

ME40064 System Modelling and Simulation - Coursework 2

2846 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).

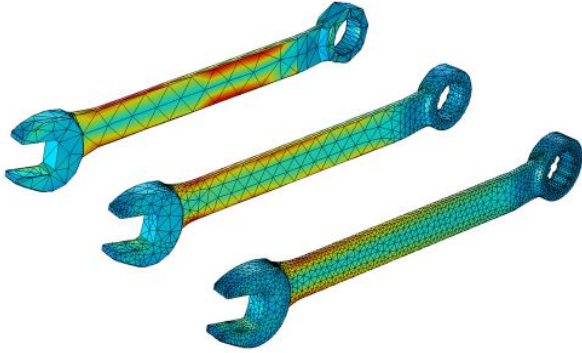


Figure 1: Finite Element Modelling of a Wrench under a Test Load Scenario [2]

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

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

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:

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

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).

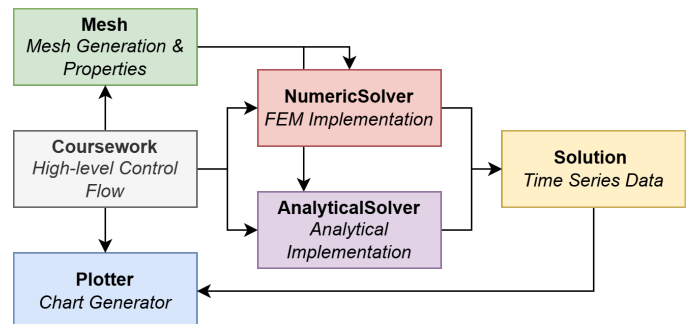


Figure 2: High-Level Software Architecture of the FEM Solver

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).

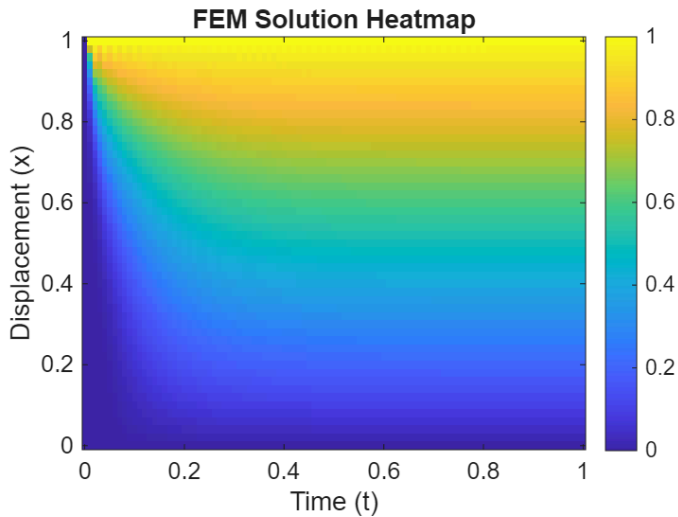


Figure 3: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

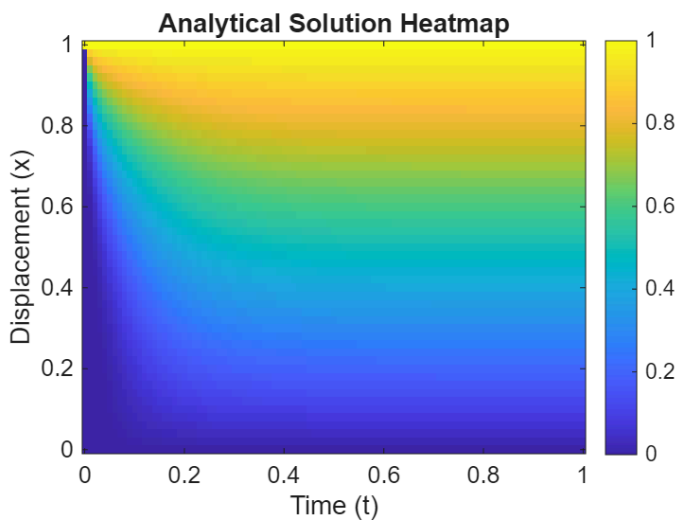


Figure 4: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

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.

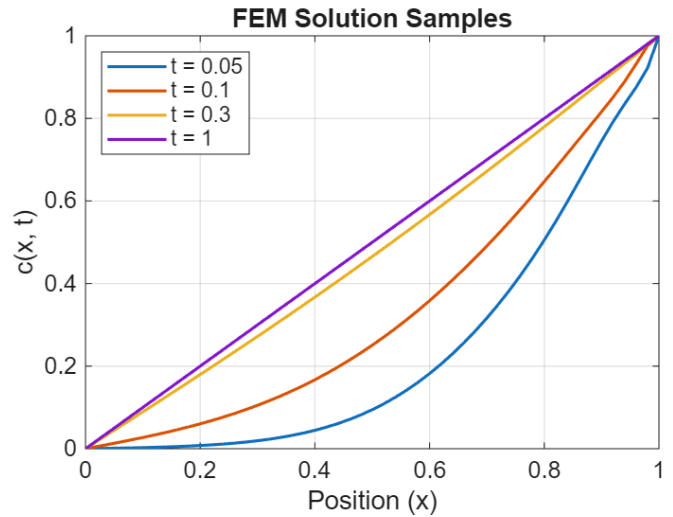


Figure 5: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

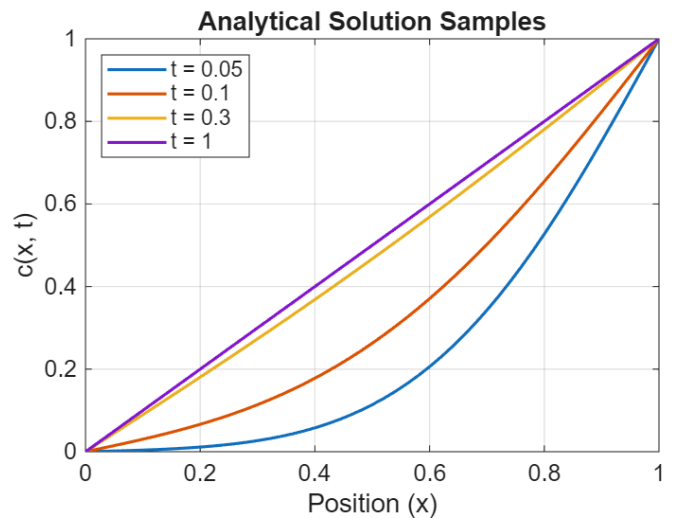


Figure 6: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

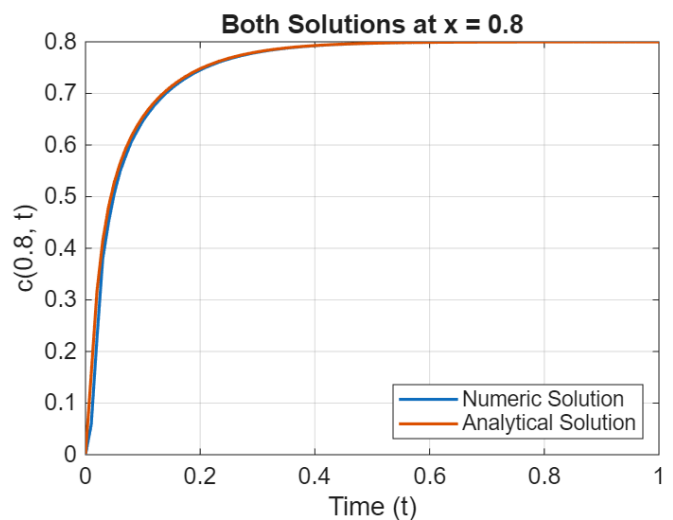


Figure 7: Comparison of Analytical and FEM Solutions at $x = 0.8$ over $0 \leq t \leq 1s$

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.

