ME40064 System Modelling and Simulation - Coursework 2

2848 Words, Candidate No. 11973, 2nd December 2025
Department of Mechanical Engineering, University of Bath

1. Introduction

Finite Element Method (FEM) is a powerful numerical technique for solving equations over a discrete domain. The simulated system is split into small regions called **elements**, connected by **nodes** which represent discrete points in the domain, together making up a **mesh**. Elements are evaluated using **basis functions** which approximate the solution within each element based on node values [1]. This approach allows for practical solutions to problems that may be difficult or impossible to solve analytically. In addition to this, the size and shape of elements can be adjusted to improve accuracy or reduce computational cost, making FEM a powerful and flexible tool for modelling (Figure 1).

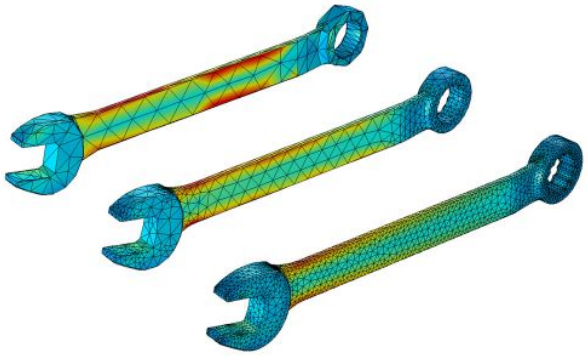

Figure 1: Finite Element Modelling of a Wrench under a Test Load Scenario [2]

This coursework focuses on the implementation and verification of a FEM solver for the transient diffusion-reaction equation, given by

$$\frac{\delta c}{\delta t} = D\frac{\delta^2 c}{\delta x^2} + \lambda c + f, \tag{1}$$

where c is the concentration level, D is the diffusion coefficient, λ is the reaction rate and f is a source term [3].

The transient diffusion-reaction equation models processes where substances diffuse through a medium while undergoing reactions or being influenced by boundary interactions. Examples of situations modelled by this equation include the transfer of heat through a material or (as explored in Part 3 of this report) the diffusion of a drug through biological tissue.

This coursework describes the development and validation of a FEM solver for the transient diffusion-reaction equation. To keep the scope manageable, the solver was implemented in 1D, using MATLAB as the scripting language [4].

2. Part 1: Software Verification

2.1. Background

A static FEM solver was implemented in a previous coursework for the steady-state diffusion-reaction equation. This solver was subsequently adapted to solve the transient form of the equation (Equation 1).

For the initial case, the values of $D = 1$ and $\lambda = 0$ were used, representing a pure diffusion scenario with linear behaviour. The **Crank-Nicolson** finite difference method was used for time integration. It has unconditional stability but no damping of oscillations, providing a good compromise between accuracy and stability at this stage [5].

The problem space was further defined with the following conditions:

Problem Space	$0 \leq x \leq 1$
Left Boundary Condition	Dirichlet: $c(0,t) = 0$
Right Boundary Condition	Dirichlet: $c(1,t) = 1$
Initial Condition	$c(x,0) = 0$

Table 1: Initial Case Conditions

These conditions have a known analytical solution, given by Equation 2:

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

The analytical solution allows for direct comparison of results between the FEM solver and expected values, providing a quantitative measure of accuracy.

2.2. Software Architecture

The solver was implemented with a modular, object-oriented software architecture to improve readability and control flow. Classes were created to encapsulate well-defined functions of the solver, such as mesh generation or plotting (Figure 2).

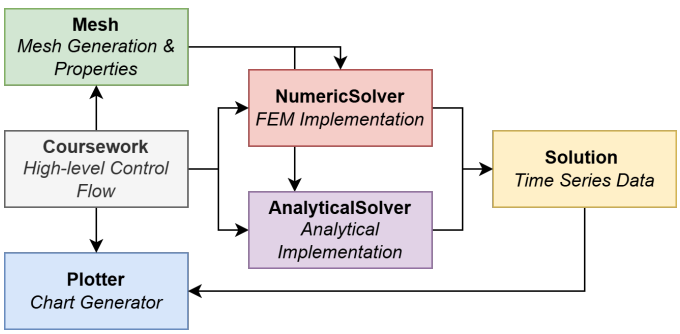

Figure 2: High-Level Software Architecture of the FEM Solver

2.3. Results

Having implemented the FEM solver as described above, a simulation was run using a mesh size of 50 elements and a time step of 0.01s, over the time period $0 < t \leq 1s$.

After this, the results were plotted on a series of charts for a visual comparison of the two solutions. The first of these were heatmaps which are an effective method for visualising the 1D diffusion over time (Figure 4, Figure 3).

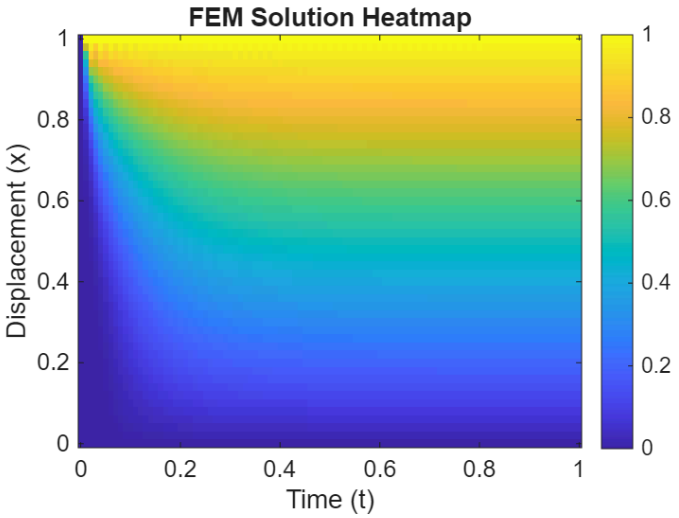

Figure 3: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

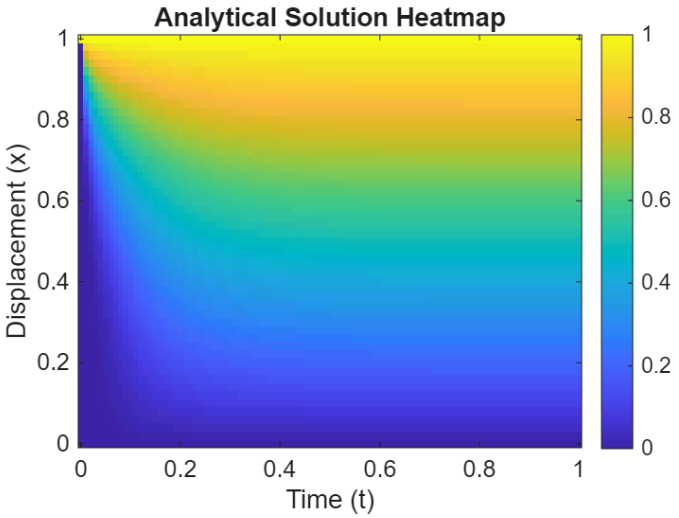

Figure 4: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

The data was also represented in a 2D plot, showing the concentration through the mesh at sample times of $t = 0.05s, 0.1s, 0.3s, 1.0s$, shown in Figure 5 and Figure 6.

Additionally, a chart was created for both solutions at a single point in the mesh ($x = 0.8$), shown in Figure 7. Unlike previous plots, this shows both methods on the same axes for direct comparison, demonstrating the agreement between the two solutions.

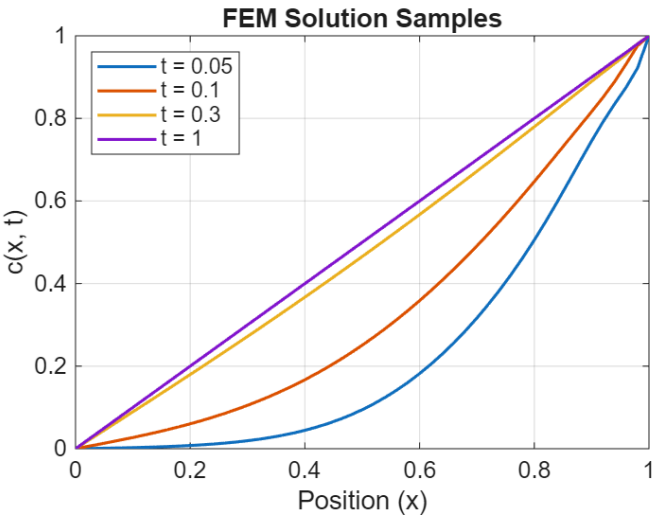

Figure 5: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

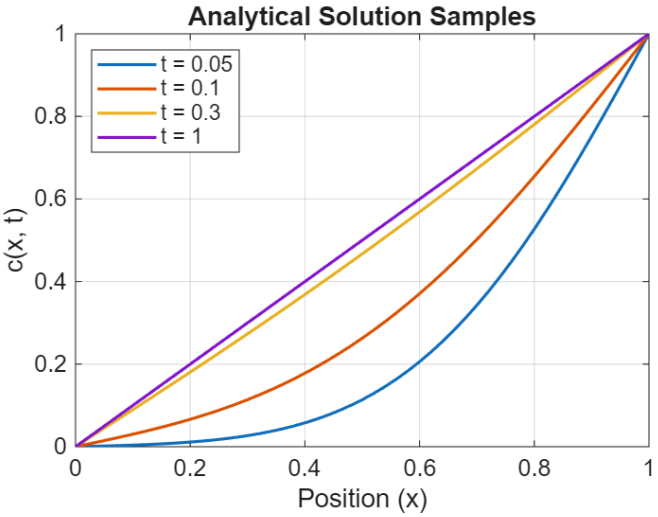

Figure 6: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

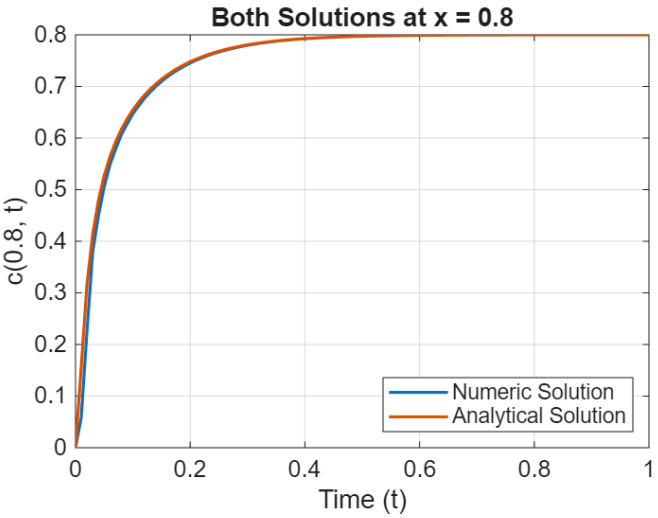

Figure 7: Comparison of Analytical and FEM Solutions at $x = 0.8$ over $0 \leq t \leq 1s$

2.4. Spacial and Temporal Convergence

To quantitatively assess the accuracy of the FEM solver, the **Root Mean Square (RMS)** error between numerical and analytical solutions was evaluated over a range of element and time step sizes. As shown in Figure 8 and Figure 9, the RMS error decreases with both smaller element sizes and smaller time steps, demonstrating convergence of the numerical solution towards the analytical solution with increasing resolution.

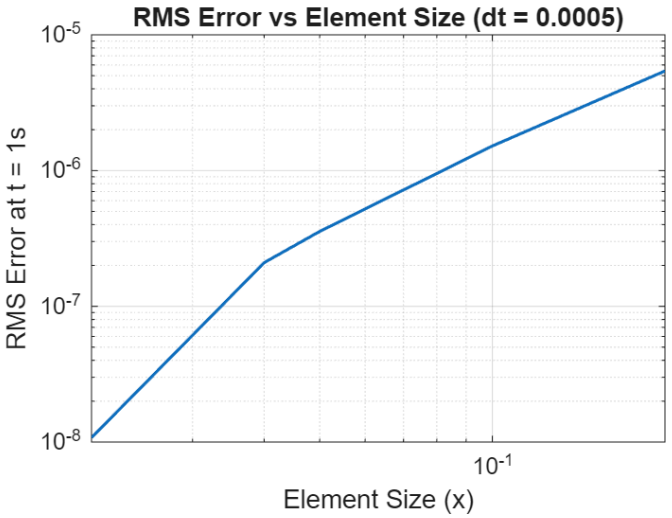

Figure 8: Comparison of RMS errors at $t = 1s$ for Varying Element Sizes

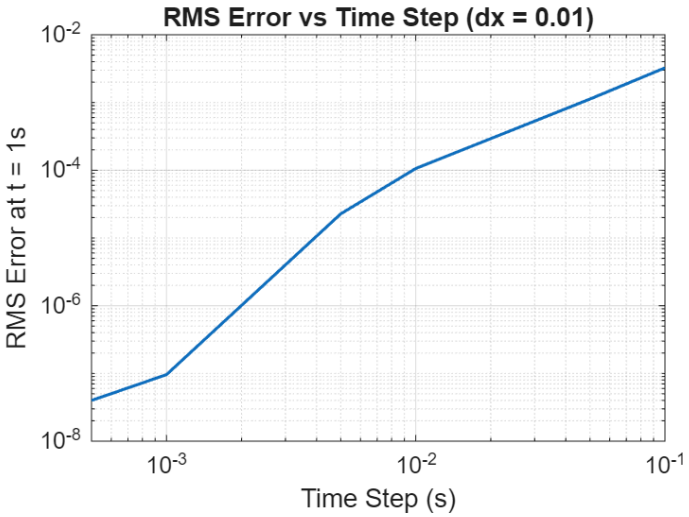

Figure 9: Comparison of RMS errors at $t = 1s$ for Varying Time Steps

2.5. Testing and Validation

A set of unit tests were created alongside the FEM solver, to verify the functionality of individual components such as mesh generation, element assembly, and time integration. As part of the development process, the project was continuously tested to ensure it passed all scenarios.

In particular, a unit test was created to validate the solver against a manufactured solution of the transient diffusion-reaction equation. This involved selecting specific values for D, λ, and f such that the solution could be expressed in a simple analytical form.

3. Part 2: Software features

3.1. Error Evaluation

In Part 1 of the coursework, the RMS error term was used to evaluate the accuracy of the FEM solver. While RMS is a useful metric, it can be sensitive to outliers and therefore may not always provide a complete picture of the solution accuracy. L2 norm doesn't suffer as much from this, and is more widely used in literature as a result [3]. To address this, a dedicated L2 error evaluation class was added to the solver, allowing for more robust error analysis.

