Add the legend after plotting, more housekeeping
legend('show', 'Location', 'best');

titlename = append('Temperature against x, Wool with direct contact');

thetascheme = append('θ = ', num2str(0.5));
noems = append('no. elements = ', num2str(msh.ne));
dt = append('dt = ', num2str( (t(end)-t(1)) / (length(t)-1) ) );
parameters = {thetascheme, noems, dt};
% text(x(ceil(end/4)),y(ceil(3*end/4)), parameters);
text(x(ceil(end/4)), 0.6, parameters, ...
    'FontSize', 10, 'Color', 'black', ...
    'BackgroundColor', 'white', 'EdgeColor', 'black', ...
    'HorizontalAlignment', 'center', 'VerticalAlignment', 'middle');

title(titlename)
xlabel('x', 'FontSize', 16);
```

```
ylabel('T (K)', 'FontSize', 16);
 % Ensure grid settings are applied
grid minor;
ax = gca;
ax.XAxis.Exponent = 0; % Disable the scaling exponent
ax.XMinorTick = 'on';
ax.YMinorTick = 'on';
ax.MinorGridLineStyle = ':';
ax.MinorGridColor = [0.3, 0.3, 0.3];
ax.MinorGridAlpha = 0.7;
ax.Layer = 'top'; % Bring grid to the top
% %Save in form "Formativeplot_lambdaval_DVal_Fval
graphname = append('P1Plot_t =', num2str(t(CID)), '.png');
saveas(gcf,graphname)

end
```

B.6. plot_C_at_XTissue.m

```
function  plot_C_atXtissue(msh, C, t, xvals)

%A function that plots the solution vector C against x, using outputs
%generated from transienttissue.m or transienttissue.insu.m

%Example: plot_C_atXtissue(results.msh, results.C, results.t, [0.001, 0.005,
0.009])

%msh is mesh
% C is a given solution matrix
% t is a time vector
% xvals is a vector of x values for plotting, e.g. xvals = [0.001, 0.005, 0.009]

tol = 1e-12; % Set a small tolerance for floating-point comparison
xplot = t; %x axis of the plot

figure(1)
hold on

colours = lines(length(xvals)); % Use the 'lines' colormap for distinct colors
greyLegendAdded = false; % so all analytical solutions only have one label on
the legend

for i = 1: length(xvals) %for all values in xval vector

    CID = find(abs(msh.nvec-xvals(i)) < tol, 1);    %Find the ID corresponding
to x value in the solution matrix
    if numel(size(CID)) ~= 1  % Check if C is not a number
        disp('Exact x value does not exist in solution matrix')
        disp('Calculating for next closest t value')
        [~, CID] = min(abs(msh.nvec - xvals(i))); %If does not exist, find next
closest x value
    else
        disp('Exact x value available')
    end

    %For graph plotting
    y = C(CID, :);
```

```
    xname = append('x = ', num2str(round(msh.nvec(CID),3))); %name of the plot
    text(median(xplot),(median(y)-0.025),xname);
        % plot(xplot, y, 'LineStyle', "-", 'Marker', "x", 'MarkerSize', 8,
'Color', colours(i, :), 'LineWidth', 1, 'DisplayName', xname);
        % plot(xplot, y, 'LineStyle', "-", 'Color', colours(i, :), 'LineWidth',
1, 'DisplayName', xname);
        plot(xplot, y, 'LineStyle', "-", 'Color', colours(i, :), 'LineWidth', 1,
'HandleVisibility', 'off');


end

% Add the legend after plotting
% legend('show', 'Location', 'best');
ylim([310 380]);
xlim([0 0.5])
text(0.3, 350,'x = 0.001')
text(0.4, 330, 'x = 0.002')
text (0.03, 370, 'dt = 0.05s')
%titlename = append('Temperature Plot at x = ',num2str(round(msh.nvec(CID),
3)),' no elems (epi, derm, sub) = 3, 3, 3');
%titlename = append('Temperature - x plot, ',num2str(round(msh.nvec(CID), 3)),'
no elems (epi, derm, sub) = 3, 3, 3');
titlename = 'Temperature - t plot at select x, no. elems(epi, derm, sub) = 5, 5,
5';

