

ME40064 System Modelling and Simulation - Coursework 2

1704 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).

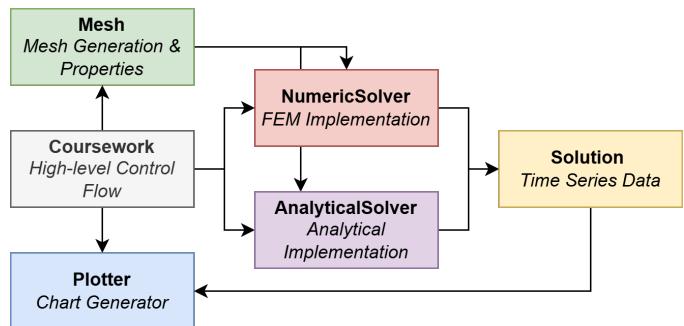


Figure 2: High-Level Software Architecture of the FEM Solver

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).

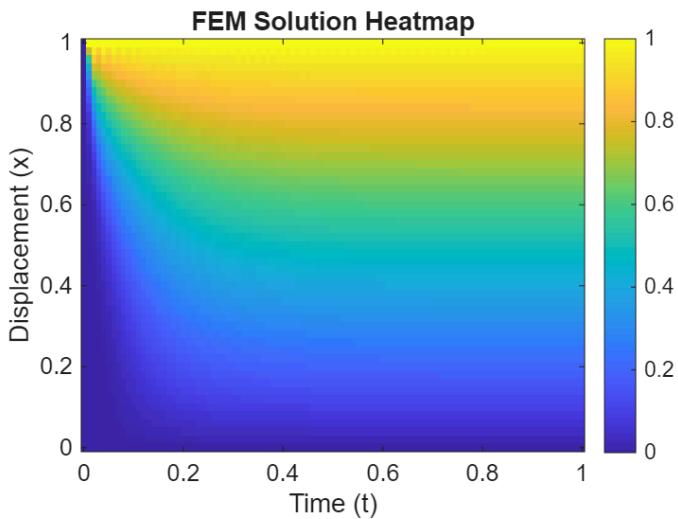


Figure 3: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

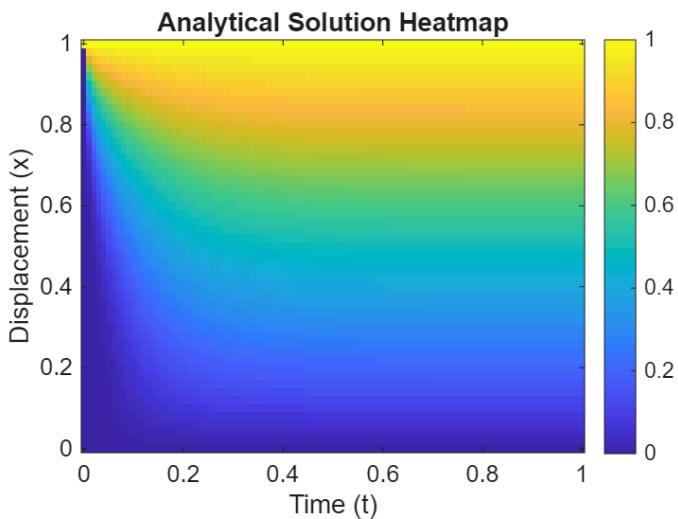


Figure 4: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

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.