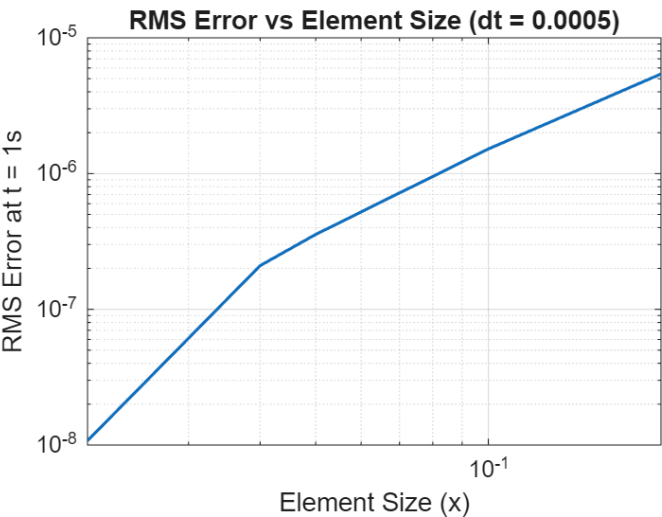


Figure 8: Comparison of RMS errors at $t = 1s$ for Varying Element Sizes

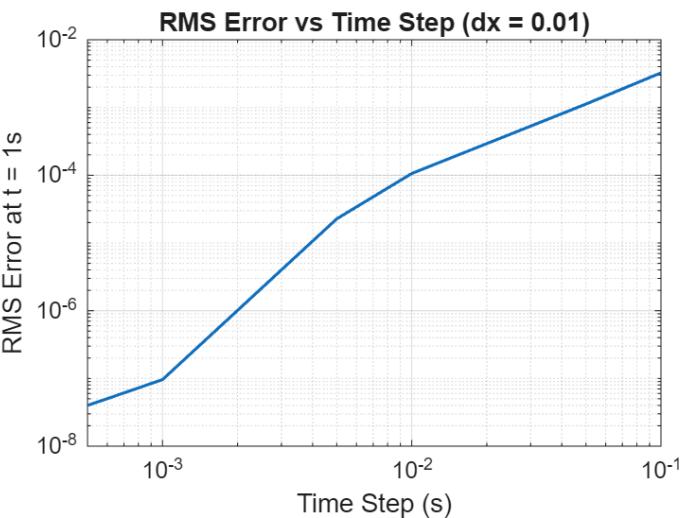


Figure 9: Comparison of RMS errors at $t = 1s$ for Varying Time Steps

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 evaluation 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.

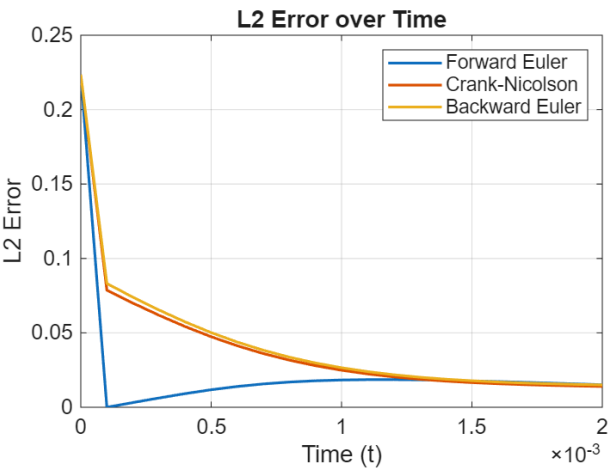


Figure 10: Comparison of L2 Errors for Different Time Integration Methods

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]:

$$dt \leq \frac{dx^2}{2D} \quad (3)$$

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:

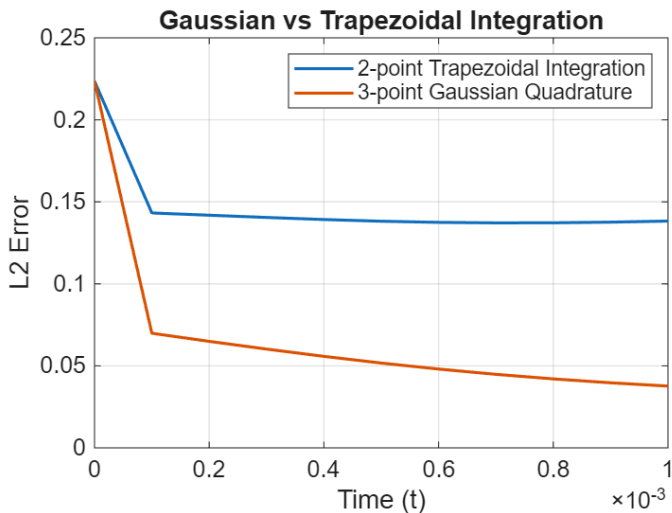


Figure 11: Comparison of L2 Errors for Gaussian Quadrature and Trapezoidal Integration

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, 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 intermediary points.

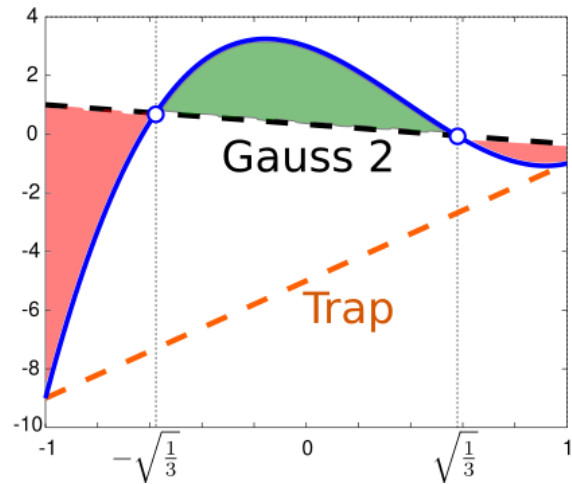


Figure 12: Visualisation of Gaussian Quadrature vs Trapezoidal Integration [8]

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:

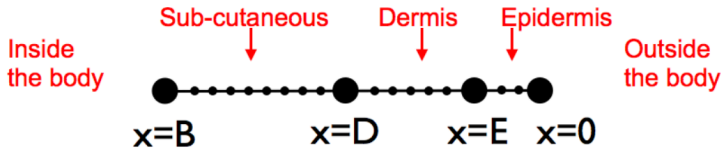


Figure 13: 1D Multilayer Finite Element Mesh of Skin Tissue Layers [9]

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

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

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.

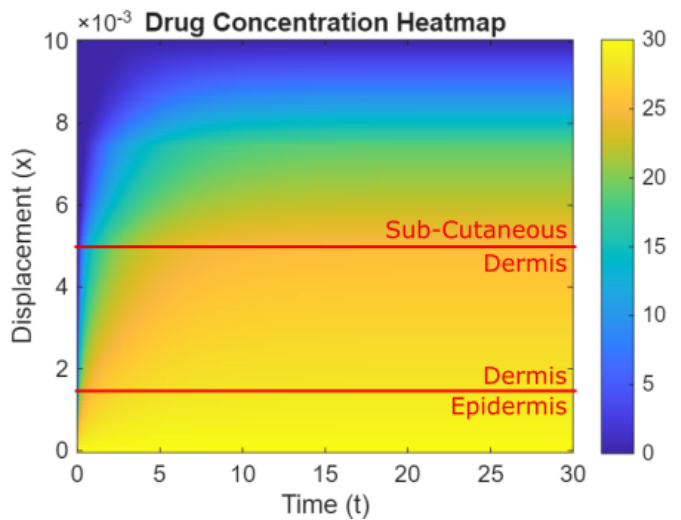


Figure 14: FEM Solution of Drug Diffusion through Multilayer Skin Structure over $0 \leq x \leq 0.01$ and $0 \leq t \leq 30s$

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

$$K = \int_{t_{\text{eff}}}^{t=30} c \, dt \quad (5)$$

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 identify 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 iteratively 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:

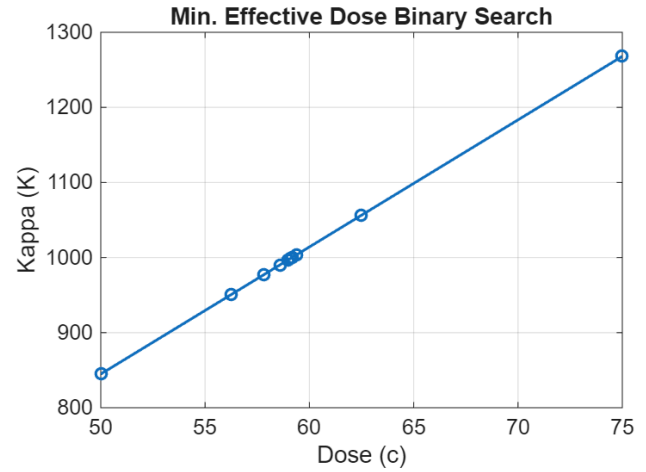


Figure 15: Binary Search for Minimum Effective Dose

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.