thetascheme = append('θ = ', num2str(0.5));
noems = append('no. elements = ', num2str(msh.ne));
dt = append('dt = ', num2str( (t(end)-t(1)) / (length(t)-1) ) );
parameters = {thetascheme, noems, dt};
% text(x(ceil(end/4)),y(ceil(3*end/4)), parameters);
text(0.8, 0.6, parameters, ...
    'FontSize', 10, 'Color', 'black', ...
    'BackgroundColor', 'white', 'EdgeColor', 'black', ...
    'HorizontalAlignment', 'center', 'VerticalAlignment', 'middle');
% text(0.05, 0.8, parameters, ...
%     'FontSize', 10, 'Color', 'black', ...
%     'BackgroundColor', 'white', 'EdgeColor', 'black', ...
%     'HorizontalAlignment', 'center', 'VerticalAlignment', 'middle');

title(titlename)
xlabel('t (s)', 'FontSize', 16);
ylabel('C', 'FontSize', 16);
 % Ensure grid settings are applied
grid minor;
ax = gca;
ax.XMinorTick = 'on';
ax.YMinorTick = 'on';
ax.MinorGridLineStyle = ':';
ax.MinorGridColor = [0.3, 0.3, 0.3];
ax.MinorGridAlpha = 0.7;
ax.Layer = 'top'; % Bring grid to the top
% %Save in form "Formativeplot_lambdaval_DVal_Fval
% graphname = append('P1_X_Plot dt = ', num2str(dt), '.png');
% saveas(gcf,graphname)

end
```

B.7. set_BC_Tissue.m

```
function [NBC, DBC] = set_BC_tissue(IC)
%Function that uses UI to ask if you want to set boundary conditions or
%not. USed mostly for code development, but not in the final product

if IC == 1
    DBC = [1, 310.15; 2, 373.15];
    NBC = 'no NBCs';
%(x = B, t) = 310.15K
%T(x = 0, t) = 373.15K
else


%Applying boundary conditionss, tweaked from 'set_all_BC function'
%Neumann BC section of code
answer = questdlg('Enforce Neumann conditions? (D(dc/dx) known at given xmin
and/or xmax)','Yes','No');
% Handle response
switch answer
    case 'Yes'
        calc_Neu = 1;
    case 'No'
        calc_Neu = 2;
end

if calc_Neu ==1
    NBC = setNBC();
else
    NBC = 'no NBCs'; %overwrites previously stored NBC's
end

% %Dirichlet BC section of code
answer = questdlg('Enforce dirichlet? (C known at given xmin and/or
xmax)','Yes','No');
% Handle response
switch answer
    case 'Yes'
        calc_Dir = 1;
    case 'No'
        calc_Dir = 2;
end

%%
if calc_Dir ==1
    DBC = setDBC();
else
    DBC = 'No DBCs';
end

end
```

B.8. tissue.m

```
function tissuemesh = tissue(t, B, D, E)
```

```
%Function that assembles the tissue mesh
% t is a vestigial input from another life, don't worry about it too much
% B, D, E are the tissue layer boundaries as defined in the assignment
% brief

%Number of elements in each layer
noel_sub = 5;
noel_derm = 5;
noel_epi = 5;


[epi, derm, sub] = tissueparameters(); %Made a separate function because it was
getting way too clunky


%%
% 'material parameters, by definition

%epidermis
D_func_epi = @(t, x) (epi.k)/(epi.rho*epi.c);
Lin_func_epi = @(t, x) 0; %(epi.G*epi.rhob*epi.cb)/(epi.rho*epi.c);
f_func_epi = @(t, x) 0; %(epi.G*epi.rhob*epi.cb)/(epi.rho*epi.c)*epi.Tb;