3.2. Integration Methods

Using the L2 norm error evalutation class, the performance of three different time integration methods was compared: Forward (Explicit) Euler, Backward (Implicit) Euler, and Crank-Nicolson. The comparison test was run using a mesh with 10 elements and a time step size of 0.0001s.

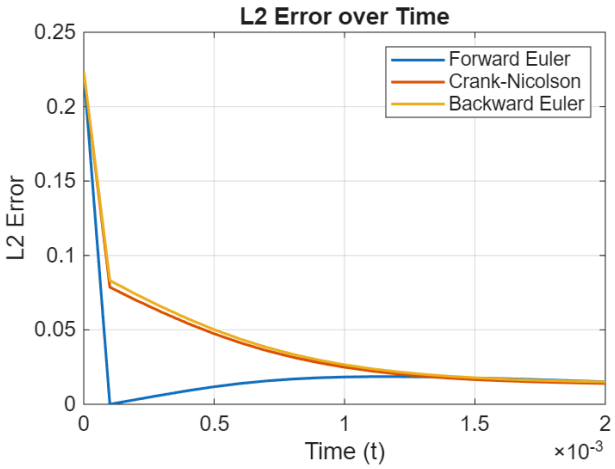

Figure 10: Comparison of L2 Errors for Different Time Integration Methods

This shows that the Forward Euler method had a higher initial accuracy, approaching the solution more quickly than the other two methods, but that it started to decrease in accuracy again afterwards. This was likely caused by instability in the method, as it is only conditionally stable.

To illustrate this further, a stability analysis was performed for all three methods, using a larger mesh of 50 elements. The stability results are shown in Table 2:

dt	Forward Euler	Backward Euler	Crank-Nicolson
0.0001	Stable	Stable	Stable
0.001	Unstable	Stable	Stable
0.01	Unstable	Stable	Stable
0.1	Unstable	Stable	Stable
0.25	Unstable	Stable	Stable

Table 2: Integration Method Stability Comparison

This shows that the Forward Euler method was only stable for very small time steps, while the other two methods demonstrated **unconditional stability**, remaining stable across all tested time steps.

For linear finite elements, the stability condition for the Forward Euler method is given by the following equation [6]:

$$dt \leq \frac{dx^2}{2D} \tag{3}$$

Therefore, for a mesh with 50 elements over the domain $0 \leq x \leq 1$ and $D = 1$, the value of dt must be no more than 0.0002s for stability, which aligns with the results shown in Table 2.

3.3. Gaussian Quadrature

So far, the solver has only been used with a simple 2-point trapezoidal integration method for evaluating element matrices. While this method is easy to implement, it treats all elements as linear, requiring meshes with high numbers of elements to achieve good accuracy for non-linear problems.

Gaussian Quadrature is an alternative integration method that can provide a more accurate result with the same number of integration points as trapezoidal integration, resulting in a more efficient solution [7].

3.4. Quadratic Basis Functions

For 2-point basis functions like those used in the coursework so far, Gaussian Quadrature with 2 points will produce an identical result to trapezoidal integration. The mesh was therefore updated to support higher-order basis functions, such as quadratic (3-point) elements, where each element has a node at each end and one in the middle.

The L2 error of a quadratic mesh with both trapezoidal and Gaussian integration methods is shown below in Figure 11:

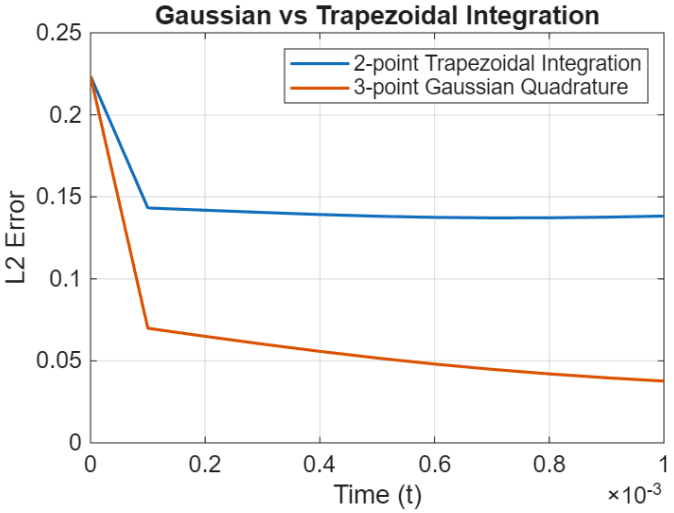

Figure 11: Comparison of L2 Errors for Gaussian Quadrature and Trapezoidal Integration

This shows a clear improvement in accuracy when using Gaussian Quadrature over trapezoidal integration with quadratic basis functions, approaching the analytical solution in a shorter time.

The reason for this improved performance is that Gaussian Quadrature evaluates the integrand at specially chosen points (Gaussian points), capturing a more accurate representation of the function being integrated. Figure 12 shows a visual comparison of a trapezoidal integration, alongside a Gaussian Quadrature with 2 Gaussian points.

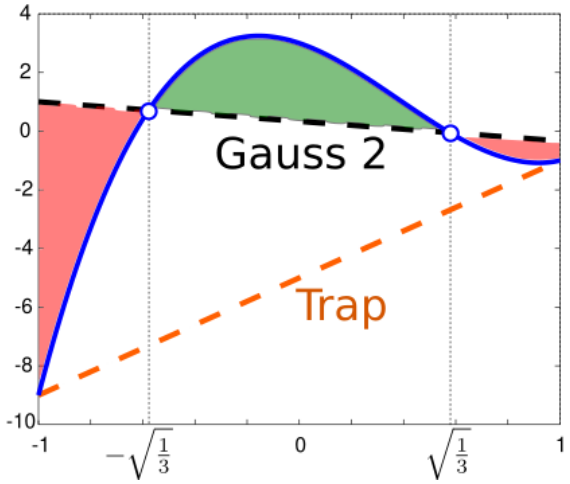

Figure 12: Visualisation of Gaussian Quadrature vs Trapezoidal Integration [8]

In the code implementation, the mesh was implemented with an arbitrary order parameter, dynamically calculating the number of nodes based off the baseline element count and order. In comparison, the code for Gaussian Quadrature was implemented with pre-defined functions for 1, 2 and 3 integration points, and shape functions for linear and quadratic elements.

3.5. Summary of Features

The addition of L2 error evaluation was an effective way to quantitatively assess the accuracy of the FEM solver, with varying configurations. It was found that the Crank-Nicolson method remained a suitable choice for time integration, balancing accuracy and stability, while the addition of Gaussian Quadrature and higher-order basis functions showed a significant improvement to solution accuracy.

Together these features enhance the capability and robustness of the FEM solver, allowing it to tackle a wider range of problems with improved accuracy and efficiency.

The robust, object-oriented software architecture introduced in Part 1 was also extended, with new classes and tests for the additional functionality.

4. Part 3: Modelling & Simulation Results

4.1. Overview

The transient FEM solver developed in Parts 1 and 2 was then applied to a practical problem: modelling the diffusion of a drug through a multilayer skin structure, as shown in the diagram below:

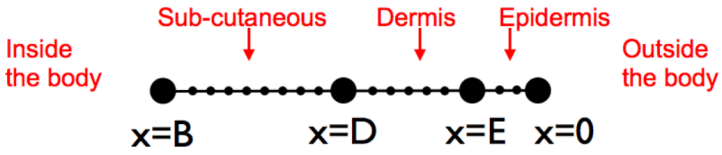

Figure 13: 1D Multilayer Finite Element Mesh of Skin Tissue Layers [9]

The concentration of the drug is modelled by the transient diffusion-reaction equation

$$\frac{\delta c}{\delta t} = D\frac{\delta^2 c}{\delta x^2} - \beta c - \gamma c, \tag{4}$$

where c is the drug concentration, D is the diffusion coefficient, β is the extra-vascular diffusivity, and γ is the drug degradation rate. For the purposes of modelling, β and γ are combined into a single reaction rate term i.e $\lambda = \beta + \gamma$, as they both act as sink terms that reduce the drug concentration.

4.2. Solver Modification

The main difference between the skin application and previous problems is the use of a multilayer mesh. A new **MultilayerMesh** class was created, inheriting from the original **Mesh** class, and overriding a method to generate a mesh made up of discrete layers (**MeshLayer** class), each with different properties.

In addition to variable diffusion and reaction rates, a 'density ratio' property was added to each layer, allowing for a non-uniform distribution of elements across the mesh. Thinner layers can therefore be assigned a higher density ratio, resulting in a local mesh with higher resolution, and improved solution accuracy.

This was implemented in three passes. First, the total density of all layers was calculated. Then, the number of elements in each layer was found by multiplying the total element count by the ratio of the layer density to total density. As an example, if there were two layers with density ratios of 1 and 3 respectively, and a total of 40 elements, the layers would be assigned 10 and 30 elements respectively. After this, the node co-ordinates were generated for each layer in sequence, with a uniform distribution according to the number of elements assigned to that layer, and it's range of x values.

4.3. Simulation Results

The modified solver was then configured to use solve the coursework-specified problem, with the following conditions:

Problem Space	$0 \leq x \leq 0.01$
Left Boundary Condition	Dirichlet: $c(0,t) = 30$
Right Boundary Condition	Dirichlet: $c(0.01,t) = 0$
Initial Condition	$c(x,0) = 0$

Table 3: Drug Concentration Problem Conditions

This used a multilayer mesh with three layers representing the epidermis, dermis and sub-cutaneous tissue, with the following parameters:

Parameter	**Epidermis**	**Dermis**	**Sub-Cutaneous**
x Range	$0 \leq x < 0.00166667$	$0.00166667 \leq x < 0.005$	$0.005 \leq x \leq 0.01$
D	4e-6	5e-6	2e-6
β	0.0	0.01	0.01
γ	0.02	0.02	0.02
Density Ratio	2.0	1.0	1.0

Table 4: Mesh Layer Parameters

The simulation was then run using an initial mesh size of 50 elements and a time step of 0.01s, over the specified time period of $0 < t \leq 30s$.

The results were plotted as a heatmap (Figure 14), showing the diffusion of the drug through the multilayer skin structure over time. Additionally marked on this plot are the approximate boundaries between each layer.

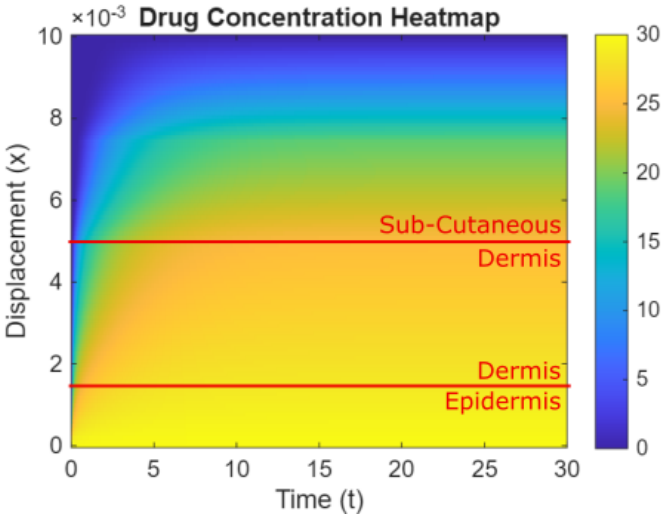

Figure 14: FEM Solution of Drug Diffusion through Multilayer Skin Structure over $0 \leq x \leq 0.01$ and $0 \leq t \leq 30s$

A stable profile is visible after around 10 seconds, with the epidermis layer almost immediately saturated to a high level, and the dermis soon after with a slightly lower concentration. The sub-cutaneous layer shows a much more gradual concentration gradient, mainly due to the Dirichlet boundary at $x = 0.01$ of $c(0.01, t) = 0$ forcing a perfect sink along the far edge of the mesh.

4.4. Dose Evaluation

After establishing a baseline simulation profile, the next step was to calculate the effectiveness of the drug diffusion. This was evaluated using the **kappa** metric, defined as the integral of the concentration above a threshold level over a specified time period, given by

$$K = \int_{t_{\text{eff}}}^{t=30} c \; dt \tag{5}$$

where t_{eff} is the time at which the concentration first exceeds the threshold level (here 4.0), $t = 30$ is the end of the simulation period, and c is the drug concentration at a target point in the mesh.

A new class, **DoseEvaluator**, was created to provide this functionality, with a static method **EvaluateSolution** that returns the kappa value for a given solution dataset, target location and threshold concentration. The logic is quite simple; first the node closest to the target location is identified, and then the concentration values for that node are checked against the target threshold. If the concentration exceeds the threshold, the solution is integrated between the time this occurs and the end of the simulation, using a trapezoidal method.

4.5. Minimum Dose Search

The **DoseEvaluator** class was then modified to add a new function, **FindMinimumDose**. The purpose of this was to indentify the minimum required dose of the drug that would achieve a sufficient value of kappa.

This process was achieved with a **binary search** algorithm, which iteratevely narrows down the range of possible dose values, converging on the minimum effective dose. It relies on having a correct initial upper and lower bound for the dose, but provides an effective search method with relatively few iterations.

The minimum effective dose, c_{DOSE}, was defined as the value at which the concentration at $x = 0.005$ exceeds a threshold of $K > 1000$. A search was run with an initial search range of $0 \leq c \leq 100$, using a tolerance of 0.1 for convergence. The results of this search are shown below in Figure 15:

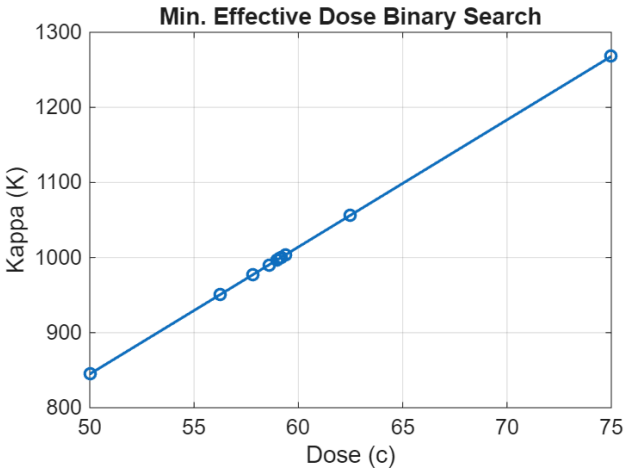

Figure 15: Binary Search for Minimum Effective Dose

This shows that the relationship between dose and kappa is linear, and that the minimum effective dose was found to be approximately **59.18**.