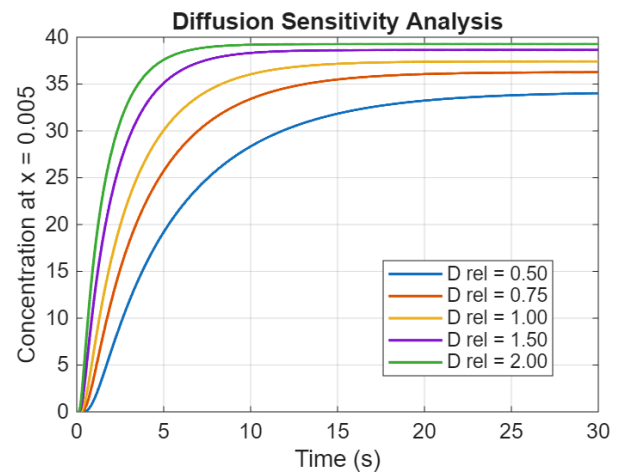
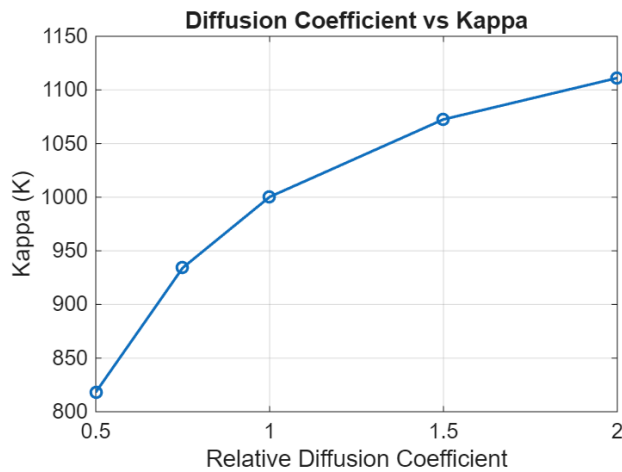
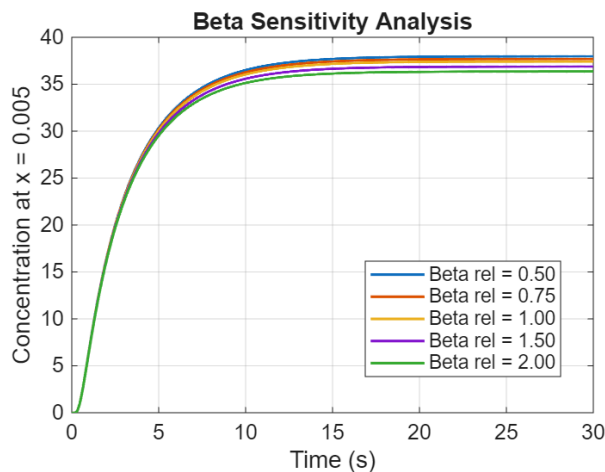
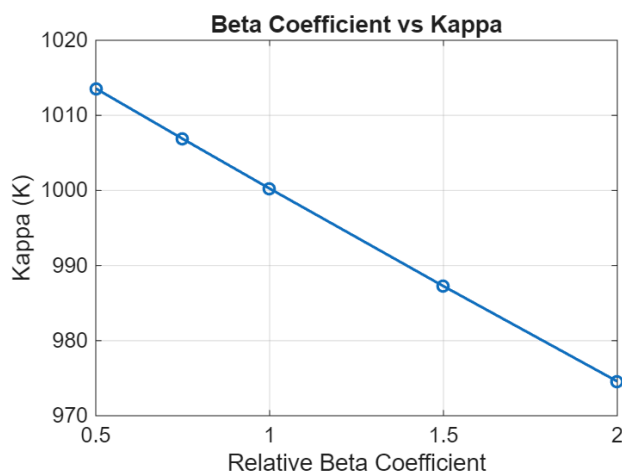


Figure 16: Concentration at $x = 0.005$ for Varying Values of D

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.

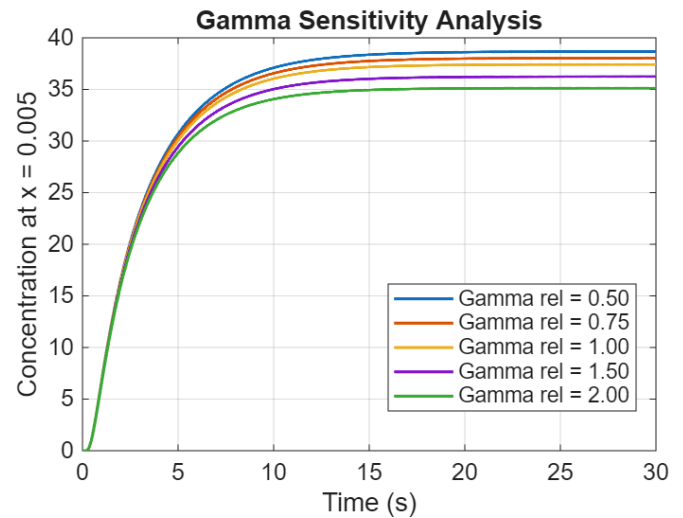
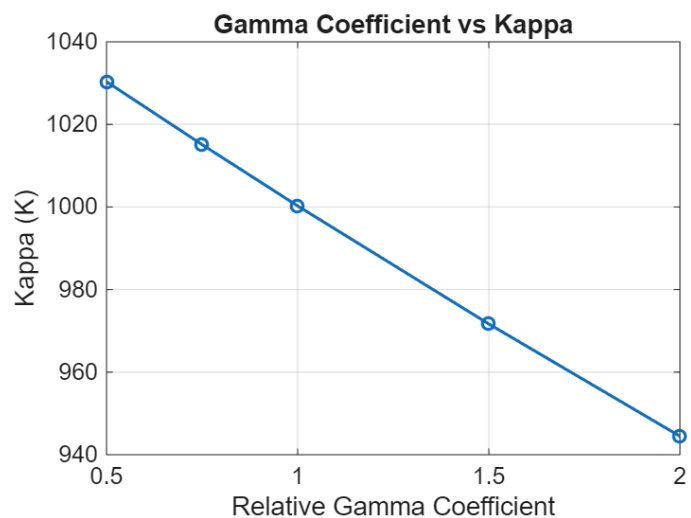
Figure 18: Concentration at $x = D$ for Varying Values of β Figure 19: Dose Effectiveness for Varying Values of β

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 degradation 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 its 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 accurately 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 multi-layer 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

```
1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : main.m
6 % Author    : 11973
7 % Created   : 2025-11-24 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : Main function for solving transient diffusion equation
10 %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12
13 function main()
14     fprintf("ME40064 Coursework 2 Starting...\n");
15
16     % Part 1: Software Verification
17     Coursework.Part1Plots();
18     Coursework.Part1Convergence();
19
20     % Part 2: Software Features
21     Coursework.Part2TimeIntegrationComparison();
22     Coursework.Part2GaussianQuadrature();
23
24     % Part 3: Modelling & Simulation Results
25     Coursework.Part3InitialResults();
26     Coursework.Part3MinimumEffectiveDose();
27     Coursework.Part3DoseSensitivityAnalysis();
28
29     fprintf("...ME40064 Coursework 2 Complete\n");
30 end
31
```