%Dermis
D_func_derm = @(t, x) (derm.k)/(derm.rho*derm.c);
Lin_func_derm = @(t, x) (derm.G*derm.rhob*derm.cb)/(derm.rho*derm.c);
f_func_derm = @(t, x) (derm.G*derm.rhob*derm.cb)/(derm.rho*derm.c)*derm.Tb;

%sub
D_func_sub = @(t, x) (sub.k)/(sub.rho*sub.c);
Lin_func_sub = @(t, x) (sub.G*sub.rhob*sub.cb)/(sub.rho*sub.c);
f_func_sub = @(t, x) (sub.G*sub.rhob*sub.cb)/(sub.rho*sub.c)*sub.Tb;



%%
%Generate a mesh for subcutaneous layer
mesh_sub = QuadMeshGen(B, D, noel_sub);
mesh_sub = MatFields(mesh_sub, t, D_func_sub, Lin_func_sub, f_func_sub);

%Generate a mesh for the Dermis Layer
mesh_derm = QuadMeshGen(D, E, noel_derm);
mesh_derm = MatFields(mesh_derm, t, D_func_derm, Lin_func_derm, f_func_derm);

%Generate a mesh for the Epidermis Layer
mesh_epi = QuadMeshGen(E, 0, noel_epi);
mesh_epi = MatFields(mesh_epi, t, D_func_epi, Lin_func_epi, f_func_epi);

%%Join them together

tissuemesh1 = megamesh(mesh_sub, mesh_derm);
tissuemesh = megamesh(tissuemesh1, mesh_epi);


end
```

B.9. tissuebisection.m

```
function Results = tissuebisection()
%A script that uses the bisection method
% method to find the
%max temperature that the outside of the skin can be to prevent second and
%third degree burns in 50 seconds

%The bisection method is an algorithm similiar to the binary algorithm, but
%deals with continuous functions as opposed to sorted arrays

tic %Time how long the process takes

tol = 10e-12; %Accounting for floating points
tolT = 0.5; %Temperature tolerance

%Initialising the algorithm. Assume that Toptimal exists within this range
%where a burn would not occur
T0 = 310.15; %Bottom range of bisection
T1 = 373.15; %Top of range of bisection
Trange = T1-T0;

%X values for second and third degree burns
second = 1/600;
third = 0.005;

%For simulation
theta = 0.5; %Crank Nicolson scheme
IC = 0; %Not using initial conditions as we are constantly changing the exposure
temperature
plt = 0; %Don;t need transient plots, we're only interested in the data
xvals = [second, third];

Results.noloops.second = 0; %Initialise loops
Results.noloops.third = 0; %Initialise loops

for i = 1:2 %Loop and optimise for both second and third degree burns

    counter = 0; %Initialise counter
    while T1 - T0 > tolT %Loop until range of two consecutive temperatures falls
within a certain tolerance

        surftemp0 = (T1+T0)/2; %Initialise the starting temperature by bisecting
the temperature range

        %Producing outputs
        outputs = transienttissue(theta, IC, surftemp0, plt);
        msh = outputs.msh;
        C = outputs.C;
        t = outputs.t;
        dt = (t(end)-t(1))/(length(t)-1);

        CXID = find(abs(msh.nvec-xvals(i)) < tol, 1); %Locate the ID within the
solution matrix C corresponding with the x values for second or third deg burns

        if numel(size(CXID)) ~= 1  % Check if C is not a number
            disp('Exact x value does not exist in solution matrix')
            disp('Calculating for next closest t value')
            [~, CXID] = min(abs(msh.nvec - xvals(i))); %If does not exist, find
the index of closest x value
        else
```