4.6. Dose Sensitivity Analysis

A sensitivity analysis was then performed on the model, investigating the impact of varying the diffusion coefficient D, extra-vascular diffusivity β, and drug degradation rate γ on the concentration at $x = 0.005$ over time, and the resulting dose effectiveness K (Kappa).

4.6.1. Diffusion Coefficient

The diffusion coefficient D was investigated by scaling the original values for each layer by factors of 0.5, 0.75, 1.0, 1.5 and 2.0. The results of this analysis are shown below in Figure 16 and Figure 17, illustrating the effect of varying D on concentration over time, and of the resultant dose effectiveness. These plots show a clear trend of increasing diffusion coefficient leading to higher concentrations at the target point, and therefore higher dose effectiveness K.

This is expected, as a higher diffusion coefficient allows the drug to spread more rapidly through the tissue layers, reaching the target point in a shorter time, with less degradation.

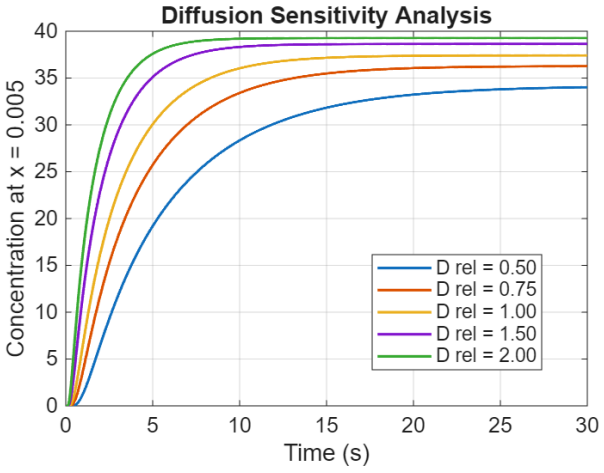

Figure 16: Concentration at $x = D$ for Varying Values of D

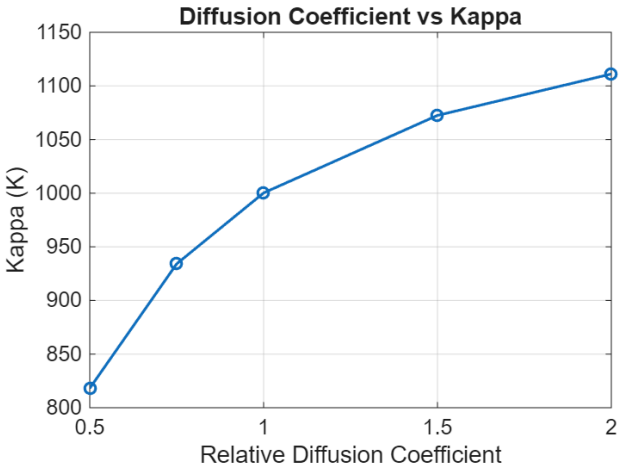

Figure 17: Dose Effectiveness for Varying Values of D

4.6.2. Extra-Vascular Diffusivity

The next parameter to be investigated was extra-vascular diffusivity (β), varied in the same way as the diffusion coefficient. The results of this analysis are shown below in Figure 18 and Figure 19.

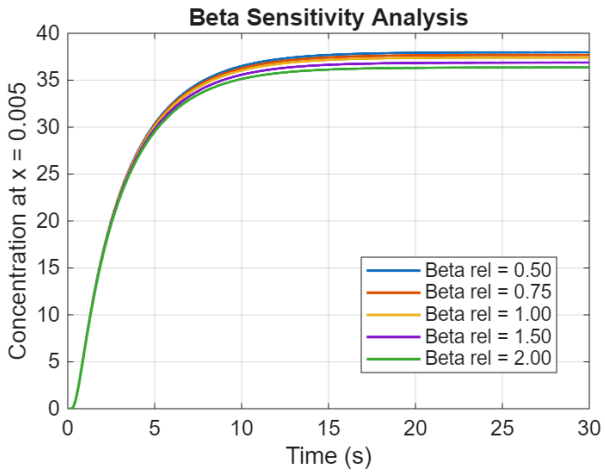

Figure 18: Concentration at $x = D$ for Varying Values of β

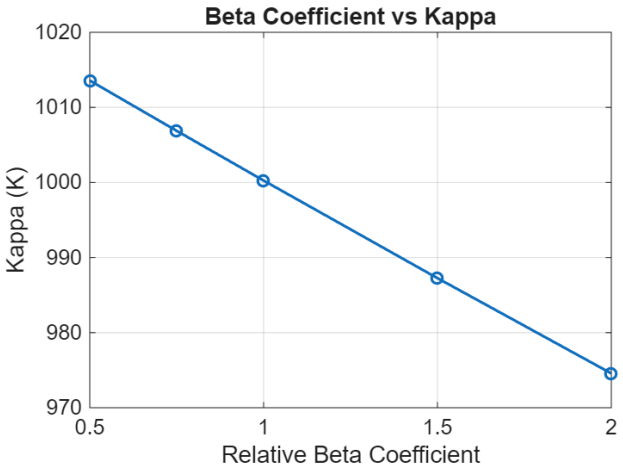

Figure 19: Dose Effectiveness for Varying Values of β

These plots show an inverse, linear relationship between β and dose effectiveness. As β increases, the faster the drug diffuses out of the vascular system into surrounding tissue, reducing the concentration at the target point and therefore lowering K.

4.6.3. Drug Degradation Rate

Finally, the drug degredation rate (γ) was investigated, again varied in the same way as previous parameters. The results of this analysis are shown below Figure 20 and Figure 21.

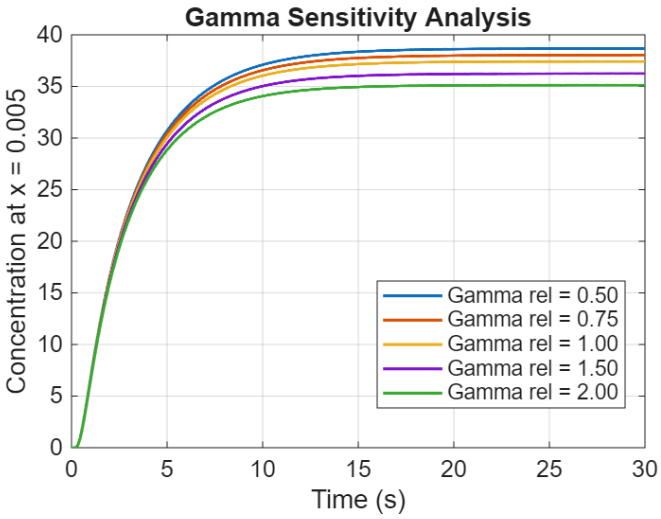

Figure 20: Concentration at $x = D$ for Varying Values of γ

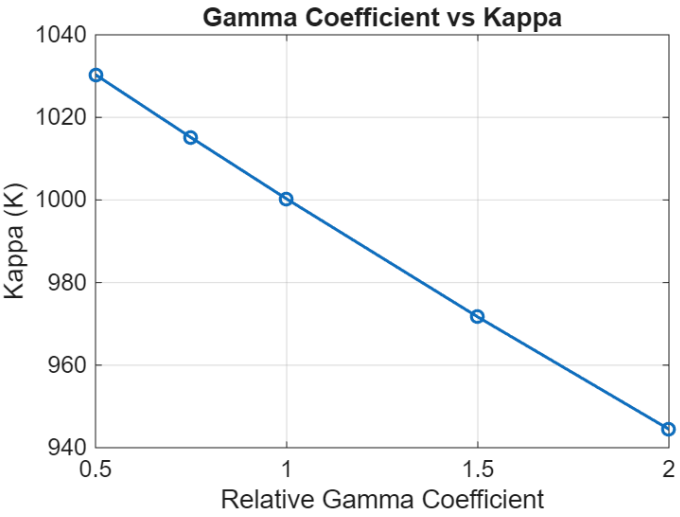

Figure 21: Dose Effectiveness for Varying Values of γ

As with β, these plots show an inverse, linear relationship between γ and dose effectiveness. A higher degradation rate results in the drug breaking down more quickly, reducing the concentration at the target point and lowering K.

From a mathematical perspective, this is also expected as both β and γ act as sink terms in the diffusion-reaction equation, reducing the overall concentration. However, as the original values of γ were larger than those of β, the impact of varying γ was more pronounced.

4.7. Further Work

While the FEM solver developed in this coursework has proved effective for modelling the 1D drug diffusion problem, there are several areas where further work could improve it's performance.

One example is the implementation of continuous diffusion-reaction parameters across the mesh, instead of the discrete steps that are currently used. This would more accuractly represent the gradual changes in tissue properties that occur in real biological systems.

Alongside this, the boundary conditions of a perfect source at one side and a perfect sink at the other are idealised scenarios. More realistic boundary conditions, such as Neumann or Robin conditions that vary over time, could be implemented to better simulate real-world situations [10].

The values chosen for the model parameters were based on the values provided in the coursework brief. However, these were stated as approximate or in some places unrealistic. More researched and physically accurate values would therefore improve the validity of the simulation results.

Increasing the mesh or time fidelity would also be likely improve accuracy, although both result in higher computational cost. More advanced meshing techniques could be explored, taking the multi-density approach further by implementing adaptive techniques that refine the mesh further in areas of high gradient.

Finally, the solver could be extended to 2D or 3D problems [11], allowing for more complex geometries and diffusion scenarios to be modelled. This would require significant changes to the mesh generation and element assembly processes, but would greatly expand the range of applications for the solver.

5. Conclusion

The FEM solver developed in this coursework was successfully implemented and validated against an analytical solution for the transient diffusion-reaction equation in Part 1. It was then improved in Part 2 with the addition of more advanced features such as L2 error evaluation, higher-order basis functions, and Gaussian Quadrature integration.

In Part 3, the solver was then applied to a practical problem, modelling the diffusion of a drug through a multilayer skin structure. The initial results and subsequent analysis demonstrated a physically plausible diffusion profile, with a binary search method effectively identifying the minimum effective drug dose level.

Changes to key parameters in a sensitivity analysis produced results that aligned with both mathematical and physical interpretations of the sample problem, further validating the solver's performance.

Finally, several areas for further work were discussed and identified, providing a roadmap for future improvements to the solver. These further demonstrate the flexibility and suitability of the FEM approach for solving complex diffusion-reaction problems.

6. References

[1] J. L. G. Dhatt G. Touzot, *Finite Element Method*. Wiley.

[2] "Finite Element Mesh Refinement." [Online]. Available: https://www.comsol.com/multiphysics/mesh-refinement

[3] W. Hundsdorfer, "Numerical Solution of Advection-Diffusion-Reaction Equations," 2000. [Online]. Available: https://bpb-us-e1.wpmucdn.com/blogs.gwu.edu/dist/9/297/files/2018/01/66bdd115ac105ea17af303e73d4fec449754-v448bk.pdf

[4] "MATLAB." [Online]. Available: https://mathworks.com/products/matlab.html

[5] C. W. T. C. Sun, "Unconditionally stable Crank-Nicolson scheme for solving two-dimensional Maxwell's equations," 2003. [Online]. Available: https://doi.org/10.1049/el:20030416

[6] C. Connaughton, "The Diffusion Equation," 2009. [Online]. Available: https://warwick.ac.uk/fac/cross_fac/complexity/study/msc_and_phd/co906/co906online/lecturenotes_2009/chap3.pdf

[7] T. Amisaki, "Gaussian Quadrature as a Numerical Integration Method for Estimating Area Under the Curve," 2001. [Online]. Available: https://www.jstage.jst.go.jp/article/bpb/24/1/24_1_70/_pdf/-char/ja

[8] Paolostar, *Comparison between 2-point Gaussian and trapezoidal quadrature.* [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_quadrature#/media/File:Comparison_Gaussquad_trapezoidal.svg

[9] A. Cookson, "Assignment Transient MATLAB-Based FEM Modelling," 2025.

[10] M. A. Esmili Sikarudi et al, "Neumann and Robin boundary conditions for heat conduction modeling using smoothed particle hydrodynamics," 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0010465515002738

[11] I. Lencina, "Comparison between 1D and 2D models to analyze the dam break wave using the FEM method and the shallow water equations," 2007.

7. Appendix - MATLAB Source Code

8. Main

8.1. main.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  main.m
% Author       :  11973
% Created      :  2025-11-24 (YYYY-MM-DD)
% License      :  MIT
% Description  :  Main function for solving transient diffusion equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function main()
    fprintf("ME40064 Coursework 2 Starting...\n");

    % Part 1: Software Verification
    Coursework.Part1Plots();
    Coursework.Part1Convergence();

    % Part 2: Software Features
    Coursework.Part2TimeIntegrationComparison();
    Coursework.Part2GaussianQuadrature();