9. Coursework

9.1. Coursework.m

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % ME40064 Coursework 2
4  %
5  % File      : Coursework.m
6  % Author    : 11973
7  % Created   : 2025-11-27 (YYYY-MM-DD)
8  % License   : MIT
9  % Description : Static methods for each part of the coursework.
10 %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12
13 classdef Coursework
14
15     methods (Static)
16
17         function Part1Plots()
18             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
19             %
20             % Function:      Part1Plots()
21             %
22             % Arguments:     None
23             % Returns:       None
24             %
25             % Description:   Runs the start Part 1 of the coursework,
26                             % generating a simple mesh, running numeric and
27                             % analytical solvers, and plotting the results.
28             %
29             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
30
31             % time parameters
32             tmax = 1.0;
33             dt = 0.01;
34
35             % mesh parameters
36             xmin = 0.0;
37             xmax = 1.0;
38             element_count = 50;
39             order = 1;
40
41             % Crank-Nicholson method
42             theta = 0.5;
43
44             % diffusion and reaction coefficients
45             D = 1.0;
46             lambda = 0.0;
47
48             % concentrations
49             c_max = 1.0;
50             c_min = 0.0;
51
52
53             % generate mesh
54             mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
55             mesh.Generate();
56
57             % solver parameters
58             lhs_boundary = BoundaryCondition();
59             lhs_boundary.Type = BoundaryType.Dirichlet;
60             lhs_boundary.Value = c_min;
61
62             rhs_boundary = BoundaryCondition();
63             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 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     end
244
245     % plot element counts
246     dx = rms_errr_table_elem_count(:, 3); % element size
247     err_spatial = rms_errr_table_elem_count(:, 4);
248
249     Plotter.PlotConvergenceError(dx, err_spatial, ...
250         "RMS Error vs Element Size (dt = 0.0005)", ...
251         "cw2/report/resources/part1/ElementSizeConvergence", ...
252         "Element Size (x)", "RMS Error at t = 1s");
253
254     % plot time steps
255     dt_vals = rms_errr_table_time_step(:, 2); % time steps
256     err_temporal = rms_errr_table_time_step(:, 4);
257
258     Plotter.PlotConvergenceError(dt_vals, err_temporal, ...
259         "RMS Error vs Time Step (dx = 0.01)", ...
260         "cw2/report/resources/part1/TimeStepConvergence", ...
261         "Time Step (s)", "RMS Error at t = 1s");
262
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);

```

```

332         l2_errors = [l2_errors, l2_error];
333     end
334
335     Plotter.PlotL2Errors(l2_errors, "L2 Error over Time", ...
336         "cw2/report/resources/part2/L2ErrorTimeIntegration", ...
337         method_names);
338
339     % perform stability analysis
340
341     tmax = 1.0;
342     element_count = 50;
343     dt_list = [0.0001, 0.001, 0.01, 0.1, 0.25];
344
345     % generate mesh
346     mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
347     mesh.Generate();
348
349     l2_errors_stability = [];
350
351     for i = 1:length(thetas)
352         theta = thetas(i);
353
354         l2_errors_dt = [];
355
356         for j = 1:length(dt_list)
357             dt = dt_list(j);
358
359             try
360
361                 fprintf("Testing %s with dt = %.4f...\n", method_names{i}, dt);
362
363                 numeric_solution = NumericSolver.SolveNumeric(...
364                     mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
365                     @(~, ~) 0.0, integration_method);
366
367                 % CHECK FOR NaN/Inf at each timestep
368                 for t_idx = 1:length(numeric_solution.time)
369                     vals = numeric_solution.values(:, t_idx);
370                     if any(isnan(vals))
371                         fprintf("%s: NaN at step %d (t=%.4f)\n", ...
372                             method_names{i}, t_idx, numeric_solution.time(t_idx));
373                         break;
374                     end
375                     if any(isinf(vals))
376                         fprintf("%s: Inf at step %d (t=%.4f)\n", ...
377                             method_names{i}, t_idx, numeric_solution.time(t_idx));
378                         break;
379                     end
380
381                     if max(abs(vals)) > 1e10
382                         fprintf("%s: Explosion at step %d (t=%.4f), max=%.2e\n",
383                             method_names{i}, t_idx, numeric_solution.time(t_idx), max(abs(vals)));
384                         break;
385                     end
386
387                     analytical_solution = AnalyticalSolver.SolveAnalytical(...
388                         mesh, tmax, dt);
389
390                     l2_error = L2Error(analytical_solution, numeric_solution);
391
392                     l2_errors_dt = [l2_errors_dt, l2_error];
393                 catch
394                     l2_errors_dt = [l2_errors_dt, NaN];
395                     fprintf("%s EXPLODED \n", method_names{i});
396                 end
397             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     % time parameters
418     tmax = 0.001;
419     dt = 0.0001;
420
421     % mesh parameters
422     xmin = 0.0;
423     xmax = 1.0;
424     element_count = 5;
425     order = 2;
426
427     % diffusion and reaction coefficients
428     D = 0.5;
429     lambda = 0.0;
430
431     % concentrations
432     c_max = 1.0;
433     c_min = 0.0;
434
435     % generate mesh
436     mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
437     mesh.Generate();
438
439     % solve analytically
440     analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
441
442     % solver parameters
443
444     theta = 0.5; % Crank-Nicholson
445
446     lhs_boundary = BoundaryCondition();
447     lhs_boundary.Type = BoundaryType.Dirichlet;
448     lhs_boundary.Value = c_min;
449
450     rhs_boundary = BoundaryCondition();
451     rhs_boundary.Type = BoundaryType.Dirichlet;
452     rhs_boundary.Value = c_max;
453
454     % trapezoidal method
455     trapezoidal_method = IntegrationMethod();
456     trapezoidal_method.type = IntegrationType.Trapezoidal;
457     trapezoidal_method.gauss_points = 0; % not used for trapezoidal
458
459     trapezoidal_solution = NumericSolver.SolveNumeric(...
460         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0, ...
461         trapezoidal_method);
462
463     % gaussian quadrature method

```

```

465     gaussian_method = IntegrationMethod();
466     gaussian_method.type = IntegrationType.Gaussian;
467     gaussian_method.gauss_points = 3; % 3-point Gaussian quadrature
468
469     gaussian_solution = NumericSolver.SolveNumeric(...
470         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0, ...
471         gaussian_method);
472
473     % compute L2 error
474     l2_error_trapezoidal = L2Error(analytical_solution, trapezoidal_solution);
475     l2_error_gaussian = L2Error(analytical_solution, gaussian_solution);
476     l2_errors = [l2_error_trapezoidal, l2_error_gaussian];
477
478     method_names = {"2-point Trapezoidal Integration", ...
479         "3-point Gaussian Quadrature"};
480
481     Plotter.PlotL2Errors(l2_errors, "Gaussian vs Trapezoidal Integration", ...
482         "cw2/report/resources/part2/L2ErrorGaussianTrapezoidal", ...
483         method_names);
484 end
485
486 function Part3InitialResults()
487 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
488 %
489 % Function:      Part3InitialResults()
490 %
491 % Arguments:     None
492 % Returns:       None
493 %
494 % Description:   Runs a basic case for Part 3 of the coursework
495 %
496 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
497
498     % Generate mesh
499     xmin = 0;
500     xmax = 0.01;
501     element_count = 50;
502     order = 2;
503
504     theta = 0.5; % Crank-Nicholson
505     D = 1;
506     lambda = 0;
507
508     epidermis_layer = MeshLayer(0.0, 4e-6, 0.0, 0.02, 2.0);
509     dermis_layer = MeshLayer(0.00166667, 5e-6, 0.01, 0.02, 1.0);
510     sub_cutaneous_layer = MeshLayer(0.005, 2e-6, 0.01, 0.02, 1.0);
511
512     layers = [epidermis_layer, dermis_layer, sub_cutaneous_layer];
513
514     mesh = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers);
515     mesh.Generate();
516
517     tmax = 30.0;
518     dt = 0.01; % works well with element_count = 50
519
520     % concentrations
521     c_max = 30.0;
522     c_min = 0.0;
523
524     lhs_boundary = BoundaryCondition();
525     lhs_boundary.Type = BoundaryType.Dirichlet;
526     lhs_boundary.Value = c_max;
527
528     rhs_boundary = BoundaryCondition();
529     rhs_boundary.Type = BoundaryType.Dirichlet;
530     rhs_boundary.Value = c_min;
531

```