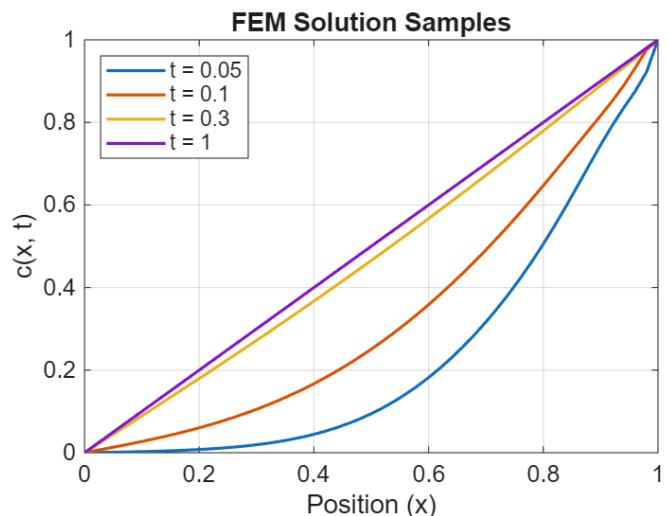


Figure 5: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

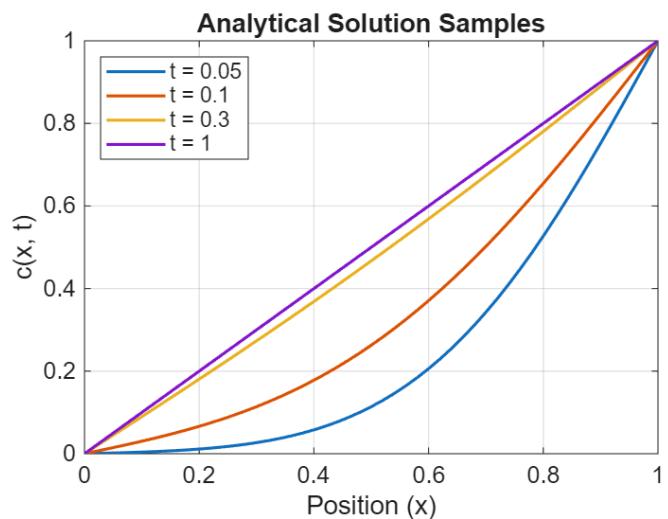


Figure 6: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

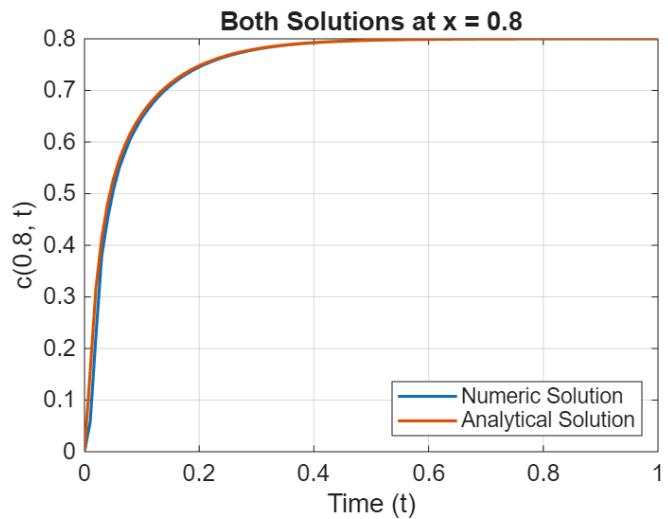


Figure 7: Comparison of Analytical and FEM Solutions at $x = 0.8$ over $0 \leq t \leq 1s$

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.

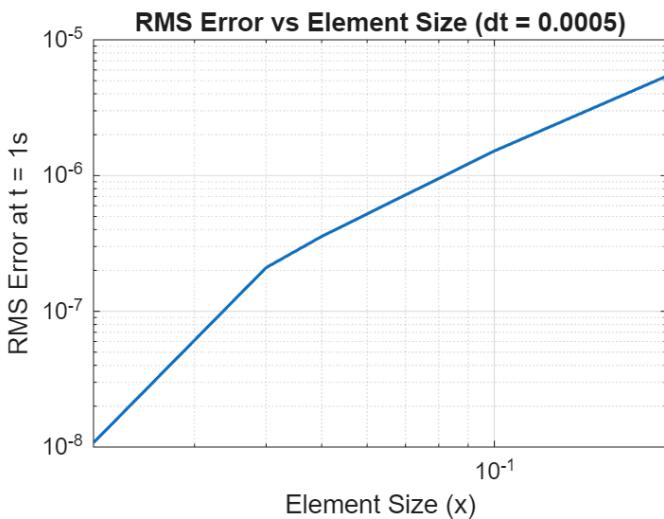


Figure 8: Comparison of RMS errors at $t = 1s$ for Varying Element Sizes

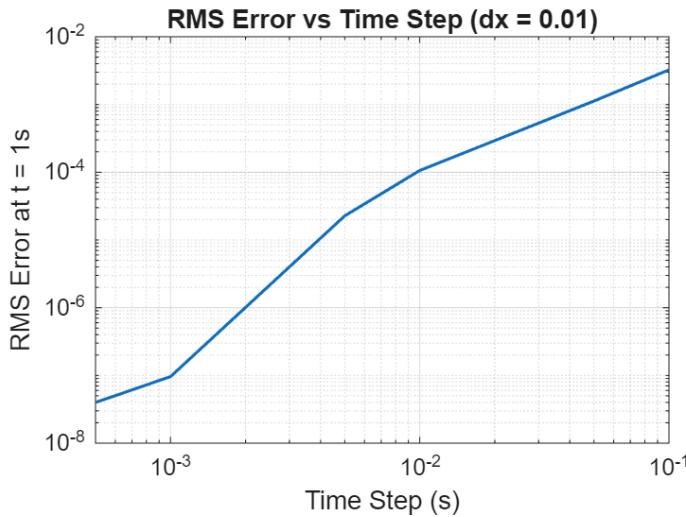


Figure 9: Comparison of RMS errors at $t = 1s$ for Varying Time Steps

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. This test was run using a mesh with 10 elements and a time step size of 0.0001s.