    % Part 3: Modelling & Simulation Results
    Coursework.Part3InitialResults();
    Coursework.Part3MinimumEffectiveDose();
    Coursework.Part3DoseSensitivityAnalysis();

    fprintf("...ME40064 Coursework 2 Complete\n");
end

```

9. Coursework

9.1. Coursework.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  Coursework.m
% Author       :  11973
% Created      :  2025-11-27 (YYYY-MM-DD)
% License      :  MIT
% Description  :  Static methods for each part of the coursework.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef Coursework

    methods (Static)

        function Part1Plots()
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     Part1Plots()
        %
        % Arguments:    None
        % Returns:      None
        %
        % Description:  Runs the start Part 1 of the coursework,
        %               generating a simple mesh, running numeric and
        %               analytical solvers, and plotting the results.
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % time parameters
            tmax = 1.0;
            dt = 0.01;

            % mesh parameters
            xmin = 0.0;
            xmax = 1.0;
            element_count = 50;
            order = 1;

            % Crank-Nicholson method
            theta = 0.5;

            % diffusion and reaction coefficients
            D = 1.0;
            lambda = 0.0;

            % concentrations
            c_max = 1.0;
            c_min = 0.0;


            % generate mesh
            mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
            mesh.Generate();

            % solver parameters
            lhs_boundary = BoundaryCondition();
            lhs_boundary.Type = BoundaryType.Dirichlet;
            lhs_boundary.Value = c_min;

            rhs_boundary = BoundaryCondition();
            rhs_boundary.Type = BoundaryType.Dirichlet;
```

```
64              rhs_boundary.Value = c_max;
65
66              integration_method = IntegrationMethod();
67              integration_method.type = IntegrationType.Trapezoidal;
68              integration_method.gauss_points = 0; % not used for trapezoidal
69
70              % solve numerically
71              numeric_solution = NumericSolver.SolveNumeric(...
72                  mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
73                  @(~, ~) 0.0, integration_method);
74
75              % solve analytically
76              analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
77
78              % plot solutions as a heatmaps
79              Plotter.PlotHeatMap(numeric_solution, "FEM Solution Heatmap", ...
80                  "cw2/report/resources/part1/NumericHeatmap", c_max);
81
82              Plotter.PlotHeatMap(analytical_solution, "Analytical Solution Heatmap", ...
83                  "cw2/report/resources/part1/AnalyticalHeatmap", c_max);
84
85              % plot solution samples at specified times
86              sample_times = [0.05, 0.1, 0.3, 1.0];
87
88              Plotter.PlotTimeSamples(numeric_solution, dt, sample_times, ...
89              "FEM Solution Samples", "cw2/report/resources/part1/NumericSamples");
90
91              Plotter.PlotTimeSamples(analytical_solution, dt, sample_times, ...
92              "Analytical Solution Samples", "cw2/report/resources/part1/AnalyticalSamples");
93
94              % plot both solutions at a specific position over time
95              sample_x = 0.8;
96              legend_strings = {"Numeric Solution", "Analytical Solution"};
97
98              Plotter.PlotSampleOverTime(numeric_solution, analytical_solution, ...
99                  sample_x, "Both Solutions at x = 0.8", ...
100                 "cw2/report/resources/part1/BothX08", legend_strings);
101
102         end
103
104         function Part1Convergence()
105         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
106         %
107         % Function:     Part1Convergence()
108         %
109         % Arguments:    None
110         % Returns:      None
111         %
112         % Description:  Runs a convergence study for Part 1 of the
113         %               coursework, calculating RMS error between numeric
114         %               and analytical solutions over a range of element
115         %               counts and time steps.
116         %
117         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
118
119             % time parameters
120             tmax = 1.0;
121
122             % mesh parameters
123             xmin = 0.0;
124             xmax = 1.0;
125             element_count = 50;
126             order = 1;
127
128             % Crank-Nicholson method
129             theta = 0.5;
130
```

```
131             % diffusion and reaction coefficients
132             D = 1.0;
133             lambda = 0.0;
134
135             % concentrations
136             c_max = 1.0;
137             c_min = 0.0;
138
139
140             % generate mesh
141             mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
142             mesh.Generate();
143
144             % solver parameters
145             lhs_boundary = BoundaryCondition();
146             lhs_boundary.Type = BoundaryType.Dirichlet;
147             lhs_boundary.Value = c_min;
148
149             rhs_boundary = BoundaryCondition();
150             rhs_boundary.Type = BoundaryType.Dirichlet;
151             rhs_boundary.Value = c_max;
152
153             integration_method = IntegrationMethod();
154             integration_method.type = IntegrationType.Trapezoidal;
155             integration_method.gauss_points = 0; % not used for trapezoidal
156
157             % calculate RMS error with varying mesh sizes and time steps
158
159             element_counts = [5, 10, 20, 25, 50];
160             time_steps = [0.1, 0.05, 0.01, 0.005, 0.001, 0.0005];
161
162             num_cases = length(element_counts) * length(time_steps);
163
164             % columns: elem_count, dt, dx, RMS error
165             rms_errr_table_elem_count = zeros(num_cases, 4);
166
167             % columns: elem_count, dt, dx, RMS error
168             rms_errr_table_time_step = zeros(num_cases, 4);
169
170             k = 1;
171
172             % vary element count with fixed time step
173             for i = 1:length(element_counts)
174                 elem_count = element_counts(i);
175                 dt = 0.0005;
176
177                 % generate mesh
178                 mesh = Mesh(xmin, xmax, elem_count, order, D, lambda);
179                 mesh.Generate();
180
181                 % solve numerically
182                 numeric_solution = NumericSolver.SolveNumeric(...
183                     mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
184                     @(~, ~) 0.0, integration_method);
185
186                 % solve analytically
187                 analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
188
189                 % compute RMS error
190                 [~, final_time] = min(abs(analytical_solution.time - tmax));
191
192                 c_numeric = numeric_solution.values(:, final_time);
193                 c_analytical = analytical_solution.values(:, final_time);
194
195                 error = c_numeric - c_analytical;
196                 rms_error = sqrt(mean(error.^2));
197
```

```
198                 rms_errr_table_elem_count(k,:) = [elem_count, dt, ...
199                     (xmax-xmin)/elem_count, rms_error];
200 
201                 k = k + 1;
202 
203                 fprintf("Elements: %d, dt: %.4f, dx: %.4f, RMS Error: %.6f\n", ...
204                     elem_count, dt, (xmax-xmin)/elem_count, rms_error);
205             end
206 
207             k = 1;
208 
209             % vary time step with fixed element count
210             for j = 1:length(time_steps)
211 
212                 elem_count = 1 / 0.01;
213                 dt = time_steps(j);
214 
215                 % generate mesh
216                 mesh = Mesh(xmin, xmax, elem_count, order, D, lambda);
217                 mesh.Generate();
218 
219                 % solve numerically
220                 numeric_solution = NumericSolver.SolveNumeric(...
221                     mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
222                     @(~, ~) 0.0, integration_method);
223 
224                 % solve analytically
225                 analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
226 
227                 % compute RMS error
228                 [~, final_time] = min(abs(analytical_solution.time - tmax));
229 
230                 c_numeric = numeric_solution.values(:, final_time);
231                 c_analytical = analytical_solution.values(:, final_time);
232 
233                 error = c_numeric - c_analytical;
234                 rms_error = sqrt(mean(error.^2));
235 
236                 rms_errr_table_time_step(k,:) = [elem_count, dt, ...
237                     (xmax-xmin)/elem_count, rms_error];
238 
239                 k = k + 1;
240 
241                 fprintf("Elements: %d, dt: %.4f, dx: %.4f, RMS Error: %.6f\n", ...
242                     elem_count, dt, (xmax-xmin)/elem_count, rms_error);
243 
244             end
245 
246             % plot element counts
247             dx = rms_errr_table_elem_count(:, 3);  % element size
248             err_spatial = rms_errr_table_elem_count(:, 4);
249 
250             Plotter.PlotConvergenceError(dx, err_spatial, ...
251                 "RMS Error vs Element Size (dt = 0.0005)", ...
252                 "cw2/report/resources/part1/ElementSizeConvergence", ...
253                 "Element Size (x)", "RMS Error at t = 1s");
254 
255             % plot time steps
256             dt_vals = rms_errr_table_time_step(:, 2);  % time steps
257             err_temporal = rms_errr_table_time_step(:, 4);
258 
259             Plotter.PlotConvergenceError(dt_vals, err_temporal, ...
260                 "RMS Error vs Time Step (dx = 0.01)", ...
261                 "cw2/report/resources/part1/TimeStepConvergence", ...
262                 "Time Step (s)", "RMS Error at t = 1s");
263         end
264 
```

```
265         function Part2TimeIntegrationComparison()
266         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
267         %
268         % Function:     Part2TimeIntegrationComparison()
269         %
270         % Arguments:    None
271         % Returns:      None
272         %
273         % Description:  Runs a study comparing different time integration
274         %               methods for Part 2 of the coursework.
275         %
276         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
277
278             % time parameters
279             tmax = 0.002;
280             dt = 0.0001;
281
282             % mesh parameters
283             xmin = 0.0;
284             xmax = 1.0;
285             element_count = 10;
286             order = 1;
287
288             % diffusion and reaction coefficients
289             D = 1.0;
290             lambda = 0.0;
291
292             % concentrations
293             c_max = 1.0;
294             c_min = 0.0;
295
296              % generate mesh
297             mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
298             mesh.Generate();
299
300             % solve analytically
301             analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
302
303             % solver parameters
304             lhs_boundary = BoundaryCondition();
305             lhs_boundary.Type = BoundaryType.Dirichlet;
306             lhs_boundary.Value = c_min;
307
308             rhs_boundary = BoundaryCondition();
309             rhs_boundary.Type = BoundaryType.Dirichlet;
310             rhs_boundary.Value = c_max;
311
312             integration_method = IntegrationMethod();
313             integration_method.type = IntegrationType.Trapezoidal;
314             integration_method.gauss_points = 0; % not used for trapezoidal
315
316
317             l2_errors = [];
318
319             thetas = [0.0, 1.0, 0.5]; % Explicit Euler, Implicit Euler, Crank-Nicholson
320             method_names = {"Forward Euler", "Crank-Nicolson", "Backward Euler"};
321
322             for i = 1:length(thetas)
323                 theta = thetas(i);
324
325                 % solve numerically
326                 numeric_solution = NumericSolver.SolveNumeric(...
327                     mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0, ...
328                     integration_method);
329
330                 % compute L2 error
331                 l2_error = L2Error(analytical_solution, numeric_solution);
```

```
            l2_errors = [l2_errors, l2_error];
        end

        Plotter.PlotL2Errors(l2_errors, "L2 Error over Time", ...
            "cw2/report/resources/part2/L2ErrorTimeIntegration", ...
            method_names);

        % perform stability analysis

        tmax = 1.0;
        element_count = 50;
        dt_list = [0.0001, 0.001, 0.01, 0.1, 0.25];

        % generate mesh
        mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
        mesh.Generate();

        l2_errors_stability = [];

        for i = 1:length(thetas)
            theta = thetas(i);

            l2_errors_dt = [];

            for j = 1:length(dt_list)
                dt = dt_list(j);

                try

                    fprintf("Testing %s with dt = %.4f...\n", method_names{i}, dt);

                    numeric_solution = NumericSolver.SolveNumeric(...
                        mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
                        @(~, ~) 0.0, integration_method);

                    % CHECK FOR NaN/Inf at each timestep
                    for t_idx = 1:length(numeric_solution.time)
                        vals = numeric_solution.values(:, t_idx);
                        if any(isnan(vals))
                            fprintf("%s: NaN at step %d (t=%.4f)\n", ...
                                method_names{i}, t_idx, numeric_solution.time(t_idx));
                            break;
                        end
                        if any(isinf(vals))
                            fprintf("%s: Inf at step %d (t=%.4f)\n", ...
                                method_names{i}, t_idx, numeric_solution.time(t_idx));
                            break;
                        end

                        if max(abs(vals)) > 1e10
                            fprintf("%s: Explosion at step %d (t=%.4f), max=%.2e\n",
method_names{i}, t_idx, numeric_solution.time(t_idx), max(abs(vals)));
                            break;
                        end
                    end

                    analytical_solution = AnalyticalSolver.SolveAnalytical(...
                        mesh, tmax, dt);

                    l2_error = L2Error(analytical_solution, numeric_solution);

                    l2_errors_dt = [l2_errors_dt, l2_error];
                catch
                    l2_errors_dt = [l2_errors_dt, NaN];
                    fprintf("%s EXPLODED \n", method_names{i});
                end
            end
```

```
398
399             l2_errors_stability = [l2_errors_stability; l2_errors_dt];
400         end
401
402     end
403
404     function Part2GaussianQuadrature()
405     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
406     %
407     % Function:     Part2GaussianQuadrature()
408     %
409     % Arguments:    None
410     % Returns:      None
411     %
412     % Description:  Runs a study comparing L2 error with and without
413     %               Gaussian Quadrature.
414     %
415     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
416
417
418         % time parameters
419         tmax = 0.001;
420         dt = 0.0001;
421
422         % mesh parameters
423         xmin = 0.0;
424         xmax = 1.0;
425         element_count = 5;
426         order = 2;
427
428         % diffusion and reaction coefficients
429         D = 0.5;
430         lambda = 0.0;
431
432         % concentrations
433         c_max = 1.0;
434         c_min = 0.0;
435
436         % generate mesh
437         mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
438         mesh.Generate();
439
440         % solve analytically
441         analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
442
443         % solver parameters
444
445         theta = 0.5; % Crank-Nicholson
446
447         lhs_boundary = BoundaryCondition();
448         lhs_boundary.Type = BoundaryType.Dirichlet;
449         lhs_boundary.Value = c_min;
450
451         rhs_boundary = BoundaryCondition();
452         rhs_boundary.Type = BoundaryType.Dirichlet;
453         rhs_boundary.Value = c_max;
454
455         % trapezoidal method
456         trapezoidal_method = IntegrationMethod();
457         trapezoidal_method.type = IntegrationType.Trapezoidal;
458         trapezoidal_method.gauss_points = 0; % not used for trapezoidal
459
460         trapezoidal_solution = NumericSolver.SolveNumeric(...
461             mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0, ...
462             trapezoidal_method);
463
464         % gaussian quadrature method
```

```
            gaussian_method = IntegrationMethod();
            gaussian_method.type = IntegrationType.Gaussian;
            gaussian_method.gauss_points = 3; % 3-point Gaussian quadrature

            gaussian_solution = NumericSolver.SolveNumeric(...
                mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0, ...
                gaussian_method);

            % compute L2 error
            l2_error_trapezoidal = L2Error(analytical_solution, trapezoidal_solution);
            l2_error_gaussian = L2Error(analytical_solution, gaussian_solution);
            l2_errors = [l2_error_trapezoidal, l2_error_gaussian];

            method_names = {"2-point Trapezoidal Integration", ...
                "3-point Gaussian Quadrature"};

            Plotter.PlotL2Errors(l2_errors, "Gaussian vs Trapezoidal Integration", ...
                "cw2/report/resources/part2/L2ErrorGaussianTrapezoidal", ...
                method_names);
        end

        function Part3InitialResults()
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     Part3InitialResults()
        %
        % Arguments:    None
        % Returns:      None
        %
        % Description:  Runs a basic case for Part 3 of the coursework
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % Generate mesh
            xmin = 0;
            xmax = 0.01;
            element_count = 50;
            order = 2;

            theta = 0.5; % Crank-Nicholson
            D = 1;
            lambda = 0;

            epidermis_layer = MeshLayer(0.0, 4e-6, 0.0, 0.02, 2.0);
            dermis_layer = MeshLayer(0.00166667, 5e-6, 0.01, 0.02, 1.0);
            sub_cutaneous_layer = MeshLayer(0.005, 2e-6, 0.01, 0.02, 1.0);

            layers = [epidermis_layer, dermis_layer, sub_cutaneous_layer];

            mesh = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers);
            mesh.Generate();

            tmax = 30.0;
            dt = 0.01; % works well with element_count = 50

            % concentrations
            c_max = 30.0;
            c_min = 0.0;

            lhs_boundary = BoundaryCondition();
            lhs_boundary.Type = BoundaryType.Dirichlet;
            lhs_boundary.Value = c_max;

            rhs_boundary = BoundaryCondition();
            rhs_boundary.Type = BoundaryType.Dirichlet;
            rhs_boundary.Value = c_min;