```
          disp('Exact x value available')
        end

        CX = C(CXID, :); %Remove all values where the temperature is below
Tcrit, i.e. 317.15 K
        for k = 1: length(CX)
            if CX(k) < 317.15
                CX(k) = NaN;%If lower than Tcrit, reassign as NaN so easily
identified/omitted in damage calculations
            end
        end

        %% Calculating damage
        trapCZ.vals = (2*10^98).*exp(-(12017./(CX-273.15))); %For use in the
integral
        TX = trapCZ.vals;
        nonNaNTX = TX(~isnan(TX)); % Exclude NaNs
        int50 = dt*trapz(nonNaNTX); %Total damage at 50 s

        %% Decide which half to bisect next depending on magnitude of the
calculated damage
        if int50 > 1
            Trange = Trange/2;
            T1 = T0+Trange;

        elseif int50 < 1
            Trange = Trange/2;
            T0 = T1-Trange;

        end

        counter = counter + 1; %Add to counter
    end

    %At end of loop, save the final temperatures under respective names
    if i == 1 %second degree burn
        Results.TempCrit.second = [T0, T1];
        T0 = 310;
        T1 = 370;
        Trange = T1-T0;

        Results.noloops.second = counter;

    elseif i == 2 %Third degree burn
        Results.TempCrit.third = [T0, T1];
        T0 = 310;
        T1 = 370;
        Trange = T1-T0;

        Results.noloops.third = counter;


    end

end

Results.time = toc; %How long process took to run

close all
```

```
end
```

B.10. tissueburn.m

```
function results = tissueburn(outputs, xval)
%A function that calculates the time at which the Integral BIGT equals one
%at a specified x value 'xval

% Uses the binary search algorithm as it burn damage is a essentially a sorted
array

% e.g. burnresults = tissueburn((results from transienttissue.m), 1/600)

%Extract the parameters
msh = outputs.msh;
C = outputs.C;
t = outputs.t;
dt = (t(end)-t(1))/(length(t)-1);

tol = 1e-12; %account for floating point
CXID = find(abs(msh.nvec-xval) < tol, 1); %find the x ID in the mesh

if numel(size(CXID)) ~= 1  % Check if C is not a number
    disp('Exact x value does not exist in solution matrix')
    disp('Calculating for next closest t value ')
    [~, CXID] = min(abs(msh.nvec - xval)); %If does not exist, find the ID of
the value closest to it
else
  disp('Exact x value available')
end

%This is the vector we're going to use the algorithm on
CX = C(CXID, :);

%Remove all values less than tburn
for i = 1: length(CX)
    if CX(i) < 317.15
        CX(i) = NaN;
    end
end

%Start of burning is the first time it exceeds 317.15K
IDburnstart = find(~isnan(CX), 1);

%%

trapCZ.vals = (2*10^98).*exp(-(12017./(CX-273.15))); %For use in the integral
trapCZ.IDs = 1:1:length(CX); %Id's for binary search

TX = trapCZ.vals; %Initialise for integral loop
IDs = trapCZ.IDs; %Initialise the ID's for bit testing
testIDcur = IDs(end);

no_bits = 0;
ints = 200;
damage = ints(end);
```

```
looplog = 'initiate';
IDscur = IDs;

%Keep looping until we find the ID
while  damage ~= 1 && length(IDs) > 1 && no_bits < 50 %&& length(IDs)

    IDs = IDscur;

    % Calculate the middle index
    middle = floor(length(IDs) / 2);

    % Split into two halves
    IDsLeft = IDs(1:middle);% First half of vector
    IDsRight= IDs(middle+1:end);% Second half of vector

    %Decide which half to investigate
    if damage > 1
        IDscur = IDsLeft;
        looplog = [looplog, '>1,'];
    elseif damage < 1
        IDscur = IDsPrevRight;
        looplog = [looplog, '<1,'];
    end