```

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     Plotter.PlotDoseEffectiveness(dose_vals, kappa_vals, ...
599         "Min. Effective Dose Binary Search", ...
600         "cw2/report/resources/part3/MinDoseBinarySearch", ...
601         "Dose (c)", "Kappa (K)");
602
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     epidermis_layer = MeshLayer(0.0, 4e-6, 0.0 * beta_rel, 0.02, 2.0);
733     dermis_layer = MeshLayer(0.00166667, 5e-6, 0.01 * beta_rel, 0.02, 1.0);
734     sub_cutaneous_layer = MeshLayer(0.005, 2e-6, 0.01 * beta_rel, 0.02, 1.0);
735
736     layers = [epidermis_layer, dermis_layer, sub_cutaneous_layer];
737
738     mesh = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers);
739     mesh.Generate();
740
741     numeric_solution = NumericSolver.SolveNumeric(...
742         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
743         @(~, ~) 0.0, integration_method);
744
745     kappa_values = [kappa_values, ...
746         DoseEvaluator.EvaluateSolution(numeric_solution, 0.005, 4.0, 0.0)];
747
748     % first, find the closest node to the target
749     node_index = 0;
750
751     for i = 1:numeric_solution.mesh.node_count
752         x = numeric_solution.mesh.node_coords(i);
753         if x >= target_x
754             node_index = i;
755             break;
756         end
757     end
758
759     if d == 1
760         x_values = numeric_solution.time;
761     end
762
763     y_values(d, :) = numeric_solution.values(node_index, :);
764 end
765
766 legend_strings = [];
767
768 for d = 1:length(relative_betas)
769     legend_strings = [legend_strings, ...
770         sprintf("Beta rel = %.2f", relative_betas(d))];
771 end
772
773 fprintf("Plotting Dose Sensitivity Analysis...\n");
774
775 Plotter.PlotSensitivityAnalysis(x_values, y_values, ...
776     "Beta Sensitivity Analysis", ...
777     "cw2/report/resources/part3/BetaSensitivityAnalysis", ...
778     "Time (s)", "Concentration at x = 0.005", legend_strings);
779
780 Plotter.PlotKappaValues(relative_betas, kappa_values, ...
781     "Beta Coefficient vs Kappa", ...
782     "cw2/report/resources/part3/BetaKappa", ...
783     "Relative Beta Coefficient", "Kappa (K)");
784
785 % gamma coefficients analysis
786 relative_gammas = [0.5, 0.75, 1.0, 1.5, 2.0];
787
788 target_x = 0.005; % use x = D as an example
789
790 x_values = [];
791 y_values = zeros(length(relative_gammas), tmax/dt + 1);
792 kappa_values = [];
793
794 for d = 1:length(relative_gammas)
795     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

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File           : Mesh.m
6 % Author        : 11973
7 % Created       : 2025-11-26 (YYYY-MM-DD)
8 % License       : MIT
9 % Description    : A class defining a one-dimensional mesh for
10 %                 finite element analysis.
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14 classdef Mesh < handle
15     % inherit from handle to allow pass-by-reference
16
17     properties
18
19         xmin           double % min x coordinate
20         xmax           double % max x coordinate
21         dx             double % element size
22
23         order          double % element order (1=linear, 2=quadratic etc)
24
25         D              double % diffusion coefficient
26         lambda         double % reaction coefficient
27
28         node_count     uint64 % total number of global nodes
29         node_coords    double % coordinates of global nodes
30
31         element_count  uint64
32         elements       MeshElement % array of mesh elements
33
34     end
35
36     methods
37
38         function obj = Mesh(xmin, xmax, element_count, order, D, lambda)
39             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
40             %
41             % Function:      Mesh()
42             %
43             % Arguments:    parameters to initialise
44             % Returns:      Mesh handle
45             %
46             % Description:  Initialises a one-dimensional mesh object
47             %
48             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
49
50             % assign properties
51             obj.xmin = xmin;
52             obj.xmax = xmax;
53             obj.dx = (xmax - xmin) / element_count;
54             obj.D = D;
55             obj.lambda = lambda;
56
57             obj.order = order;
58
59             % total number of nodes
60             obj.node_count = (element_count * order) + 1;
61             obj.node_coords = zeros(1, obj.node_count);
62
63             obj.element_count = element_count;

```



```

64         obj.elements = MeshElement.empty(element_count, 0);
65
66     end
67
68     function obj = Generate(obj)
69         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
70         %
71         % Function:      Generate()
72         %
73         % Arguments:    Mesh handle
74         % Returns:      Mesh handle
75         %
76         % Description:  Generates the mesh for the given object
77         %
78         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
79
80         disp('Generating normal mesh...');
81
82         % generate uniform node coordinates
83         obj.node_coords = linspace(obj.xmin, obj.xmax, obj.node_count);
84
85         % generate elements
86         for e = 1:obj.element_count
87
88             % determine global node IDs for this element
89             node_start = (e - 1) * obj.order + 1;
90             node_ids = node_start:(node_start + obj.order);
91
92             % coordinates for this element
93             coords = obj.node_coords(node_ids);
94
95             % create MeshElement object
96             obj.elements(e) = MeshElement(...
97                 node_ids, coords, obj.order, obj.D, obj.lambda);
98         end
99     end
100
101 end
102 end
103

```

10.2. MeshElement.m

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % ME40064 Coursework 2
4  %
5  % File           : MeshElement.m
6  % Author        : 11973
7  % Created       : 2025-11-26 (YYYY-MM-DD)
8  % License       : MIT
9  % Description    : A class defining a one-dimensional mesh element
10 %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12
13 classdef MeshElement
14
15     properties
16
17         order      uint8    % polynomial order (1 = linear, 2 = quadratic)
18         node_ids    uint64   % global node IDs
19         node_coords double   % node coordinates
20         jacobian    double   % element jacobian d(x)/d(xi)
21         D           double   % diffusion coefficient
22         lambda      double   % reaction coefficient

```

```

23     end
24
25     methods
26
27         function obj = MeshElement(ids, coords, order, D, lambda)
28             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29             %
30             % Function:      MeshElement()
31             %
32             % Arguments:    parameters to initialise
33             % Returns:      MeshElement handle
34             %
35             % Description:  Initialises a one-dimensional mesh element object
36             %
37             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
38
39             % assign properties
40             obj.node_ids = ids;
41             obj.node_coords = coords;
42             obj.order = order;
43             obj.D = D;
44             obj.lambda = lambda;
45
46             % linear mapping from [-1, 1] to [x1, x2]
47             % jacobian = dx/dxi = (x2 - x1) / 2
48             obj.jacobian = (coords(end) - coords(1)) / 2;
49
50         end
51     end
52 end
53
54

```

10.3. MultilayerMesh.m

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % ME40064 Coursework 2
4  %
5  % File      : MultilayerMesh.m
6  % Author    : 11973
7  % Created   : 2025-11-26 (YYYY-MM-DD)
8  % License   : MIT
9  % Description : A class defining a multilayer one-dimensional mesh for
10 %               finite element analysis.
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14
15 classdef MultilayerMesh < Mesh
16     % inherit from Mesh
17
18     properties
19         layers          MeshLayer % array of layer properties
20         total_density    double % total density ratio across all layers
21     end
22
23     methods
24
25         function obj = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers)
26             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
27             %
28             % Function:      MultilayerMesh()
29             %
30             % 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)

```