```

```
532             integration_method = IntegrationMethod();
533             integration_method.type = IntegrationType.Gaussian;
534             integration_method.gauss_points = order + 1;
535 
536             numeric_solution = NumericSolver.SolveNumeric(...
537                 mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
    integration_method);
538 
539             Plotter.PlotHeatMap(numeric_solution, "Drug Concentration Heatmap", ...
540                 "cw2/report/resources/part3/InitialNumericHeatmap", c_max);
541         end
542 
543         function Part3MinimumEffectiveDose()
544         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
545         %
546         % Function:     Part3MinimumEffectiveDose()
547         %
548         % Arguments:    None
549         % Returns:      None
550         %
551         % Description:  Runs a study to find the minimum effective dose for
552         %               Part 3 of the coursework
553         %
554         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
555 
556             % Generate mesh
557             xmin = 0;
558             xmax = 0.01;
559             element_count = 50;
560             order = 2;
561 
562             theta = 0.5; % Crank-Nicholson
563             D = 1;
564             lambda = 0;
565 
566             epidermis_layer = MeshLayer(0.0, 4e-6, 0.0, 0.02, 2.0);
567             dermis_layer = MeshLayer(0.00166667, 5e-6, 0.01, 0.02, 1.0);
568             sub_cutaneous_layer = MeshLayer(0.005, 2e-6, 0.01, 0.02, 1.0);
569 
570             layers = [epidermis_layer, dermis_layer, sub_cutaneous_layer];
571 
572             mesh = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers);
573             mesh.Generate();
574 
575             tmax = 30.0;
576             dt = 0.01; % works well with element_count = 50
577 
578             % concentrations
579             c_max = 30.0;
580             c_min = 0.0;
581 
582             lhs_boundary = BoundaryCondition();
583             lhs_boundary.Type = BoundaryType.Dirichlet;
584             lhs_boundary.Value = c_max;
585 
586             rhs_boundary = BoundaryCondition();
587             rhs_boundary.Type = BoundaryType.Dirichlet;
588             rhs_boundary.Value = c_min;
589 
590             integration_method = IntegrationMethod();
591             integration_method.type = IntegrationType.Gaussian;
592             integration_method.gauss_points = order + 1;
593 
594             % Find minimum dose
595             [c_dose_min, dose_vals, kappa_vals] = DoseEvaluator.FindMinimumDose(...
596                 mesh, tmax, dt, theta, integration_method, ...
597                 0.005, 4.0, 1000.0);
```

```
598
599             Plotter.PlotDoseEffectiveness(dose_vals, kappa_vals, ...
600                 "Min. Effective Dose Binary Search", ...
601                 "cw2/report/resources/part3/MinDoseBinarySearch", ...
602                 "Dose (c)", "Kappa (K)");
603
604             fprintf("Minimum Effective Dose: %.2f\n", c_dose_min);
605         end
606
607         function Part3DoseSensitivityAnalysis()
608         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
609         %
610         % Function:     Part3DoseSensitivityAnalysis()
611         %
612         % Arguments:    None
613         % Returns:      None
614         %
615         % Description:  Runs a study to plot the sensitivity of dose to
616         %               diffusion, beta, and gamma coefficients for
617         %               Part 3 of the coursework
618         %
619         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
620
621             % common mesh parameters
622             xmin = 0;
623             xmax = 0.01;
624             element_count = 50;
625             order = 2;
626
627             tmax = 30.0;
628             dt = 0.01; % works well with element_count = 50
629
630             % concentrations
631             c_max = 59.18;
632             c_min = 0.0;
633
634             theta = 0.5; % Crank-Nicholson
635             D = 1;
636             lambda = 0;
637
638
639             lhs_boundary = BoundaryCondition();
640             lhs_boundary.Type = BoundaryType.Dirichlet;
641             lhs_boundary.Value = c_max;
642
643             rhs_boundary = BoundaryCondition();
644             rhs_boundary.Type = BoundaryType.Dirichlet;
645             rhs_boundary.Value = c_min;
646
647             integration_method = IntegrationMethod();
648             integration_method.type = IntegrationType.Gaussian;
649             integration_method.gauss_points = order + 1;
650
651             % diffusion coefficients analysis
652             relative_diffusions = [0.5, 0.75, 1.0, 1.5, 2.0];
653
654             target_x = 0.005; % use x = D as an example
655
656             x_values = [];
657             y_values = zeros(length(relative_diffusions), tmax/dt + 1);
658
659             kappa_values = [];
660
661             for d = 1:length(relative_diffusions)
662
663                 D_rel = relative_diffusions(d);
664
```

```
665                 epidermis_layer = MeshLayer(0.0, 4e-6 * D_rel, 0.0, 0.02, 2.0);
666                 dermis_layer = MeshLayer(0.00166667, 5e-6 * D_rel, 0.01, 0.02, 1.0);
667                 sub_cutaneous_layer = MeshLayer(0.005, 2e-6 * D_rel, 0.01, 0.02, 1.0);
668
669                 layers = [epidermis_layer, dermis_layer, sub_cutaneous_layer];
670
671                 mesh = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers);
672                 mesh.Generate();
673
674                 numeric_solution = NumericSolver.SolveNumeric(...
675                     mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
676                     @(~, ~) 0.0, integration_method);
677
678                 kappa_values = [kappa_values, DoseEvaluator.EvaluateSolution(...
679                     numeric_solution, 0.005, 4.0, 0.0)];
680
681                  % first, find the closest node to the target
682                 node_index = 0;
683
684                 for i = 1:numeric_solution.mesh.node_count
685                     x = numeric_solution.mesh.node_coords(i);
686                     if x >= target_x
687                         node_index = i;
688                         break;
689                     end
690                 end
691
692                 if d == 1
693                     x_values = numeric_solution.time;
694                 end
695
696                 y_values(d, :) = numeric_solution.values(node_index, :);
697             end
698
699             legend_strings = [];
700
701             for d = 1:length(relative_diffusions)
702                 legend_strings = [legend_strings, ...
703                     sprintf("D rel = %.2f", relative_diffusions(d))];
704             end
705
706             fprintf("Plotting Dose Sensitivity Analysis...\n");
707
708             Plotter.PlotSensitivityAnalysis(x_values, y_values, ...
709                 "Diffusion Sensitivity Analysis", ...
710                 "cw2/report/resources/part3/DiffusionSensitivityAnalysis", ...
711                 "Time (s)", "Concentration at x = 0.005", legend_strings);
712
713             Plotter.PlotKappaValues(relative_diffusions, kappa_values, ...
714                 "Diffusion Coefficient vs Kappa", ...
715                 "cw2/report/resources/part3/DiffusionKappa", ...
716                 "Relative Diffusion Coefficient", "Kappa (K)");
717
718
719             % beta coefficients analysis
720
721             relative_betas = [0.5, 0.75, 1.0, 1.5, 2.0];
722
723             target_x = 0.005; % use x = D as an example
724
725             x_values = [];
726             y_values = zeros(length(relative_betas), tmax/dt + 1);
727             kappa_values = [];
728
729             for d = 1:length(relative_betas)
730
731                 beta_rel = relative_betas(d);
```

```
732
733                 epidermis_layer = MeshLayer(0.0, 4e-6, 0.0 * beta_rel, 0.02, 2.0);
734                 dermis_layer = MeshLayer(0.00166667, 5e-6, 0.01 * beta_rel, 0.02, 1.0);
735                 sub_cutaneous_layer = MeshLayer(0.005, 2e-6, 0.01 * beta_rel, 0.02, 1.0);
736
737                 layers = [epidermis_layer, dermis_layer, sub_cutaneous_layer];
738
739                 mesh = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers);
740                 mesh.Generate();
741
742                 numeric_solution = NumericSolver.SolveNumeric(...
743                     mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
744                     @(~, ~) 0.0, integration_method);
745
746                 kappa_values = [kappa_values, ...
747                     DoseEvaluator.EvaluateSolution(numeric_solution, 0.005, 4.0, 0.0)];
748
749                  % first, find the closest node to the target
750                 node_index = 0;
751
752                 for i = 1:numeric_solution.mesh.node_count
753                     x = numeric_solution.mesh.node_coords(i);
754                     if x >= target_x
755                         node_index = i;
756                         break;
757                     end
758                 end
759
760                 if d == 1
761                     x_values = numeric_solution.time;
762                 end
763
764                 y_values(d, :) = numeric_solution.values(node_index, :);
765             end
766
767             legend_strings = [];
768
769             for d = 1:length(relative_betas)
770                 legend_strings = [legend_strings, ...
771                     sprintf("Beta rel = %.2f", relative_betas(d))];
772             end
773
774             fprintf("Plotting Dose Sensitivity Analysis...\n");
775
776             Plotter.PlotSensitivityAnalysis(x_values, y_values, ...
777                 "Beta Sensitivity Analysis", ...
778                 "cw2/report/resources/part3/BetaSensitivityAnalysis", ...
779                 "Time (s)", "Concentration at x = 0.005", legend_strings);
780
781             Plotter.PlotKappaValues(relative_betas, kappa_values, ...
782                 "Beta Coefficient vs Kappa", ...
783                 "cw2/report/resources/part3/BetaKappa", ...
784                 "Relative Beta Coefficient", "Kappa (K)");
785
786             % gamma coefficients analysis
787             relative_gammas = [0.5, 0.75, 1.0, 1.5, 2.0];
788
789             target_x = 0.005; % use x = D as an example
790
791             x_values = [];
792             y_values = zeros(length(relative_gammas), tmax/dt + 1);
793             kappa_values = [];
794
795
796             for d = 1:length(relative_gammas)
797
798                 gamma_rel = relative_gammas(d);
```

```
799
800                 epidermis_layer = MeshLayer(0.0, 4e-6, 0.0, 0.02 * gamma_rel, 2.0);
801                 dermis_layer = MeshLayer(0.00166667, 5e-6, 0.01, 0.02 * gamma_rel, 1.0);
802                 sub_cutaneous_layer = MeshLayer(0.005, 2e-6, 0.01, 0.02 * gamma_rel, 1.0);
803
804                 layers = [epidermis_layer, dermis_layer, sub_cutaneous_layer];
805
806                 mesh = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers);
807                 mesh.Generate();
808
809                 numeric_solution = NumericSolver.SolveNumeric(...
810                     mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
811                     @(~, ~) 0.0, integration_method);
812                 kappa_values = [kappa_values, ...
813                     DoseEvaluator.EvaluateSolution(numeric_solution, 0.005, 4.0, 0.0)];
814
815
816                  % first, find the closest node to the target
817                 node_index = 0;
818
819                 for i = 1:numeric_solution.mesh.node_count
820                     x = numeric_solution.mesh.node_coords(i);
821                     if x >= target_x
822                         node_index = i;
823                         break;
824                     end
825                 end
826
827                 if d == 1
828                     x_values = numeric_solution.time;
829                 end
830
831                 y_values(d, :) = numeric_solution.values(node_index, :);
832             end
833
834             legend_strings = [];
835
836             for d = 1:length(relative_gammas)
837                 legend_strings = [legend_strings, ...
838                 sprintf("Gamma rel = %.2f", relative_betas(d))];
839             end
840
841             fprintf("Plotting Dose Sensitivity Analysis...\n");
842
843             Plotter.PlotSensitivityAnalysis(x_values, y_values, ...
844                 "Gamma Sensitivity Analysis", ...
845                 "cw2/report/resources/part3/GammaSensitivityAnalysis", ...
846                 "Time (s)", "Concentration at x = 0.005", legend_strings);
847
848             Plotter.PlotKappaValues(relative_gammas, kappa_values, ...
849                 "Gamma Coefficient vs Kappa", ...
850                 "cw2/report/resources/part3/GammaKappa", ...
851                 "Relative Gamma Coefficient", "Kappa (K)");
852
853         end
854
855     end
856
857 end
858
```

10. Mesh

10.1. Mesh.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  Mesh.m
% Author       :  11973
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a one-dimensional mesh for
%                 finite element analysis.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef Mesh < handle
    % inherit from handle to allow pass-by-reference

    properties

        xmin            double % min x coordinate
        xmax            double % max x coordinate
        dx              double % element size

        order           double % element order (1=linear, 2=quadratic etc)

        D               double % diffusion coefficient
        lambda          double % reaction coefficient

        node_count      uint64 % total number of global nodes
        node_coords     double % coordinates of global nodes

        element_count   uint64
        elements        MeshElement % array of mesh elements

    end

    methods

        function obj = Mesh(xmin, xmax, element_count, order, D, lambda)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     Mesh()
        %
        % Arguments:    parameters to initialise
        % Returns:      Mesh handle
        %
        % Description:  Initialises a one-dimensional mesh object
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % assign properties
            obj.xmin = xmin;
            obj.xmax = xmax;
            obj.dx = (xmax - xmin) / element_count;
            obj.D = D;
            obj.lambda = lambda;

            obj.order = order;

            % total number of nodes
            obj.node_count = (element_count * order) + 1;
            obj.node_coords = zeros(1, obj.node_count);

            obj.element_count = element_count;
```

```
            obj.elements = MeshElement.empty(element_count, 0);

        end

        function obj = Generate(obj)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     Generate()
        %
        % Arguments:    Mesh handle
        % Returns:      Mesh handle
        %
        % Description:  Generates the mesh for the given object
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            disp('Generating normal mesh...');

            % generate uniform node coordinates
            obj.node_coords = linspace(obj.xmin, obj.xmax, obj.node_count);

            % generate elements
            for e = 1:obj.element_count

                % determine global node IDs for this element
                node_start = (e - 1) * obj.order + 1;
                node_ids = node_start:(node_start + obj.order);

                %  coordinates for this element
                coords = obj.node_coords(node_ids);

                % create MeshElement object
                obj.elements(e) = MeshElement(...
                    node_ids, coords, obj.order, obj.D, obj.lambda);
            end
        end

    end
end

```

10.2. MeshElement.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  MeshElement.m
% Author       :  11973
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a one-dimensional mesh element
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef MeshElement

    properties

        order       uint8   % polynomial order (1 = linear, 2 = quadratic)
        node_ids    uint64  % global node IDs
        node_coords double  % node coordinates
        jacobian    double  % element jacobian d(x)/d(xi)
        D           double  % diffusion coefficient
        lambda      double  % reaction coefficient
```

```
    end

    methods

        function obj = MeshElement(ids, coords, order, D, lambda)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     MeshElement()
        %
        % Arguments:    parameters to initialise
        % Returns:      MeshElement handle
        %
        % Description:  Initialises a one-dimensional mesh element object
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % assign properties
            obj.node_ids = ids;
            obj.node_coords = coords;
            obj.order = order;
            obj.D = D;
            obj.lambda = lambda;

            % linear mapping from [-1, 1] to [x1, x2]
            % jacobian = dx/dxi = (x2 - x1) / 2
            obj.jacobian = (coords(end) - coords(1)) / 2;

        end
    end
end