    %Reassigning values for next loop
    IDsPrevRight = IDsRight;

    if length(IDscur) >= 1
        testIDcur = IDscur(end);
    else
        testIDcur = testIDprev-1;
    end
    %% Whats the damage
    nonNaNdam = TX(1:(testIDcur)); % Select the range first
    nonNaNdam = nonNaNdam(~isnan(nonNaNdam)); % Exclude NaNs
    damage = dt * trapz(nonNaNdam); %integral at the timestep

    %Damage at previous timestep
    nonNaNminus1 = TX(1:(testIDcur-1)); % Select the range first
    nonNaNminus1 = nonNaNminus1(~isnan(nonNaNminus1)); % Exclude NaNs
    % Perform the integral with trapz
    damageminus1 = dt * trapz(nonNaNminus1);

    %Damage two timesteps ago
    nonNaNminus2 = TX(1:(testIDcur-2)); % Select the range first
    nonNaNminus2 = nonNaNminus2(~isnan(nonNaNminus2)); % Exclude NaNs
    % Perform the integral with trapz
    damageminus2 = dt * trapz(nonNaNminus2); %integral at the timestep before

    %Damage at the NEXT timestep
    if testIDcur <= length(TX)-1
        nonNaNplus = TX(1:(testIDcur+1)); % Select the range first
        nonNaNplus = nonNaNplus(~isnan(nonNaNplus)); % Exclude NaNs
        % Perform the integral with trapz
        damageplus1 = dt * trapz(nonNaNplus); %integral at the timestep after
    end

    ints = [ints, damage]; %Save values to see rate of convergence
    no_bits = no_bits + 1;
```

```
    testIDprev = testIDcur;
end

if no_bits > 49
    disp('infinite loop') %for debugging
end

%Housekeeping the data
if damageminus1 > 1
    damageplus1 = damage;
    damage = damageminus1;
    damageminus1 = damageminus2;
end

if damage < 1
    damageminus1 = damage;
    damage = damageplus1;
    damageplus1 = 'Not calculated';
end

% the results
results.damageplus1 = damageplus1;
results.damage = damage;
results.damageminus1 = damageminus1;
results.damagevals = ints;

results.looplog = looplog;
results.ID = testIDcur;
results.IDscur = IDscur;
results.lengthID = length(IDs);

results.tburnstart = t(IDburnstart);
results.tburntotal = t(results.ID);
results.tburnexp = results.tburntotal-results.tburnstart;

results.burn50 =  dt*trapz(TX(~isnan(TX))); % Exclude NaNs

results.loops = no_bits;
end
```

B.11. tissueparameters.m

```
function [epi, derm, sub] = tissueparameters()

%parameters are stored in form of [subcutaeneous, dermis, epidermis]
% A script that stores all the parameters needed for the coursework

%Epidermis
epi.k = 25;
epi.G = 0;
epi.rho = 1200;
epi.c = 3300;
epi.rhob = 0; %technically NaN
epi.cb = 0; %technically NaN
epi.Tb = 0; %technically NaN

%Dermis
```

```
derm.k = 40;
derm.G = 0.0375;
derm.rho = 1200;
derm.c = 3300;
derm.rhob = 1060;
derm.cb = 3770;
derm.Tb = 310.15;

%Subcutaneous
sub.k = 20;
sub.G = 0.0375;
sub.rho = 1200;
sub.c = 3300;
sub.rhob = 1060;
sub.cb = 3770;
sub.Tb = 310.15;

end
```

B.12. transienttissue.m

```
% A script that is extremely similiar to summative.m, capturing part of the
assignment coinciding with modelling tissue.
% For the part of the assignment relating to tissue insulation, view
% transienttissue.m
function outputs = transienttissue(theta, IC, surftemp, plt)

% Example:
% P2Q1 = transienttissue(0.5, 1, '-', 1) %Runs simulation for P2Q1 in the
assignment

% theta = dictates the time stepping scheme. 0.5 for Crank, 0 for forward euler,
1 for backward euler2dc
% IC = 1, simulation will run using the conditions specified in P1Q2
% surftemp is the temperature that the tissue is exposed to
% plt indicates whether transient plot will be made whilst simulation is
% running. It plot == 1, user will be able to see a constantly updating
% plot, otherwise, data will be generated, but no plot will occur (useful for
running optimisation studies).