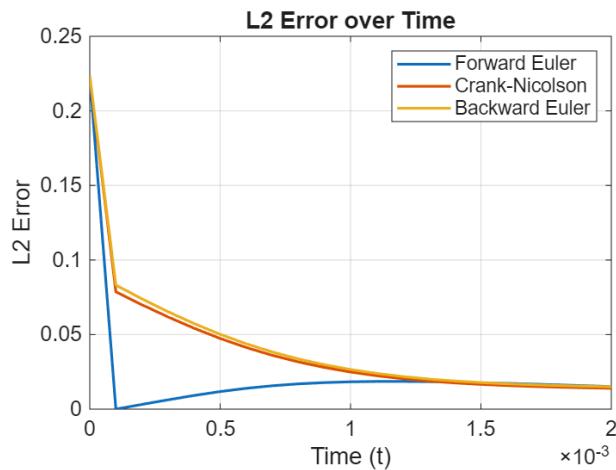


Figure 10: Comparison of L2 Errors for Different Time Integration Methods

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:

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:

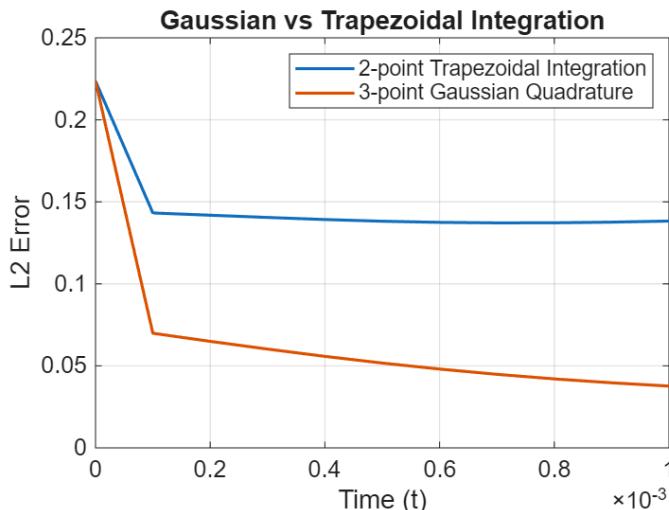


Figure 11: Comparison of L2 Errors for Gaussian Quadrature and Trapezoidal Integration

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.

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.

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:

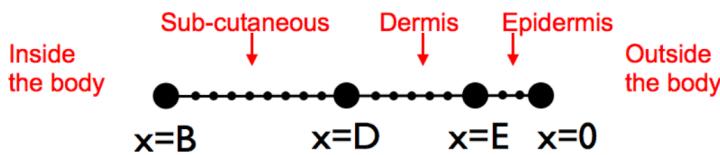


Figure 12: 1D Multilayer Finite Element Mesh of Skin Tissue Layers [8]

The concentration of the drug is modelled by the following 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 13), 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.

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.

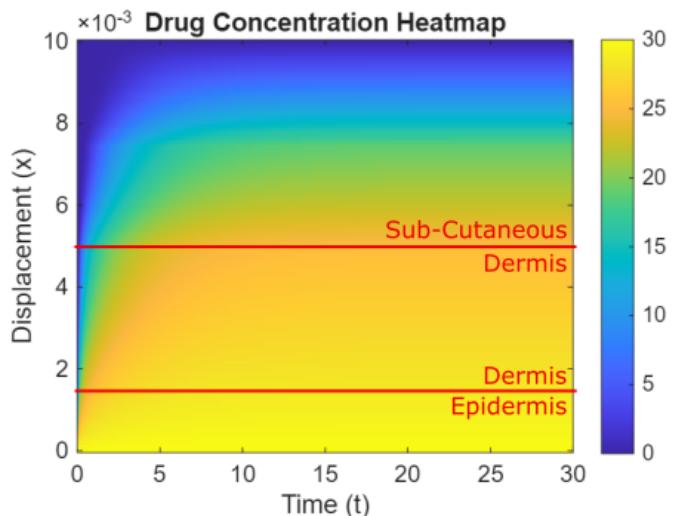


Figure 13: FEM Solution of Drug Diffusion through Multilayer Skin Structure over $0 \leq x \leq 0.01$ and $0 \leq t \leq 30s$

4.4. Dose Evaluation

4.5. Dose Sensitivity Analysis

4.6. Further Work

5. Conclusion

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] A. Cookson, “Assignment Transient MATLAB-Based FEM Modelling,” 2025.