```

10.3. MultilayerMesh.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File          :  MultilayerMesh.m
% Author        :  11973
% Created       :  2025-11-26 (YYYY-MM-DD)
% License       :  MIT
% Description   :  A class defining a multilayer one-dimensional mesh for
%                  finite element analysis.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


classdef MultilayerMesh < Mesh
    % inherit from Mesh

    properties
        layers              MeshLayer % array of layer properties
        total_density       double % total density ratio across all layers
    end

    methods

        function obj = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     MultilayerMesh()
        %
        % Arguments:    initialisation parameters including layers
```

```
31          % Returns:      MultilayerMesh handle
32          %
33          % Description:  Initialises a multilayer one-dimensional mesh object
34          %
35          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
36
37              % call superclass constructor
38              obj = obj@Mesh(xmin, xmax, element_count, order, D, lambda);
39              obj.layers = layers;
40
41
42              % recalculate nodes and element counts based on layer densities
43              obj.total_density = 0.0;
44
45              for l = 1:length(layers)
46                  obj.total_density = obj.total_density + layers(l).density_ratio;
47              end
48
49              obj.element_count = 0;
50
51              for l = 1:length(layers)
52
53                  layer_density = obj.layers(l).density_ratio;
54                  layer_element_count = round(...
55                      (layer_density / obj.total_density) * element_count);
56
57                  obj.layers(l).element_count = layer_element_count;
58
59                  % starting index for this layer
60                  obj.layers(l).layer_offset = obj.element_count + 1;
61
62                  obj.element_count = obj.element_count + layer_element_count;
63              end
64
65              obj.node_count = (obj.element_count * order) + 1;
66              obj.node_coords = zeros(1, obj.node_count);
67
68              obj.elements = MeshElement.empty(obj.element_count, 0);
69
70          end
71
72          function obj = Generate(obj)
73          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
74          %
75          % Function:     Generate()
76          %
77          % Arguments:    MultilayerMesh handle
78          % Returns:      MultilayerMesh handle
79          %
80          % Description:  Generates the mesh for the given object,
81          %               overriding the base Mesh.Generate() method.
82          %
83          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
84
85              disp('Generating multilayer mesh...');
86
87              % generate per-layer uniform node coordinates
88
89              current_node = 1;
90
91              for l = 1:length(obj.layers)
92
93                  % calculate layer xmin and xmax
94
95                  layer_xmin = obj.layers(l).x;
96
97                  if l < length(obj.layers)
```

```
                    layer_xmax = obj.layers(l + 1).x;
                else
                    layer_xmax = obj.xmax;
                end

                layer_nodes = obj.layers(l).element_count * obj.order + 1;

                layer_coords = linspace(layer_xmin, layer_xmax, layer_nodes);


                if l == 1
                    nodes_to_add = layer_coords;
                else
                    nodes_to_add = layer_coords(2:end);  % Skip duplicate boundary node
                end

                % Add nodes to global coordinate array
                num_new_nodes = length(nodes_to_add);
                obj.node_coords(current_node : current_node + num_new_nodes - 1)...
                    = nodes_to_add;

                current_node = current_node + num_new_nodes;


            end

            % Generate elements
            for e = 1:obj.element_count

                % Determine global node IDs for this element
                node_start = (e - 1) * obj.order + 1;
                node_ids = node_start:(node_start + obj.order);

                % Coordinates for this element
                coords = obj.node_coords(node_ids);

                midpoint = (coords(1) + coords(end)) / 2;

                % Determine which layer this element is in
                layer_index = 1;
                for l = 1:length(obj.layers)
                    if midpoint >= obj.layers(l).x
                        layer_index = l;
                    end
                end

                D = obj.layers(layer_index).D;
                lambda = -(obj.layers(layer_index).beta + obj.layers(layer_index).gamma);

                % Create MeshElement object
                obj.elements(e) = MeshElement(node_ids, coords, obj.order, D, lambda);
            end
        end

    end
end


```

10.4. MeshLayer.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
```

```
% File         :  MeshLayer.m
% Author       :  11973
% Created      :  2025-11-29 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a layer in a multi-layer mesh
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef MeshLayer
    properties
        x               double  % min x coordinate for this layer
        density_ratio   double  % density ratio for this layer
        D               double  % diffusion coefficient
        beta            double  % extra-vascular diffusivity
        gamma           double  % drug degradation rate

        element_count   uint64  % number of elements in this layer
        layer_offset    uint64  % starting element index for this layer
    end

    methods

        function obj = MeshLayer(x, D, beta, gamma, density_ratio)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     MeshLayer()
        %
        % Arguments:    parameters to initialise
        % Returns:      MeshLayer handle
        %
        % Description:  Initialises a mesh layer object
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            obj.x = x;
            obj.D = D;
            obj.beta = beta;
            obj.gamma = gamma;
            obj.density_ratio = density_ratio;

            obj.element_count = 0;
        end

    end
end
```

11. Analytical

11.1. AnalyticalSolver.m

```m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  AnalyticalSolver.m
% Author       :  11973
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A static class defining an analytical solver
%                 for the transient diffusion equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef AnalyticalSolver

    methods (Static)

        function solution = SolveAnalytical(mesh, tmax, dt)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     SolveAnalytical()
        %
        % Arguments:    mesh, max time and time step
        % Returns:      solution to the equation
        %
        % Description:  Solves the transient diffusion equation using
        %               the analytical solution.
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % time vector
            time_vector = 0:dt:tmax;
            solution = Solution(mesh, time_vector);

            % loop over time steps
            for step = 1:length(time_vector)

                t = time_vector(step);
                timestep_results = zeros(1, mesh.node_count);

                % loop over nodes
                for i = 1:mesh.node_count
                    x = mesh.node_coords(i);
                    timestep_results(i) = TransientAnalyticSoln(x, t);
                end

                solution.SetValues(timestep_results, step);
            end

        end
    end
end
```

11.2. TransientAnalyticSoln.m

```m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  TransientAnalyticSoln.m
% Author       :  A. N. Cookson
% Created      :  2025-11-11 (YYYY-MM-DD)
```

```
% License      :  -
% Description  :  Analytical solution to transient diffusion equation
%                 provided in the coursework materials.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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
```

12. Plotter

12.1. Plotter.m

```m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  Plotter.m
% Author       :  11973
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A collection of static methods for plotting
%                 results for the coursework.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef Plotter

    methods (Static)

        function PlotHeatMap(solution, title_str, name, c_max)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     PlotHeatMap()
        %
        % Arguments:    Plotting parameters
        % Returns:      None
        %
        % Description:  Plots a 2D heat map of solution values
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            set(0, "DefaultAxesFontSize", 12);
            set(0, "DefaultTextFontSize", 12);

            % Plot a heat map of the solution values over time
            figure;
            imagesc(solution.time, solution.mesh.node_coords, solution.values);
            colorbar;
            xlabel("Time (t)");
            ylabel("Displacement (x)");
            caxis([0 c_max]) % lock color axis for consistency
            axis xy; % ensure y-axis is oriented correctly
            title(title_str);
            grid off;

            set(gcf, 'Position', [0, 0, 500, 350]);

            % Save figure
            saveas(gcf, name, "png");
            saveas(gcf, name, "fig");
            openfig(name + ".fig");

        end

        function PlotTimeSamples(solution, dt, time_samples, title_str, name)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     PlotTimeSamples()
        %
        % Arguments:    Plotting parameters
        % Returns:      None
        %
        % Description:  Plots a solution at multiple time samples
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
64
65              set(0, "DefaultAxesFontSize", 12);
66              set(0, "DefaultTextFontSize", 12);
67
68              figure;
69              plot_handle = 0;
70
71              for i = 1:length(time_samples)
72                  t_sample = time_samples(i);
73
74                  step_index = round(t_sample / dt) + 1; % +1 for MATLAB indexing
75
76                  plot_handle = plot(solution.mesh.node_coords, ...
77                      solution.values(:, step_index));
78                  set(plot_handle, "LineWidth", 1.5);
79
80                  hold on;
81              end
82
83              xlabel("Position (x)");
84              ylabel("c(x, t)");
85              title(title_str);
86
87              grid on;
88
89              legend_strings = cell(1, length(time_samples));
90              for i = 1:length(time_samples)
91                  legend_strings{i} = ['t = ', num2str(time_samples(i))];
92              end
93
94              legend(legend_strings, "Location", "northwest");
95
96              set(gcf, 'Position', [0, 0, 500, 350]);
97
98              % Save figure
99              saveas(gcf, name, "png");
100             saveas(gcf, name, "fig");
101             openfig(name + ".fig");
102
103         end
104
105         function PlotSampleOverTime(solution_1, solution_2, x_sample, title_str, name,
    legend_strings)
106         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
107         %
108         % Function:     PlotSampleOverTime()
109         %
110         % Arguments:    Plotting parameters
111         % Returns:      None
112         %
113         % Description:  Plots two solutions at a given spatial sample over time
114         %
115         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
116
117             set(0, "DefaultAxesFontSize", 12);
118             set(0, "DefaultTextFontSize", 12);
119
120             % find x index, +1 for MATLAB indexing
121             x_index = round((x_sample - solution_1.mesh.xmin) / (solution_1.mesh.xmax ...
122                 - solution_1.mesh.xmin) * solution_1.mesh.element_count) + 1;
123
124             figure;
125             plot_handle = plot(solution_1.time, solution_1.values(x_index, :));
126             set(plot_handle, "LineWidth", 1.5);
127
128             hold on;
129
```

```
            plot_handle = plot(solution_2.time, solution_2.values(x_index, :));
            set(plot_handle, "LineWidth", 1.5);

            xlabel("Time (t)");

            ylabel("c(" + num2str(x_sample) + ", t)");
            title(title_str);

            grid on;

            legend(legend_strings, "Location", "southeast");
            set(gcf, 'Position', [0, 0, 500, 350]);

            % Save figure
            saveas(gcf, name, "png");
            saveas(gcf, name, "fig");
            openfig(name + ".fig");

        end

        function PlotConvergenceError(x_values, y_values, title_str, name, x_label,
    y_label)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     PlotConvergenceError()
        %
        % Arguments:    Plotting parameters
        % Returns:      None
        %
        % Description:  Plots convergence error on a log-log scale
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            set(0, "DefaultAxesFontSize", 12);
            set(0, "DefaultTextFontSize", 12);

            figure;

            plot_handle = loglog(x_values, y_values);
            set(plot_handle, "LineWidth", 1.5);

            xlabel(x_label);
            ylabel(y_label);
            title(title_str);
            grid on;

            set(gcf, 'Position', [0, 0, 500, 350]);

            % Save figure
            saveas(gcf, name, "png");
            saveas(gcf, name, "fig");
            openfig(name + ".fig");

        end

        function PlotL2Errors(l2_errors, title_str, name, legend_strings)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     PlotL2Errors()
        %
        % Arguments:    Plotting parameters
        % Returns:      None
        %
        % Description:  Plots L2 errors over time for multiple simulations
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```
            set(0, "DefaultAxesFontSize", 12);
            set(0, "DefaultTextFontSize", 12);

            figure;

            for i = 1:length(l2_errors)
                l2_error = l2_errors(i);
                plot_handle = plot(l2_error.time, l2_error.l2_error);
                set(plot_handle, "LineWidth", 1.5);
                hold on;
            end

            xlabel("Time (t)");
            ylabel("L2 Error");
            title(title_str);

            grid on;

            legend(legend_strings, "Location", "northeast");
            set(gcf, 'Position', [0, 0, 500, 350]);

            % Save figure
            saveas(gcf, name, "png");
            saveas(gcf, name, "fig");

            openfig(name + ".fig");

        end

        function PlotTwoConvergenceLines(x_values, y1_values, y2_values, title_str, name,
    x_label, y_label, legend_strings)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     PlotTwoConvergenceLines()
        %
        % Arguments:    Plotting parameters
        % Returns:      None
        %
        % Description:  Plots two convergence lines on a log-log scale
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            set(0, "DefaultAxesFontSize", 12);
            set(0, "DefaultTextFontSize", 12);

            figure;

            loglog(x_values, y1_values, '-o', 'LineWidth', 1.5, 'MarkerSize', 8);
            hold on;
            loglog(x_values, y2_values, '-s', 'LineWidth', 1.5, 'MarkerSize', 8);

            xlabel(x_label);
            ylabel(y_label);
            title(title_str);
            legend(legend_strings, 'Location', 'best');
            grid on;

            set(gcf, 'Position', [0, 0, 500, 350]);

            saveas(gcf, name, "png");
            saveas(gcf, name, "fig");
            openfig(name + ".fig");
        end

        function PlotDoseEffectiveness(dose_values, kappa_values, title_str, name, x_label,
    y_label)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
        %
        % Function:     PlotDoseEffectiveness()
        %
        % Arguments:    Plotting parameters
        % Returns:      None
        %
        % Description:  Plots dose vs effectiveness
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            set(0, "DefaultAxesFontSize", 12);
            set(0, "DefaultTextFontSize", 12);

            figure;

            plot_handle = plot(dose_values, kappa_values, "-o", 'LineWidth', 1.5);
            xlabel(x_label);
            ylabel(y_label);
            title(title_str);
            grid on;

            set(gcf, 'Position', [0, 0, 500, 350]);

            % Save figure
            saveas(gcf, name, "png");
            saveas(gcf, name, "fig");
            openfig(name + ".fig");
        end

        function PlotSensitivityAnalysis(x_values, y_values, title_str, name, x_label,
    y_label, legend_strings)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     PlotSensitivityAnalysis()
        %
        % Arguments:    Plotting parameters
        % Returns:      None
        %
        % Description:  Plots sensitivity analysis results
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            set(0, "DefaultAxesFontSize", 12);
            set(0, "DefaultTextFontSize", 12);

            figure;

            % y_values is a matrix where each row is a different series
            for i = 1:size(y_values, 1)
                plot_handle = plot(x_values, y_values(i, :));
                set(plot_handle, "LineWidth", 1.5);
                hold on;
            end

            xlabel(x_label);
            ylabel(y_label);
            title(title_str);
            legend(legend_strings, 'Location', 'best');
            grid on;

            set(gcf, 'Position', [0, 0, 500, 350]);

            % Save figure
            saveas(gcf, name, "png");
            saveas(gcf, name, "fig");
            openfig(name + ".fig");
        end
```

```
327
328         function PlotKappaValues(x_values, y_values, title_str, name, x_label, y_label)
329         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
330         %
331         % Function:     PlotKappaValues()
332         %
333         % Arguments:    Plotting parameters
334         % Returns:      None
335         %
336         % Description:  Plots kappa values
337         %
338         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
339
340             set(0, "DefaultAxesFontSize", 12);
341             set(0, "DefaultTextFontSize", 12);
342
343             figure;
344
345             plot_handle = plot(x_values, y_values, "-o");
346             set(plot_handle, "LineWidth", 1.5);
347             hold on;
348
349             xlabel(x_label);
350             ylabel(y_label);
351             title(title_str);
352             grid on;
353
354             set(gcf, 'Position', [0, 0, 500, 350]);
355
356             % Save figure
357             saveas(gcf, name, "png");
358             saveas(gcf, name, "fig");
359             openfig(name + ".fig");
360         end
361
362     end
363 end
```