%outputs: the mesh of the tissue, the solution matrix of the entire
%simulation, and the time vector


close all

%Initialise time vector
tstart = 0;
tend = 10;
dt = 0.01;
t = tstart:dt:tend;

%The boundary x values of the tissue
B = 0.01;
D = 0.005;
E = 1/600;
```

```
%Assemble tissue mesh using tissue function
msh = tissue(t, B, D, E);


%Getting into the loop
%Initialise global matrices and vectors using number of nodes in mesh
gbl = zeros(msh.ngn);
gblvec = zeros(msh.ngn, 1);

C = zeros(msh.ngn, length(t)); %Initialise solution for all timesteps
C0 = 310.15*ones(msh.ngn, 1); %Setting IC of solution into transient matrix
C(:, 1) = C0; %placing solution IC into transient matrix

%initialise figure for plots
figure;
hold on;
%% housekeeping for a neat graph plot
ymin = 310.15;
if IC == 1
    ymax = 373.15;
else
    ymax = surftemp;
end
yrange = ymax-ymin;
%%
ylim(sort([ymin-0.1*(yrange), ymax]))
%
%Drawing boundary lines
xline (B, LineStyle="--", Color="r", Linewidth = 1.5, DisplayName= 'Tissue
boundary')%make just one of the tissue lines visible on legend
xline (D, LineStyle="--", Color="r", Linewidth = 1.5, HandleVisibility="off")
xline (E, LineStyle="--", Color="r", Linewidth = 1.5, HandleVisibility="off")
yline (ymin, LineStyle = "-", Color = "r", LineWidth = 2, DisplayName =
'Internal skin temp')

text((B+D)/2, ymin-0.05*yrange, 'Sub', 'HorizontalAlignment', 'center')
text((E+D)/2, ymin-0.05*yrange, 'Derm', 'HorizontalAlignment', 'center')
text((E)/2, ymin-0.05*yrange, 'Epi', 'HorizontalAlignment', 'center')

% Initialize plot handles for constantly updatign figure
h_cur = []; % Handle for current
h_prev = []; % Handle for the previous timstep
% Ensure grid settings are applied
grid minor;
ax = gca;
ax.XMinorTick = 'on';
ax.YMinorTick = 'on';
ax.MinorGridLineStyle = ':';
ax.MinorGridColor = [0.3, 0.3, 0.3];
ax.MinorGridAlpha = 0.7;
ax.Layer = 'top'; % Bring grid to the top

counter = 0; %counter for graph plotting

for i = 1: length(t)-1 %loop for all timesteps

    if i == 1
        if IC == 1 %Setting the boundary conditions
```

```
            [NBC, DBC] = set_BC_tissue(IC);
        else
            %For calculating critical surface temperature
            DBC = [1, 310.15; 2, surftemp];
            disp('surftemp used')
            NBC = 'no NBCs';
        end

    end

   %% Calculating the initial condition
   gblmats = gblass(msh, gbl, gblvec, i); %assembling global arrays

   C1 = transientstep(gblmats, dt, C0, theta, DBC, NBC, i, plt); %calculting
next solution vector
   C(:, i+1) = C1; %Adding solution of next timestep into matrix
   C0 = C1; %Reassigning C0

       % Plot C(i+1) every so and so loops, but not forever
       if mod(i, 5) == 0  && counter < 50 && plt == 1
           %Delete the previous plots for neatness
           if ~isempty(h_cur), delete(h_cur); end
           if ~isempty(h_prev), delete(h_prev); end

           h_cur = plot(msh.nvec, C(:, i+1), Color = 'k', DisplayName=
'Current');
           h_prev = plot(msh.nvec, C(:, i), Color = 'b', DisplayName=
'Previous!');

           legend('show')
           pause(0.2) %pause so can see what's going on
           counter = counter + 1;
       end