7. Use of Generative AI

This coursework was completed in Visual Studio Code (with the [MATLAB Extension](#)), using Typst for report writing. The [GitHub Copilot](#) AI tool was enabled, providing generative suggestions for report phrasing and code snippets.

8. Appendix - MATLAB Source Code

9. Main

9.1. main.m

```
1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : main.m
6 % Author    : samh25
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     Coursework.Part3InitialResults();
17
18     fprintf("...ME40064 Coursework 2 Complete\n");
19 end
20
```

10. Coursework

10.1. Coursework.m

```
1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Coursework.m
6 % Author    : samh25
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, @(~, ~) 0.0,
73             integration_method);
74
75         % solve analytically
76         analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
77
78         % plot solutions as a heatmaps
79         Plotter.PlotHeatMap(numeric_solution, "FEM Solution Heatmap", ...
80             "cw2/report/resources/part1/NumericHeatmap", c_max);
81         Plotter.PlotHeatMap(analytical_solution, "Analytical Solution Heatmap", ...
82             "cw2/report/resources/part1/AnalyticalHeatmap", c_max);
83
84         % plot solution samples at specified times
85         sample_times = [0.05, 0.1, 0.3, 1.0];
86         Plotter.PlotTimeSamples(numeric_solution, dt, sample_times, "FEM Solution
87             Samples", ...
88                 "cw2/report/resources/part1/NumericSamples");
89         Plotter.PlotTimeSamples(analytical_solution, dt, sample_times, "Analytical
90             Solution Samples", ...
91                 "cw2/report/resources/part1/AnalyticalSamples");
92
93         % plot both solutions at a specific position over time
94         sample_x = 0.8;
95         legend_strings = {"Numeric Solution", "Analytical Solution"};
96         Plotter.PlotSampleOverTime(numeric_solution, analytical_solution, ...
97             sample_x, "Both Solutions at x = 0.8", "cw2/report/resources/part1/
98             BothX08", legend_strings);
99
100        end
101
102        function Part1Convergence()
103            % Function:      Part1Convergence()
104            %
105            % Arguments:    None
106            % Returns:      None
107            %
108            % Description:  Runs a convergence study for Part 1 of the
109            %               coursework, calculating RMS error between numeric
110            %               and analytical solutions over a range of element
111            %               counts and time steps.
112            %
113            % time parameters
114            tmax = 1.0;
115
116            % mesh parameters
117            xmin = 0.0;
118            xmax = 1.0;
119            element_count = 50;
120            order = 1;
121
122            % Crank-Nicholson method
123            theta = 0.5;
124
125            % diffusion and reaction coefficients
126            D = 1.0;
```

```

127     lambda = 0.0;
128
129     % concentrations
130     c_max = 1.0;
131     c_min = 0.0;
132
133
134     % generate mesh
135     mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
136     mesh.Generate();
137
138     % solver parameters
139     lhs_boundary = BoundaryCondition();
140     lhs_boundary.Type = BoundaryType.Dirichlet;
141     lhs_boundary.Value = c_min;
142
143     rhs_boundary = BoundaryCondition();
144     rhs_boundary.Type = BoundaryType.Dirichlet;
145     rhs_boundary.Value = c_max;
146
147     integration_method = IntegrationMethod();
148     integration_method.type = IntegrationType.Trapezoidal;
149     integration_method.gauss_points = 0; % not used for trapezoidal
150
151     % calculate RMS error with varying mesh sizes and time steps
152
153     element_counts = [5, 10, 20, 25, 50];
154     time_steps = [0.1, 0.05, 0.01, 0.005, 0.001, 0.0005];
155
156     num_cases = length(element_counts) * length(time_steps);
157     rms_errr_table_elem_count = zeros(num_cases, 4); % columns: elem_count, dt,
158     dx, RMS error
159     rms_errr_table_time_step = zeros(num_cases, 4); % columns: elem_count, dt,
160     dx, RMS error
161
162     k = 1;
163
164     % vary element count with fixed time step
165     for i = 1:length(element_counts)
166         elem_count = element_counts(i);
167         dt = 0.0005;
168
169         % generate mesh
170         mesh = Mesh(xmin, xmax, elem_count, order, D, lambda);
171         mesh.Generate();
172
173         % solve numerically
174         numeric_solution = NumericSolver.SolveNumeric(...
175             mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
176             integration_method);
177
178         % solve analytically
179         analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
180
181         % compute RMS error
182         [~, final_time] = min(abs(analytical_solution.time - tmax));
183
184         c_numeric = numeric_solution.values(:, final_time);
185         c_analytical = analytical_solution.values(:, final_time);
186
187         error = c_numeric - c_analytical;
188         rms_error = sqrt(mean(error.^2));
189
190         rms_errr_table_elem_count(k, :) = [elem_count, dt, (xmax-xmin)/elem_count,
191         rms_error];
192         k = k + 1;