13. Solution

13.1. Solution.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File          :  Solution.m
% Author        :  11973
% Created       :  2025-11-26 (YYYY-MM-DD)
% License       :  MIT
% Description   :  A class defining a solution to the transient
%                  diffusion equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef Solution < handle
    % inherit from handle to allow pass-by-reference behaviour

    properties

        mesh        Mesh    % handle to mesh object
        time        double  % time series - 1 x Nsteps
        values      double  % solution values - Nnodes x Nsteps

    end

    methods

        function obj = Solution(mesh, time_vector)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     Solution()
        %
        % Arguments:    parameters to initialise
        % Returns:      Solution handle
        %
        % Description:  Initialises a Solution object
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


            % assign properties
            obj.mesh = mesh;
            obj.time = time_vector;
            obj.values = zeros(mesh.node_count, length(time_vector));

        end


        function obj = SetValues(obj, values, step)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     SetValues()
        %
        % Arguments:    object, values to set, time step index
        % Returns:      Solution handle
        %
        % Description:  Helper function to set solution values at a
        %               given time step
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % set solution values at given time step
            obj.values(:, step) = values(:);
        end
```

```
    end
end


```

13.2. L2Error.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  L2Error.m
% Author       :  11973
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining an L2 error calculator between
%                 a numerical and reference solution
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef L2Error < handle
    % inherit from handle to allow pass-by-reference behaviour

    properties

        ref_solution    Solution % handle to reference solution object
        num_solution    Solution % handle to solution object

        time            double   % time series - 1 x Nsteps
        l2_error        double   % L2 error at each time step - 1 x Nsteps

    end

    methods


        function obj = L2Error(ref_solution, num_solution)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     L2Error()
        %
        % Arguments:    rfeference and numerical solutions
        % Returns:      L2Error handle
        %
        % Description:  Initialises an L2Error object and computes the
        %               L2 error between the two solutions
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % assign properties
            obj.ref_solution = ref_solution;
            obj.num_solution = num_solution;
            obj.time = ref_solution.time;

            % check compatibility
            if ref_solution.mesh.node_count ~= num_solution.mesh.node_count
                error('Reference and error solutions must have the same number of nodes');
            end

            if length(ref_solution.time) ~= length(num_solution.time)
                error('Reference and error solutions must have the same number of time
steps');
            end

            step_count = length(ref_solution.time);
```

```
58
59              obj.l2_error = zeros(1, step_count);
60
61              % compute L2 error at each time step
62              for step = 1:step_count
63
64                  c_ref = ref_solution.values(:, step);
65                  c_num = num_solution.values(:, step);
66                  x = ref_solution.mesh.node_coords;
67
68                  % compute L2 norm using trapezoidal rule
69                  integrand = (c_ref - c_num).^2;
70                  obj.l2_error(step) = sqrt(trapz(x, integrand));
71              end
72
73          end
74
75      end
76  end
77
78
```

13.3. DoseEvaluator.m

```
 1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 2  %
 3  % ME40064 Coursework 2
 4  %
 5  % File         :  DoseEvaluator.m
 6  % Author       :  11973
 7  % Created      :  2025-11-26 (YYYY-MM-DD)
 8  % License      :  MIT
 9  % Description  :  A static class defining a evaluation methods
10  %                  for dose effectiveness
11  %
12  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14  classdef DoseEvaluator
15
16      methods (Static)
17
18          function K = EvaluateSolution(solution, target_x, c_threshold, dt)
19          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
20          %
21          % Function:     EvaluateSolution()
22          %
23          % Arguments:    solution, target x location, concentration
24          % Returns:      kappa value
25          %
26          % Description:  Evaluates the solution at a target location
27          %
28          %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29
30              % first, find the closest node to the target
31              node_index = 0;
32
33              for i = 1:solution.mesh.node_count
34                  x = solution.mesh.node_coords(i);
35                  if x >= target_x
36                      node_index = i;
37                      break;
38                  end
39              end
40
41              fprintf("node index %d\n", node_index);
```

```
            c = solution.values(node_index, :);

            effective_t_index = 0;

            for i = 1:length(c)
                if c(i) >= c_threshold
                    effective_t_index = i;
                    break
                end
            end

            if effective_t_index == 0
                K = 0; % never exceeds threshold
                return;
            end

            t_effective = solution.time(effective_t_index);

            % integrate concentration over time until effective_t_index
            time_range = effective_t_index:length(solution.time);
            K = trapz(solution.time(time_range), c(time_range));
        end

        function [c_dose_min, dose_vals, kappa_vals] = FindMinimumDose(mesh, tmax, dt,
    theta, integration_method, target_x, c_threshold, K_target)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     FindMinimumDose()
        %
        % Arguments:    mesh, max time, time step, theta value,
        % Returns:      minimum effective dose, results from search
        %
        % Description:  Performs a binary search to find the minimum
        %               effective dose to achieve a target kappa value
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % Binary search for minimum effective dose - high and low starting bounds
            c_dose_low = 0;
            c_dose_high = 100;

            tolerance = 0.1;

            dose_vals = [];
            kappa_vals = [];

            while (c_dose_high - c_dose_low) > tolerance
                c_dose_test = (c_dose_low + c_dose_high) / 2;

                fprintf("Testing dose = %.2f...\n", c_dose_test);

                % Run simulation with this dose
                lhs_boundary = BoundaryCondition();
                lhs_boundary.Type = BoundaryType.Dirichlet;
                lhs_boundary.Value = c_dose_test;

                rhs_boundary = BoundaryCondition();
                rhs_boundary.Type = BoundaryType.Dirichlet;
                rhs_boundary.Value = 0.0;

                solution = NumericSolver.SolveNumeric(...
                    mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
                    @(~, ~) 0.0, integration_method);

                % Evaluate K at target location
                K = DoseEvaluator.EvaluateSolution(solution, target_x, ...
                    c_threshold, dt);
```

```
108
109                 if K > K_target
110                     c_dose_high = c_dose_test;
111                     fprintf("K = %.2f > %.0f: dose too high\n", K, K_target);
112                 else
113                     c_dose_low = c_dose_test;
114                     fprintf("K = %.2f < %.0f: dose too low\n", K, K_target);
115                 end
116
117                 dose_vals = [dose_vals, c_dose_test];
118                 kappa_vals = [kappa_vals, K];
119             end
120
121             c_dose_min = c_dose_high;  % Use upper bound to ensure K > target
122
123         end
124     end
125 end
```

14. Solver

14.1. NumericSolver.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  NumericSolver.m
% Author       :  11973
% Created      :  2025-11-24 (YYYY-MM-DD)
% License      :  MIT
% Description  :  Class definition for generic solver for the
%                 transient diffusion-reaction equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


classdef NumericSolver

    methods (Static)

        function solution = SolveNumeric(mesh, tmax, dt, theta, left_boundary, right_boundary, source_fn, integration_method)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     SolveNumeric()
        %
        % Arguments:    parameters for solver
        % Returns:      solution object
        %
        % Description:  Runs a numeric solver for the transient
        %               diffusion-reaction equation using the
        %               finite element method and theta-method for
        %               time integration.
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % time vector
            time_vector = 0:dt:tmax;
            solution = Solution(mesh, time_vector);

            % initial condition: c(x,0) = 0
            c0 = zeros(mesh.node_count, 1);
            solution.SetValues(c0, 1);  % column 1 = t=0

            % create global matrices
            [K, M] = NumericSolver.CreateGlobalMatrices(mesh, theta, integration_method);

            % loop over time steps
            for step = 1:length(time_vector) - 1

                % solve for next time step
                c_next = NumericSolver.SolveStep(mesh, solution, step, dt, theta, ...
                K, M, left_boundary, right_boundary, source_fn, integration_method);

                solution.SetValues(c_next, step + 1);

            end

        end

        function c = SolveStep(mesh, solution, step, dt, theta, K, M, left_boundary, right_boundary, source_fn, integration_method)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     SolveStep()
```

```
        %
        % Arguments:    parameters for solver
        % Returns:      solution at next time step
        %
        % Description:  Solves for the solution at the next time step
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            c_current = solution.values(:, step);

            t = (step - 1) * dt; % current time, converted to 0-based index

            % assemble system matrix and rhs vector
            system_matrix = M + theta * dt * K;
            rhs_vector = (M - (1 - theta) * dt * K) * c_current;

            % add source term
            f_current = NumericSolver.CreateSourceVector(mesh, t, ...
                source_fn, integration_method);

            f_next = NumericSolver.CreateSourceVector(mesh, t + dt, ...
                source_fn, integration_method);

            rhs_vector = rhs_vector + dt * (theta * f_next + (1 - theta) * f_current);

            % apply boundary conditions
            [system_matrix, rhs_vector] = NumericSolver.ApplyBoundaryConditions(...
                system_matrix, rhs_vector, t + dt, left_boundary, right_boundary);

            % solve system
            c = system_matrix \ rhs_vector;

        end

        function [K, M] = CreateGlobalMatrices(mesh, theta, integration_method)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     CreateGlobalMatrices()
        %
        % Arguments:    parameters for solver
        % Returns:      global stiffness and mass matrices
        %
        % Description:  Creates the global stiffness and mass matrices
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            num_elements = mesh.element_count;
            num_nodes = mesh.node_count;

            % initialise global matrix (use sparse for efficiency with large systems)
            K = sparse(num_nodes, num_nodes);
            M = sparse(num_nodes, num_nodes);

            for element_id = 1:num_elements

                element = mesh.elements(element_id);
                nodes = element.node_ids;
                local_size = length(nodes);

                diff_matrix = ElementMatrices.DiffusionElemMatrix(element, ...
                    integration_method);
                react_matrix = ElementMatrices.ReactionElemMatrix(element, ...
                    integration_method);

                k_matrix = diff_matrix - react_matrix;

                elem_size = element.node_coords(end) - element.node_coords(1);
```

```
                m_matrix = ElementMatrices.MassElemMatrix(element, integration_method);

                % assemble into global matrices
                for i = 1:local_size
                    for j = 1:local_size
                        gi = nodes(i); gj = nodes(j);
                        K(gi, gj) = K(gi, gj) + k_matrix(i, j);
                        M(gi, gj) = M(gi, gj) + m_matrix(i, j);
                    end
                end

            end

        end

        function F = CreateSourceVector(mesh, t, source_fn, integration_method)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     CreateSourceVector()
        %
        % Arguments:    parameters for solver
        % Returns:      global source vector
        %
        % Description:  Creates the global source vector
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            F = zeros(mesh.node_count, 1);

            % return if no source function defined
            if (isempty(source_fn))
                return;
            end

            for element_id = 1:mesh.element_count

                element = mesh.elements(element_id);

                elem_size = element.node_coords(end) - element.node_coords(1);
                midpoint = (element.node_coords(1) + element.node_coords(end)) / 2;

                f_val = source_fn(midpoint, t);

                % Local Force Vector for linear element (Int N^T * s dx)
                f_local = f_val * ElementMatrices.ForceMatrix(element, integration_method);

                nodes = element.node_ids;
                F(nodes) = F(nodes) + f_local;
            end

        end

        function [lhs, rhs] = ApplyBoundaryConditions(lhs, rhs, t, left_boundary,
    right_boundary)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     ApplyBoundaryConditions()
        %
        % Arguments:    parameters for solver
        % Returns:      modified system matrix and rhs vector
        %
        % Description:  Applies boundary conditions to the system
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

             % Store diagonal values before modification
            diag_left = lhs(1,1);
```

```
            diag_right = lhs(end,end);

            % apply left boundary condition
            switch left_boundary.Type

                case BoundaryType.Dirichlet
                    lhs(1, :) = 0;                  % clear row
                    lhs(1, 1) = diag_left;          % keep diagonal
                    rhs(1) = left_boundary.Value * diag_left;   % scale by diagonal

                case BoundaryType.Neumann
                    rhs(1) = rhs(1) + left_boundary.ValueFunction(t); % apply flux

            end

            % apply right boundary condition
            switch right_boundary.Type
                 case BoundaryType.Dirichlet
                    lhs(end, :) = 0;                % clear row
                    lhs(end, end) = diag_right;     % set diagonal to 1
                    rhs(end) = right_boundary.Value * diag_right; % set value

                case BoundaryType.Neumann
                    rhs(end) = rhs(end) + right_boundary.ValueFunction(t); % apply flux
            end

        end

    end
end

```

14.2. BoundaryCondition.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  BoundaryCondition.m
% Author       :  11973
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a boundary condition for the
%                 transient diffusion reaction equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef BoundaryCondition

    properties

        Type            BoundaryType % Boundary condition type
        Value           double % Boundary condition value for Dirichlet
        ValueFunction   function_handle % Boundary condition function for Neumann

    end

end
```