    %Reinitialising global parameters
    gbl = zeros(msh.ngn);
    gblvec = zeros(msh.ngn, 1);
end

%Assigning values
outputs.msh = msh;
outputs.C = C;
outputs.t = t;
```

B.13. transienttissueinsu.m

```
%Script encapsulating Summative assignment

%%
%Script that is extremely similiar to summative.m and transienttissue.m,
%but now includes insulation. Main difference is in the graph plotting bit

function outputs = transienttissueinsu(theta, IC, surftemp, plt, matchoice, air)

% Example:
% resultsusingKevlarwithAirlayer = transienttissue(0.5, 0, 400, 1, 'Kevlar,
% 1) %Runs simulation using Kevlar as insulation, assuming a gap of air
```

```
% between insulation and tissue

% theta = dictates the time stepping scheme. 0.5 for Crank, 0 for forward euler,
1 for backward euler2dc
% IC = 1, simulation will run using the conditions specified in P1Q2
% surftemp is the temperature that the tissue is exposed to
% plt indicates whether transient plot will be made whilst simulation is
% running. It plot == 1, user will be able to see a constantly updating
% plot, otherwise, data will be generated, but no plot will occur (useful for
running optimisation studies).
% Mat choice is the choice of insulation: either 'Kevlar' or 'Wool'

%outputs: the mesh of the tissue, the solution matrix of the entire
%simulation, and the time vector
close all

%initialise time vector
tstart = 0;
tend = 50;
dt = 0.01;
t = tstart:dt:tend;

%The boundary x values of the tissue
B = 0.01;
D = 0.005;
E = 1/600;

%The boundary x values of the insulative layers
noemair = 5; %Number of elements in air mesh
noemmat = 5; %Number of elements in insulative material mesh
if air == 1
    thickair = 0.02;%Scaled in proportion to skin
else
    thickair = 0; %Scaled in proportion to skin
end
thickmat = 0.005;% Scaled in proportion to skin

%For making the insulation meshes
xair = 0 - thickair; %for mesh generation
xmat = xair-thickmat; %for mesh generation
xrangeair = [0, xair];
xrangemat = [xair, xmat];

%Generating the tissue mesh WITHOUT insulation
tissuemesh = tissue(t, B, D, E);

%% Making the msh corresponding to insulation, and stitching it to the tissue
mesh
if thickair > 0
mshair = insulationmsh(t, xrangeair, noemair, 'Air'); %If air is included as a
layer
mshinsu = insulationmsh(t, xrangemat, noemmat, matchoice); %layer relating to
material

msh = megamesh(tissuemesh, mshair); %Ordering like this because precedence
matters!!
msh = megamesh(msh, mshinsu); %Fully assembled insulated layer with material
parameters
```

```
else
    mshinsu = insulationmsh(t, xrangemat, noemmat, matchoice);
    msh = megamesh(tissuemesh, mshinsu);
end


%Getting into the loop
%Initialise global matrices and vectors using number of nodes in mesh
gbl = zeros(msh.ngn);
gblvec = zeros(msh.ngn, 1);

C = zeros(msh.ngn, length(t)); %Initialise solution for all timesteps
C0 = 310.15*ones(msh.ngn, 1); %Setting IC of solution into transient matrix
C(:, 1) = C0; %placing solution IC into transient matrix


%initialise figure for plots
figure;
hold on;

%housekeeping for a neat graph plot
ymin = 310.15;
if IC == 1
    ymax = 373.15;
else
    ymax = surftemp;
end
yrange = ymax-ymin;

ylim(sort([ymin-0.1*(yrange), ymax]))