```

98         layer_xmax = obj.layers(l + 1).x;
99     else
100         layer_xmax = obj.xmax;
101     end
102
103     layer_nodes = obj.layers(l).element_count * obj.order + 1;
104
105     layer_coords = linspace(layer_xmin, layer_xmax, layer_nodes);
106
107
108     if l == 1
109         nodes_to_add = layer_coords;
110     else
111         nodes_to_add = layer_coords(2:end); % Skip duplicate boundary node
112     end
113
114     % Add nodes to global coordinate array
115     num_new_nodes = length(nodes_to_add);
116     obj.node_coords(current_node : current_node + num_new_nodes - 1)...
117         = nodes_to_add;
118
119     current_node = current_node + num_new_nodes;
120
121
122 end
123
124 % Generate elements
125 for e = 1:obj.element_count
126
127     % Determine global node IDs for this element
128     node_start = (e - 1) * obj.order + 1;
129     node_ids = node_start:(node_start + obj.order);
130
131     % Coordinates for this element
132     coords = obj.node_coords(node_ids);
133
134     midpoint = (coords(1) + coords(end)) / 2;
135
136     % Determine which layer this element is in
137     layer_index = 1;
138     for l = 1:length(obj.layers)
139         if midpoint >= obj.layers(l).x
140             layer_index = l;
141         end
142     end
143
144     D = obj.layers(layer_index).D;
145     lambda = -(obj.layers(layer_index).beta + obj.layers(layer_index).gamma);
146
147     % Create MeshElement object
148     obj.elements(e) = MeshElement(node_ids, coords, obj.order, D, lambda);
149 end
150 end
151 end
152 end
153 end
154
155

```

10.4. MeshLayer.m

[illegible]

```

5 % File      : MeshLayer.m
6 % Author    : samh25
7 % Created   : 2025-11-29 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A class defining a layer in a multi-layer mesh
10 %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12
13 classdef MeshLayer
14     properties
15         x            double % min x coordinate for this layer
16         density_ratio double % density ratio for this layer
17         D            double % diffusion coefficient
18         beta         double % extra-vascular diffusivity
19         gamma        double % drug degradation rate
20
21         element_count uint64 % number of elements in this layer
22         layer_offset  uint64 % starting element index for this layer
23     end
24
25     methods
26
27         function obj = MeshLayer(x, D, beta, gamma, density_ratio)
28             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29             %
30             % Function:      MeshLayer()
31             %
32             % Arguments:    parameters to initialise
33             % Returns:      MeshLayer handle
34             %
35             % Description:  Initialises a mesh layer object
36             %
37             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
38
39             obj.x = x;
40             obj.D = D;
41             obj.beta = beta;
42             obj.gamma = gamma;
43             obj.density_ratio = density_ratio;
44
45             obj.element_count = 0;
46         end
47     end
48 end
49 end

```


11. Analytical

11.1. AnalyticalSolver.m

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : AnalyticalSolver.m
6 % Author    : 11973
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A static class defining an analytical solver
10 %              for the transient diffusion equation
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14 classdef AnalyticalSolver
15
16     methods (Static)
17
18         function solution = SolveAnalytical(mesh, tmax, dt)
19             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
20             %
21             % Function:      SolveAnalytical()
22             %
23             % Arguments:    mesh, max time and time step
24             % Returns:      solution to the equation
25             %
26             % Description:  Solves the transient diffusion equation using
27                             the analytical solution.
28             %
29             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
30
31             % time vector
32             time_vector = 0:dt:tmax;
33             solution = Solution(mesh, time_vector);
34
35             % loop over time steps
36             for step = 1:length(time_vector)
37
38                 t = time_vector(step);
39                 timestep_results = zeros(1, mesh.node_count);
40
41                 % loop over nodes
42                 for i = 1:mesh.node_count
43                     x = mesh.node_coords(i);
44                     timestep_results(i) = TransientAnalyticSoln(x, t);
45                 end
46
47                 solution.SetValues(timestep_results, step);
48             end
49         end
50     end
51 end
52 end

```

11.2. TransientAnalyticSoln.m

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : TransientAnalyticSoln.m
6 % Author    : A. N. Cookson
7 % Created   : 2025-11-11 (YYYY-MM-DD)

```

```
8 % License      : -
9 % Description  : Analytical solution to transient diffusion equation
10 %              provided in the coursework materials.
11 %
12 %%%%%%%%%%%%%%
13
14 function [ c ] = TransientAnalyticSoln(x,t)
15 %TransientAnalyticSoln Analytical solution to transient diffusion equation
16 %   Computes the analytical solution to the transient diffusion equation for
17 %   the domain x=[0,1], subject to initial condition: c(x,0) = 0, and Dirichlet
18 %   boundary conditions: c(0,t) = 0, and c(1,t) = 1.
19 %   Input Arguments:
20 %   x is the point in space to evaluate the solution at
21 %   t is the point in time to evaluate the solution at
22 %   Output Argument:
23 %   c is the value of concentration at point x and time t, i.e. c(x,t)
24
25 trans = 0.0;
26
27 for k=1:1000
28     trans = trans + (((-1)^k)/k) * exp(-k^2*pi^2*t)*sin(k*pi*x));
29 end
30
31 c = x + (2/pi)*trans;
32
33 end
```

[illegible]

```

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

```

[illegible]

[illegible]

```

261 %
262 % Function:      PlotDoseEffectiveness()
263 %
264 % Arguments:    Plotting parameters
265 % Returns:      None
266 %
267 % Description:   Plots dose vs effectiveness
268 %
269 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
270
271     set(0, "DefaultAxesFontSize", 12);
272     set(0, "DefaultTextFontSize", 12);
273
274     figure;
275
276     plot_handle = plot(dose_values, kappa_values, "-o", 'LineWidth', 1.5);
277     xlabel(x_label);
278     ylabel(y_label);
279     title(title_str);
280     grid on;
281
282     set(gcf, 'Position', [0, 0, 500, 350]);
283
284     % Save figure
285     saveas(gcf, name, "png");
286     saveas(gcf, name, "fig");
287     openfig(name + ".fig");
288 end
289
290 function PlotSensitivityAnalysis(x_values, y_values, title_str, name, x_label,
y_label, legend_strings)
291 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
292 %
293 % Function:      PlotSensitivityAnalysis()
294 %
295 % Arguments:    Plotting parameters
296 % Returns:      None
297 %
298 % Description:   Plots sensitivity analysis results
299 %
300 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
301
302     set(0, "DefaultAxesFontSize", 12);
303     set(0, "DefaultTextFontSize", 12);
304
305     figure;
306
307     % y_values is a matrix where each row is a different series
308     for i = 1:size(y_values, 1)
309         plot_handle = plot(x_values, y_values(i, :));
310         set(plot_handle, "LineWidth", 1.5);
311         hold on;
312     end
313
314     xlabel(x_label);
315     ylabel(y_label);
316     title(title_str);
317     legend(legend_strings, 'Location', 'best');
318     grid on;
319
320     set(gcf, 'Position', [0, 0, 500, 350]);
321
322     % Save figure
323     saveas(gcf, name, "png");
324     saveas(gcf, name, "fig");
325     openfig(name + ".fig");
326 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

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Solution.m
6 % Author    : 11973
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A class defining a solution to the transient
10 %              diffusion equation
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14 classdef Solution < handle
15     % inherit from handle to allow pass-by-reference behaviour
16
17     properties
18
19         mesh      Mesh      % handle to mesh object
20         time       double    % time series - 1 x Nsteps
21         values     double    % solution values - Nnodes x Nsteps
22
23     end
24
25     methods
26
27         function obj = Solution(mesh, time_vector)
28             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29             %
30             % Function:      Solution()
31             %
32             % Arguments:    parameters to initialise
33             % Returns:      Solution handle
34             %
35             % Description:  Initialises a Solution object
36             %
37             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
38
39             % assign properties
40             obj.mesh = mesh;
41             obj.time = time_vector;
42             obj.values = zeros(mesh.node_count, length(time_vector));
43
44         end
45
46
47         function obj = SetValues(obj, values, step)
48             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
49             %
50             % Function:      SetValues()
51             %
52             % Arguments:    object, values to set, time step index
53             % Returns:      Solution handle
54             %
55             % Description:  Helper function to set solution values at a
56             %              given time step
57             %
58             %
59             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
60
61             % set solution values at given time step
62             obj.values(:, step) = values(:);
63         end

```