14.3. BoundaryType.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
```

```
%
% File         :  BoundaryType.m
% Author       :  11973
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a boundary type enumeration for
%                 the transient diffusion reaction equation.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef BoundaryType

   enumeration

      Dirichlet, Neumann

   end

end
```

14.4. ElementMatrices.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  ElementMatrices.m
% Author       :  11973
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A static class defining element matrix
%                 helper functions for the transient diffusion
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef ElementMatrices

    methods (Static)

        function matrix = DiffusionElemMatrix(element, method)
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     DiffusionElemMatrix()
        %
        % Arguments:    element and integration method
        % Returns:      diffusion element matrix
        %
        % Description:  Computes the diffusion element matrix
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            elem_size = element.node_coords(end) - element.node_coords(1);

            % check integration method
            if method.type == IntegrationType.Trapezoidal

                % create base matrix
                matrix = eye(element.order + 1);

                for i = 1:(element.order + 1)
                    for j = 1:(element.order + 1)
                        if i ~= j
                            matrix(i, j) = -1;
                        end
                    end
```

```
44                 end
45
46                 % apply matrix scaling
47                 matrix = matrix * (element.D / elem_size);
48
49             else
50                 % handle Gaussian quadrature
51
52                 matrix = zeros(element.order + 1);
53                 [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
54
55                 for i = 1:length(xi)
56                     dN_dxi = ElementMatrices.ShapeFunctionDerivatives(element.order,
   xi(i));
57
58                     J = element.jacobian;
59                     dN_dx = dN_dxi / J;
60
61                     % compute contribution to stiffness matrix
62                     matrix = matrix + (element.D * (dN_dx' * dN_dx)) * (wi(i) * J);
63                 end
64
65             end
66
67         end
68
69         function matrix = ReactionElemMatrix(element, method)
70         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
71         %
72         % Function:     ReactionElemMatrix()
73         %
74         % Arguments:    element and integration method
75         % Returns:      reaction element matrix
76         %
77         % Description:  Computes the reaction element matrix
78         %
79         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
80
81             elem_size = element.node_coords(end) - element.node_coords(1);
82
83             % check integration method
84             if method.type == IntegrationType.Trapezoidal
85
86                 % create base matrix
87                 matrix = eye(element.order + 1) * 2;
88
89                 for i = 1:(element.order + 1)
90                     for j = 1:(element.order + 1)
91                         if i ~= j
92                             matrix(i, j) = 1;
93                         end
94                     end
95                 end
96
97                 % apply matrix scaling
98                 matrix = matrix * (element.lambda * elem_size / 6);
99
100            else
101                % handle Gaussian quadrature
102
103                matrix = zeros(element.order + 1);
104                [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
105
106                for i = 1:length(xi)
107                    N = ElementMatrices.ShapeFunctions(element.order, xi(i));
108
109                    J = element.jacobian;
```

```
110
111                 % compute contribution to stiffness matrix
112                 matrix = matrix + (element.lambda * (N' * N)) * (wi(i) * J);
113             end
114         end
115     end
116
117     function matrix = MassElemMatrix(element, method)
118     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
119     %
120     % Function:     MassElemMatrix()
121     %
122     % Arguments:    element and integration method
123     % Returns:      mass element matrix
124     %
125     % Description:  Computes the mass element matrix
126     %
127     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
128
129         elem_size = element.node_coords(end) - element.node_coords(1);
130
131         % check integration method
132         if method.type == IntegrationType.Trapezoidal
133
134             % create base matrix
135             matrix = eye(element.order + 1) * 2;
136
137             for i = 1:(element.order + 1)
138                 for j = 1:(element.order + 1)
139                     if i ~= j
140                         matrix(i, j) = 1;
141                     end
142                 end
143             end
144
145             % apply matrix scaling
146             matrix = matrix * (elem_size / 6);
147
148         else
149             % handle Gaussian quadrature
150
151             matrix = zeros(element.order + 1);
152             [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
153
154             for i = 1:length(xi)
155                 N = ElementMatrices.ShapeFunctions(element.order, xi(i));
156
157                 J = element.jacobian;
158
159                 % compute contribution to stiffness matrix
160                 matrix = matrix + (N' * N) * (wi(i) * J);
161             end
162
163         end
164
165     end
166
167
168     function matrix = ForceMatrix(element, method)
169     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
170     %
171     % Function:     ForceMatrix()
172     %
173     % Arguments:    element and integration method
174     % Returns:      force element matrix
175     %
176     % Description:  Computes the force element matrix
```

```
177         %
178         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
179
180             elem_size = element.node_coords(end) - element.node_coords(1);
181
182             % check integration method
183             if method.type == IntegrationType.Trapezoidal
184
185                 % create base matrix
186                 matrix = ones(element.order + 1, 1);
187
188                 % apply matrix scaling
189                 matrix = matrix * (elem_size / 2);
190
191             else
192                 % handle Gaussian quadrature
193
194                 matrix = zeros(element.order + 1, 1);
195                 [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
196
197                 for i = 1:length(xi)
198                     N = ElementMatrices.ShapeFunctions(element.order, xi(i));
199
200                     J = element.jacobian;
201
202                     % compute contribution to stiffness matrix
203                     matrix = matrix + N' * (wi(i) * J);
204                 end
205
206             end
207
208         end
209     end
210
211     methods (Static, Access = private)
212
213         function [xi, wi] = GaussQuadraturePoints(n)
214         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
215         %
216         % Function:     GaussQuadraturePoints()
217         %
218         % Arguments:    number of points
219         % Returns:      quadrature points and weights
220         %
221         % Description:  Looks up Gauss quadrature points and weights
222         %
223         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
224
225             % look up points and weights, limited to n = 1, 2, 3
226             switch n
227                 case 1
228                     xi = 0;
229                     wi = 2;
230                 case 2
231                     xi = [-1/sqrt(3), 1/sqrt(3)];
232                     wi = [1, 1];
233                 case 3
234                     xi = [-sqrt(3/5), 0, sqrt(3/5)];
235                     wi = [5/9, 8/9, 5/9];
236                 otherwise
237                     error('Gauss quadrature for n > 3 not implemented.');
238             end
239
240         end
241
242         function N = ShapeFunctions(order, xi)
243         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
244         %
245         % Function:     ShapeFunctions()
246         %
247         % Arguments:    order and local coordinate
248         % Returns:      shape function values
249         %
250         % Description:  Computes shape function values at given local
251         %               coordinate
252         %
253         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
254
255             % switch based on element order, limited to linear and quadratic
256             switch order
257                 case 1 % linear
258                     N = [(1 - xi) / 2, (1 + xi) / 2];
259                 case 2 % quadratic
260                     N = [xi * (xi - 1) / 2, (1 - xi^2), xi * (xi + 1) / 2];
261                 otherwise
262                     error('Shape functions for order > 2 not implemented.');
263             end
264
265         end
266
267         function dN_dxi = ShapeFunctionDerivatives(order, xi)
268         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
269         %
270         % Function:     ShapeFunctionDerivatives()
271         %
272         % Arguments:    order and local coordinate
273         % Returns:      shape function derivative values
274         %
275         % Description:  Computes shape function derivative values at
276         %               given local coordinate
277         %
278         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
279
280             % switch based on element order, limited to linear and quadratic
281             switch order
282                 case 1 % linear
283                     dN_dxi = [-0.5, 0.5];
284                 case 2 % quadratic
285                     dN_dxi = [xi - 0.5, -2 * xi, xi + 0.5];
286                 otherwise
287                     error('Shape function derivatives for order > 2 not implemented.');
288             end
289
290         end
291     end
292 end
```

14.5. IntegrationMethod.m

```
1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % ME40064 Coursework 2
4  %
5  % File         :  IntegrationMethod.m
6  % Author       :  11973
7  % Created      :  2025-11-26 (YYYY-MM-DD)
8  % License      :  MIT
9  % Description  :  A class defining an integration method for element
10 %                 matrix calculations
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
```

```
classdef IntegrationMethod

    properties

        type            IntegrationType
        gauss_points    uint8

    end

end
```

14.6. IntegrationType.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File          :  IntegrationType.m
% Author        :  11973
% Created       :  2025-11-26 (YYYY-MM-DD)
% License       :  MIT
% Description   :  A class defining an integration type for element
%                  matrix calculations
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef IntegrationType

   enumeration

        Trapezoidal, Gaussian
   end

end
```

15. Tests

15.1. UnitTests.m

```m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  UnitTests.m
% Author       :  11973
% Created      :  2025-11-27 (YYYY-MM-DD)
% License      :  MIT
% Description  :  Test suite for FEM transient diffusion solver
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function tests = UnitTests
    % run unit tests for transient diffusion solver
    tests = functiontests(localfunctions);
end

function TestSolveNumericReactionOnly(testCase)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Function:     TestSolveNumericReactionOnly()
%
% Arguments:    test case
% Returns:      none
%
% Description:  Tests NumericSolver for pure reaction case
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    % mesh parameters
    xmin = 0.0;
    xmax = 1.0;
    element_count = 6;
    order = 1;
    lambda = -1.0;
    D = 0.0;

    % time parameters
    tmax = 0.5;
    dt = 0.02;
    theta = 0.5; % Crank-Nicholson

    % generate mesh
    mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
    mesh.Generate();

    % solver parameters
    lhs_boundary = BoundaryCondition();
    lhs_boundary.Type = BoundaryType.Neumann;
    lhs_boundary.ValueFunction = @(t) 0.0;

    rhs_boundary = BoundaryCondition();
    rhs_boundary.Type = BoundaryType.Neumann;
    rhs_boundary.ValueFunction = @(t) 0.0;

    integration_method = IntegrationMethod();
    integration_method.type = IntegrationType.Trapezoidal;
    integration_method.gauss_points = 0; % not used for trapezoidal

    numeric_solution = NumericSolver.SolveNumeric(...
        mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 10, integration_method);

    % analytical solution
```

```
    t_analytic = 0:dt:tmax;
    c_exact = 10 * (1 - exp(lambda * t_analytic));

    % chose random node (3 in this case) to compare
    c_numeric = numeric_solution.values(3,:);
    error = norm(c_numeric - c_exact) / norm(c_exact);

    tolerance = 1e-3;
    verifyLessThan(testCase, error, tolerance);
end

function TestSteadyStateConvergence(testCase)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Function:     TestSteadyStateConvergence()
%
% Arguments:    test case
% Returns:      none
%
% Description:  Tests that NumericSolver converges to steady
%               state for pure diffusion with Dirichlet BCs
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    xmin = 0; xmax = 1;
    mesh = Mesh(xmin, xmax, 20, 1, 1.0, 0.0);
    mesh.Generate();

    % Run to steady state
    tmax = 10.0; dt = 0.1; theta = 0.5;

    lhs_bc = BoundaryCondition();
    lhs_bc.Type = BoundaryType.Dirichlet;
    lhs_bc.Value = 0.0;

    rhs_bc = BoundaryCondition();
    rhs_bc.Type = BoundaryType.Dirichlet;
    rhs_bc.Value = 1.0;

    integration_method = IntegrationMethod();
    integration_method.type = IntegrationType.Trapezoidal;

    solution = NumericSolver.SolveNumeric(mesh, tmax, dt, theta, ...
                                          lhs_bc, rhs_bc, @(x,t) 0, integration_method);

    % Check final solution is linear
    c_final = solution.values(:, end);
    c_expected = mesh.node_coords';

    error = norm(c_final - c_expected) / norm(c_expected);
    verifyLessThan(testCase, error, 1e-3);
end

function TestGaussianQuadratureAccuracy(testCase)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Function:     TestGaussianQuadratureAccuracy()
%
% Arguments:    test case
% Returns:      none
%
% Description:  Tests that Gaussian quadrature gives better
%               accuracy than trapezoidal for the same problem
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    xmin = 0; xmax = 1;
```

```
    element_count = 5;
    order = 2; % quadratic elements

    mesh = Mesh(xmin, xmax, element_count, order, 1.0, 0.0);
    mesh.Generate();

    tmax = 0.1; dt = 0.01; theta = 0.5;

    lhs_bc = BoundaryCondition();
    lhs_bc.Type = BoundaryType.Dirichlet;
    lhs_bc.Value = 0.0;

    rhs_bc = BoundaryCondition();
    rhs_bc.Type = BoundaryType.Dirichlet;
    rhs_bc.Value = 1.0;

    % Solve with trapezoidal
    trap_method = IntegrationMethod();
    trap_method.type = IntegrationType.Trapezoidal;

    trap_solution = NumericSolver.SolveNumeric(mesh, tmax, dt, theta, ...
        lhs_bc, rhs_bc, @(x,t) 0, trap_method);

    % Solve with Gaussian
    gauss_method = IntegrationMethod();
    gauss_method.type = IntegrationType.Gaussian;
    gauss_method.gauss_points = 3;

    gauss_solution = NumericSolver.SolveNumeric(mesh, tmax, dt, theta, ...
        lhs_bc, rhs_bc, @(x,t) 0, gauss_method);

    % Get analytical solution
    analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);

    % Compute errors
    trap_error = L2Error(analytical_solution, trap_solution);
    gauss_error = L2Error(analytical_solution, gauss_solution);

    % Gaussian should be more accurate (lower final error)
    verifyLessThan(testCase, gauss_error.l2_error(end), ...
        trap_error.l2_error(end));
end

function TestQuadraticVsLinearElements(testCase)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Function:     TestQuadraticVsLinearElements()
%
% Arguments:    test case
% Returns:      none
%
% Description:  Tests that quadratic elements give better
%               accuracy than linear elements for the same
%               problem
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    xmin = 0; xmax = 1;
    element_count = 5;

    tmax = 0.1; dt = 0.01; theta = 0.5;

    lhs_bc = BoundaryCondition();
    lhs_bc.Type = BoundaryType.Dirichlet;
    lhs_bc.Value = 0.0;

    rhs_bc = BoundaryCondition();
```

```
198     rhs_bc.Type = BoundaryType.Dirichlet;
199     rhs_bc.Value = 1.0;
200 
201     integration_method = IntegrationMethod();
202     integration_method.type = IntegrationType.Gaussian;
203     integration_method.gauss_points = 3;
204 
205     % Linear elements
206     mesh_linear = Mesh(xmin, xmax, element_count, 1, 1.0, 0.0);
207     mesh_linear.Generate();
208 
209     solution_linear = NumericSolver.SolveNumeric(mesh_linear, tmax, dt, ...
210         theta, lhs_bc, rhs_bc, @(x,t) 0, integration_method);
211 
212     analytical_linear = AnalyticalSolver.SolveAnalytical(mesh_linear, tmax, dt);
213     error_linear = L2Error(analytical_linear, solution_linear);
214 
215     % Quadratic elements
216     mesh_quadratic = Mesh(xmin, xmax, element_count, 2, 1.0, 0.0);
217     mesh_quadratic.Generate();
218 
219     solution_quadratic = NumericSolver.SolveNumeric(mesh_quadratic, tmax, dt, ...
220         theta, lhs_bc, rhs_bc, @(x,t) 0, integration_method);
221 
222     analytical_quadratic = AnalyticalSolver.SolveAnalytical(mesh_quadratic, tmax, dt);
223     error_quadratic = L2Error(analytical_quadratic, solution_quadratic);
224 
225     % Quadratic should have lower error
226     verifyLessThan(testCase, error_quadratic.l2_error(end), ...
227         error_linear.l2_error(end));
228 end
```