%Drawing boundary lines
xline (B, LineStyle="--", Color="r", Linewidth = 1.5, DisplayName= 'Tissue
boundary')%make just one of the tissue lines visible on legend
xline (D, LineStyle="--", Color="r", Linewidth = 1.5, HandleVisibility="off")
xline (E, LineStyle="--", Color="r", Linewidth = 1.5, HandleVisibility="off")
yline (ymin, LineStyle = "-", Color = "r", LineWidth = 2, DisplayName =
'Internal skin temp')

text((B+D)/2, ymin-0.05*yrange, 'Sub', 'HorizontalAlignment', 'center')
text((E+D)/2, ymin-0.05*yrange, 'Derm', 'HorizontalAlignment', 'center')
text((E)/2, ymin-0.05*yrange, 'Epi', 'HorizontalAlignment', 'center')

if thickair ~= 0 %Drawing the insulation boundary layers on
   xline(0, LineStyle="--", Color="k", Linewidth = 1, DisplayName= 'air-skin',
HandleVisibility="off");
   xline(xair, LineStyle="--", Color="k", Linewidth = 1, DisplayName= 'air-
insulation', HandleVisibility="off")

   text(xair/2, ymin-0.05*yrange, 'Air','HorizontalAlignment', 'center')
   text((xmat+xair)/2, ymin-0.05*yrange, matchoice, 'HorizontalAlignment',
'center')

else
    xline(xair, LineStyle="--", Color="k", Linewidth = 1, DisplayName= 'skin-
insulation', HandleVisibility="off")
    text((xmat)/2, ymin-0.05*yrange, matchoice, 'HorizontalAlignment', 'center')

end
```

```
% Initialize plot handles for a constantly updating plot
h_cur = []; % Handle for current
h_prev = []; % Handle for the previous timstep
% Ensure grid settings are applied
grid minor;
ax = gca;
ax.XMinorTick = 'on';
ax.YMinorTick = 'on';
ax.MinorGridLineStyle = ':';
ax.MinorGridColor = [0.3, 0.3, 0.3];
ax.MinorGridAlpha = 0.7;
ax.Layer = 'top'; % Bring grid to the top

counter = 0; %counter for graph plotting

%Loop for all timesteps
for i = 1: length(t)-1

    %Set boundary conditions
    if i == 1
        if IC == 1 %Setting the boundary conditions
            [NBC, DBC] = set_BC_tissue(IC);
        else
            %For calculating critical surface temperature
            DBC = [1, 310.15; 2, surftemp];
            disp('surftemp used')
            NBC = 'no NBCs';
        end

    end

   %% Calculating the initial condition and assembling global variables
   gblmats = gblass(msh, gbl, gblvec, i); %assembling global arrays

   C1 = transientstep(gblmats, dt, C0, theta, DBC, NBC, i, plt); %Calculate
Cnext
   C(:, i+1) = C1; %Adding solution of next timestep into matrix
   C0 = C1; %Reassigning C0

       % Plot C(i+1) every so and so loops, but not forever
       if mod(i, 50) == 0  && counter < 50 && plt == 1
           %Delete the previous plots for neatness
           if ~isempty(h_cur), delete(h_cur); end %remove previous plots and
plot new one
           if ~isempty(h_prev), delete(h_prev); end

           h_cur = plot(msh.nvec, C(:, i+1), Color = 'k', DisplayName=
'Current');
           h_prev = plot(msh.nvec, C(:, i), Color = 'b', DisplayName=
'Previous');

           lgd = legend('show'); %Add the legend and title it
           lgd.Title.String = 'Legend'; %Mesh boundaries
           title('Insulated model response')
           pause(0.2) %pause for one second
           counter = counter + 1;
       end
```

```
    %Reinitialising global parameters
    gbl = zeros(msh.ngn);
    gblvec = zeros(msh.ngn, 1);
end

% Assigning names to the simulation results
outputs.msh = msh;
outputs.C = C;
outputs.t = t;
```