```

64     end
65 end
66
67

```

13.2. L2Error.m

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % ME40064 Coursework 2
4  %
5  % File      : L2Error.m
6  % Author    : 11973
7  % Created   : 2025-11-26 (YYYY-MM-DD)
8  % License   : MIT
9  % Description : A class defining an L2 error calculator between
10 %               a numerical and reference solution
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14 classdef L2Error < handle
15     % inherit from handle to allow pass-by-reference behaviour
16
17     properties
18
19         ref_solution    Solution % handle to reference solution object
20         num_solution    Solution % handle to solution object
21
22         time            double    % time series - 1 x Nsteps
23         l2_error        double    % L2 error at each time step - 1 x Nsteps
24
25     end
26
27     methods
28
29
30         function obj = L2Error(ref_solution, num_solution)
31             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
32             %
33             % Function:      L2Error()
34             %
35             % Arguments:    rference and numerical solutions
36             % Returns:      L2Error handle
37             %
38             % Description:  Initialises an L2Error object and computes the
39             %               L2 error between the two solutions
40             %
41             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
42
43             % assign properties
44             obj.ref_solution = ref_solution;
45             obj.num_solution = num_solution;
46             obj.time = ref_solution.time;
47
48             % check compatibility
49             if ref_solution.mesh.node_count ~= num_solution.mesh.node_count
50                 error('Reference and error solutions must have the same number of nodes');
51             end
52
53             if length(ref_solution.time) ~= length(num_solution.time)
54                 error('Reference and error solutions must have the same number of time
55 steps');
56             end
57
58             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);

```

```

42     c = solution.values(node_index, :);
43
44     effective_t_index = 0;
45
46     for i = 1:length(c)
47         if c(i) >= c_threshold
48             effective_t_index = i;
49             break
50         end
51     end
52
53     if effective_t_index == 0
54         K = 0; % never exceeds threshold
55         return;
56     end
57
58     t_effective = solution.time(effective_t_index);
59
60     % integrate concentration over time until effective_t_index
61     time_range = effective_t_index:length(solution.time);
62     K = trapz(solution.time(time_range), c(time_range));
63 end
64
65 function [c_dose_min, dose_vals, kappa_vals] = FindMinimumDose(mesh, tmax, dt,
theta, integration_method, target_x, c_threshold, K_target)
66 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
67 %
68 % Function:      FindMinimumDose()
69 %
70 % Arguments:    mesh, max time, time step, theta value,
71 % Returns:      minimum effective dose, results from search
72 %
73 % Description:  Performs a binary search to find the minimum
74 %               effective dose to achieve a target kappa value
75 %
76 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
77
78     % Binary search for minimum effective dose - high and low starting bounds
79     c_dose_low = 0;
80     c_dose_high = 100;
81
82     tolerance = 0.1;
83
84     dose_vals = [];
85     kappa_vals = [];
86
87     while (c_dose_high - c_dose_low) > tolerance
88         c_dose_test = (c_dose_low + c_dose_high) / 2;
89
90         fprintf("Testing dose = %.2f...\n", c_dose_test);
91
92         % Run simulation with this dose
93         lhs_boundary = BoundaryCondition();
94         lhs_boundary.Type = BoundaryType.Dirichlet;
95         lhs_boundary.Value = c_dose_test;
96
97         rhs_boundary = BoundaryCondition();
98         rhs_boundary.Type = BoundaryType.Dirichlet;
99         rhs_boundary.Value = 0.0;
100
101         solution = NumericSolver.SolveNumeric(...
102             mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, ...
103             @(~, ~) 0.0, integration_method);
104
105         % Evaluate K at target location
106         K = DoseEvaluator.EvaluateSolution(solution, target_x, ...
107             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

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : NumericSolver.m
6 % Author    : 11973
7 % Created   : 2025-11-24 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : Class definition for generic solver for the
10 %              transient diffusion-reaction equation
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14
15 classdef NumericSolver
16
17     methods (Static)
18
19         function solution = SolveNumeric(mesh, tmax, dt, theta, left_boundary,
20 right_boundary, source_fn, integration_method)
21             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
22             %
23             % Function:      SolveNumeric()
24             %
25             % Arguments:    parameters for solver
26             % Returns:      solution object
27             %
28             % Description:  Runs a numeric solver for the transient
29             %              diffusion-reaction equation using the
30             %              finite element method and theta-method for
31             %              time integration.
32             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
33
34             % time vector
35             time_vector = 0:dt:tmax;
36             solution = Solution(mesh, time_vector);
37
38             % initial condition:  $c(x,0) = 0$ 
39             c0 = zeros(mesh.node_count, 1);
40             solution.SetValues(c0, 1); % column 1 =  $t=0$ 
41
42             % create global matrices
43             [K, M] = NumericSolver.CreateGlobalMatrices(mesh, theta, integration_method);
44
45             % loop over time steps
46             for step = 1:length(time_vector) - 1
47
48                 % solve for next time step
49                 c_next = NumericSolver.SolveStep(mesh, solution, step, dt, theta, ...
50 K, M, left_boundary, right_boundary, source_fn, integration_method);
51
52                 solution.SetValues(c_next, step + 1);
53
54             end
55
56         end
57
58         function c = SolveStep(mesh, solution, step, dt, theta, K, M, left_boundary,
59 right_boundary, source_fn, integration_method)
60             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
61             %
62             % Function:      SolveStep()

```



```

62      %
63      % Arguments:      parameters for solver
64      % Returns:       solution at next time step
65      %
66      % Description:   Solves for the solution at the next time step
67      %
68      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
69
70      c_current = solution.values(:, step);
71
72      t = (step - 1) * dt; % current time, converted to 0-based index
73
74      % assemble system matrix and rhs vector
75      system_matrix = M + theta * dt * K;
76      rhs_vector = (M - (1 - theta) * dt * K) * c_current;
77
78      % add source term
79      f_current = NumericSolver.CreateSourceVector(mesh, t, ...
80          source_fn, integration_method);
81
82      f_next = NumericSolver.CreateSourceVector(mesh, t + dt, ...
83          source_fn, integration_method);
84
85      rhs_vector = rhs_vector + dt * (theta * f_next + (1 - theta) * f_current);
86
87      % apply boundary conditions
88      [system_matrix, rhs_vector] = NumericSolver.ApplyBoundaryConditions(...
89          system_matrix, rhs_vector, t + dt, left_boundary, right_boundary);
90
91      % solve system
92      c = system_matrix \ rhs_vector;
93
94  end
95
96  function [K, M] = CreateGlobalMatrices(mesh, theta, integration_method)
97  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
98  %
99  % Function:      CreateGlobalMatrices()
100  %
101  % Arguments:     parameters for solver
102  % Returns:       global stiffness and mass matrices
103  %
104  % Description:   Creates the global stiffness and mass matrices
105  %
106  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
107
108      num_elements = mesh.element_count;
109      num_nodes = mesh.node_count;
110
111      % initialise global matrix (use sparse for efficiency with large systems)
112      K = sparse(num_nodes, num_nodes);
113      M = sparse(num_nodes, num_nodes);
114
115      for element_id = 1:num_elements
116
117          element = mesh.elements(element_id);
118          nodes = element.node_ids;
119          local_size = length(nodes);
120
121          diff_matrix = ElementMatrices.DiffusionElemMatrix(element, ...
122              integration_method);
123          react_matrix = ElementMatrices.ReactionElemMatrix(element, ...
124              integration_method);
125
126          k_matrix = diff_matrix - react_matrix;
127
128          elem_size = element.node_coords(end) - element.node_coords(1);

```

```

129         m_matrix = ElementMatrices.MassElemMatrix(element, integration_method);
130
131         % assemble into global matrices
132         for i = 1:local_size
133             for j = 1:local_size
134                 gi = nodes(i); gj = nodes(j);
135                 K(gi, gj) = K(gi, gj) + k_matrix(i, j);
136                 M(gi, gj) = M(gi, gj) + m_matrix(i, j);
137             end
138         end
139
140     end
141
142 end
143
144 function F = CreateSourceVector(mesh, t, source_fn, integration_method)
145 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
146 %
147 % Function:      CreateSourceVector()
148 %
149 % Arguments:     parameters for solver
150 % Returns:       global source vector
151 %
152 % Description:   Creates the global source vector
153 %
154 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
155
156     F = zeros(mesh.node_count, 1);
157
158     % return if no source function defined
159     if (isempty(source_fn))
160         return;
161     end
162
163     for element_id = 1:mesh.element_count
164
165         element = mesh.elements(element_id);
166
167         elem_size = element.node_coords(end) - element.node_coords(1);
168         midpoint = (element.node_coords(1) + element.node_coords(end)) / 2;
169
170         f_val = source_fn(midpoint, t);
171
172         % Local Force Vector for linear element (Int N^T * s dx)
173         f_local = f_val * ElementMatrices.ForceMatrix(element, integration_method);
174
175         nodes = element.node_ids;
176         F(nodes) = F(nodes) + f_local;
177     end
178
179 end
180
181 function [lhs, rhs] = ApplyBoundaryConditions(lhs, rhs, t, left_boundary,
right_boundary)
182 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
183 %
184 % Function:      ApplyBoundaryConditions()
185 %
186 % Arguments:     parameters for solver
187 % Returns:       modified system matrix and rhs vector
188 %
189 % Description:   Applies boundary conditions to the system
190 %
191 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
192
193     % Store diagonal values before modification
194     diag_left = lhs(1,1);

```

14.2. BoundaryCondition.m

14.3. BoundaryType.m

[illegible]

```

4 %
5 % File      : BoundaryType.m
6 % Author    : 11973
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A class defining a boundary type enumeration for
10 %              the transient diffusion reaction equation.
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14 classdef BoundaryType
15
16     enumeration
17
18         Dirichlet, Neumann
19
20     end
21
22 end

```

14.4. ElementMatrices.m

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : ElementMatrices.m
6 % Author    : 11973
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A static class defining element matrix
10 %              helper functions for the transient diffusion
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14 classdef ElementMatrices
15
16     methods (Static)
17
18         function matrix = DiffusionElemMatrix(element, method)
19             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
20             %
21             % Function:      DiffusionElemMatrix()
22             %
23             % Arguments:    element and integration method
24             % Returns:      diffusion element matrix
25             %
26             % Description:  Computes the diffusion element matrix
27             %
28             %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29
30             elem_size = element.node_coords(end) - element.node_coords(1);
31
32             % check integration method
33             if method.type == IntegrationType.Trapezoidal
34
35                 % create base matrix
36                 matrix = eye(element.order + 1);
37
38                 for i = 1:(element.order + 1)
39                     for j = 1:(element.order + 1)
40                         if i ~= j
41                             matrix(i, j) = -1;
42                         end
43                     end
44                 end
45             end
46         end
47     end
48
49 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     end
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     end
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     end
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.');
```



```

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

[illegible]

```

14 classdef IntegrationMethod
15
16     properties
17
18         type          IntegrationType
19         gauss_points  uint8
20
21     end
22
23 end

```

14.6. IntegrationType.m

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : IntegrationType.m
6 % Author    : 11973
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A class defining an integration type for element
10 %              matrix calculations
11 %
12 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
13
14 classdef IntegrationType
15
16     enumeration
17
18         Trapezoidal, Gaussian
19     end
20
21 end

```

15. Tests

15.1. UnitTests.m

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : UnitTests.m
6 % Author    : 11973
7 % Created   : 2025-11-27 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : Test suite for FEM transient diffusion solver
10 %
11 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
12
13 function tests = UnitTests
14     % run unit tests for transient diffusion solver
15     tests = functiontests(localfunctions);
16 end
17
18 function TestSolveNumericReactionOnly(testCase)
19 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
20 %
21 % Function:    TestSolveNumericReactionOnly()
22 %
23 % Arguments:   test case
24 % Returns:     none
25 %
26 % Description: Tests NumericSolver for pure reaction case
27 %
28 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29
30     % mesh parameters
31     xmin = 0.0;
32     xmax = 1.0;
33     element_count = 6;
34     order = 1;
35     lambda = -1.0;
36     D = 0.0;
37
38     % time parameters
39     tmax = 0.5;
40     dt = 0.02;
41     theta = 0.5; % Crank-Nicholson
42
43     % generate mesh
44     mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
45     mesh.Generate();
46
47     % solver parameters
48     lhs_boundary = BoundaryCondition();
49     lhs_boundary.Type = BoundaryType.Neumann;
50     lhs_boundary.ValueFunction = @(t) 0.0;
51
52     rhs_boundary = BoundaryCondition();
53     rhs_boundary.Type = BoundaryType.Neumann;
54     rhs_boundary.ValueFunction = @(t) 0.0;
55
56     integration_method = IntegrationMethod();
57     integration_method.type = IntegrationType.Trapezoidal;
58     integration_method.gauss_points = 0; % not used for trapezoidal
59
60     numeric_solution = NumericSolver.SolveNumeric(...
61         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 10, integration_method);
62
63     % analytical solution

```

[illegible]

```

131     element_count = 5;
132     order = 2; % quadratic elements
133
134     mesh = Mesh(xmin, xmax, element_count, order, 1.0, 0.0);
135     mesh.Generate();
136
137     tmax = 0.1; dt = 0.01; theta = 0.5;
138
139     lhs_bc = BoundaryCondition();
140     lhs_bc.Type = BoundaryType.Dirichlet;
141     lhs_bc.Value = 0.0;
142
143     rhs_bc = BoundaryCondition();
144     rhs_bc.Type = BoundaryType.Dirichlet;
145     rhs_bc.Value = 1.0;
146
147     % Solve with trapezoidal
148     trap_method = IntegrationMethod();
149     trap_method.type = IntegrationType.Trapezoidal;
150
151     trap_solution = NumericSolver.SolveNumeric(mesh, tmax, dt, theta, ...
152         lhs_bc, rhs_bc, @(x,t) 0, trap_method);
153
154     % Solve with Gaussian
155     gauss_method = IntegrationMethod();
156     gauss_method.type = IntegrationType.Gaussian;
157     gauss_method.gauss_points = 3;
158
159     gauss_solution = NumericSolver.SolveNumeric(mesh, tmax, dt, theta, ...
160         lhs_bc, rhs_bc, @(x,t) 0, gauss_method);
161
162     % Get analytical solution
163     analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
164
165     % Compute errors
166     trap_error = L2Error(analytical_solution, trap_solution);
167     gauss_error = L2Error(analytical_solution, gauss_solution);
168
169     % Gaussian should be more accurate (lower final error)
170     verifyLessThan(testCase, gauss_error.l2_error(end), ...
171         trap_error.l2_error(end));
172 end
173
174 function TestQuadraticVsLinearElements(testCase)
175 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
176 %
177 % Function:      TestQuadraticVsLinearElements()
178 %
179 % Arguments:     test case
180 % Returns:       none
181 %
182 % Description:   Tests that quadratic elements give better
183 %                accuracy than linear elements for the same
184 %                problem
185 %
186 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
187
188     xmin = 0; xmax = 1;
189     element_count = 5;
190
191     tmax = 0.1; dt = 0.01; theta = 0.5;
192
193     lhs_bc = BoundaryCondition();
194     lhs_bc.Type = BoundaryType.Dirichlet;
195     lhs_bc.Value = 0.0;
196
197     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
```