```

```

190         fprintf("Elements: %d, dt: %.4f, dx: %.4f, RMS Error: %.6f\n", ...
191                 elem_count, dt, (xmax-xmin)/elem_count, rms_error);
192     end
193
194     k = 1;
195
196     % vary time step with fixed element count
197     for j = 1:length(time_steps)
198
199         elem_count = 1 / 0.01;
200         dt = time_steps(j);
201
202         % generate mesh
203         mesh = Mesh(xmin, xmax, elem_count, order, D, lambda);
204         mesh.Generate();
205
206         % solve numerically
207         numeric_solution = NumericSolver.SolveNumeric(...
208             mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
209             integration_method);
210
211         % solve analytically
212         analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
213
214         % compute RMS error
215         [~, final_time] = min(abs(analytical_solution.time - tmax));
216
217         c_numeric = numeric_solution.values(:, final_time);
218         c_analytical = analytical_solution.values(:, final_time);
219
220         error = c_numeric - c_analytical;
221         rms_error = sqrt(mean(error.^2));
222
223         rms_errr_table_time_step(k, :) = [elem_count, dt, (xmax-xmin)/elem_count,
224                                         rms_error];
225         k = k + 1;
226
227         fprintf("Elements: %d, dt: %.4f, dx: %.4f, RMS Error: %.6f\n", ...
228                 elem_count, dt, (xmax-xmin)/elem_count, rms_error);
229     end
230
231     % plot element counts
232     dx = rms_errr_table_elem_count(:, 3); % element size
233     err_spatial = rms_errr_table_elem_count(:, 4);
234
235     Plotter.PlotConvergenceError(dx, err_spatial, ...
236         "RMS Error vs Element Size (dt = 0.0005)", ...
237         "cw2/report/resources/part1/ElementSizeConvergence", "Element Size (x)",
238         "RMS Error at t = 1s");
239
240     % plot time steps
241     dt_vals = rms_errr_table_time_step(:, 2); % time steps
242     err_temporal = rms_errr_table_time_step(:, 4);
243
244     Plotter.PlotConvergenceError(dt_vals, err_temporal, ...
245         "RMS Error vs Time Step (dx = 0.01)", ...
246         "cw2/report/resources/part1/TimeStepConvergence", "Time Step (s)", "RMS
247     Error at t = 1s");
248
249     function Part2TimeIntegrationComparison()
250         %
251         % Function:      Part2TimeIntegrationComparison()
252         %
253         % Arguments:    None

```

```
253      % Returns:      None
254      %
255      % Description:  Runs a study comparing different time integration
256      %                  methods for Part 2 of the coursework.
257      %
258      %%%%%%%%%%%%%%
259
260      % time parameters
261      tmax = 0.002;
262      dt = 0.0001;
263
264      % mesh parameters
265      xmin = 0.0;
266      xmax = 1.0;
267      element_count = 10;
268      order = 1;
269
270      % diffusion and reaction coefficients
271      D = 1.0;
272      lambda = 0.0;
273
274      % concentrations
275      c_max = 1.0;
276      c_min = 0.0;
277
278      % generate mesh
279      mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
280      mesh.Generate();
281
282      % solve analytically
283      analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
284
285      % solver parameters
286      lhs_boundary = BoundaryCondition();
287      lhs_boundary.Type = BoundaryType.Dirichlet;
288      lhs_boundary.Value = c_min;
289
290      rhs_boundary = BoundaryCondition();
291      rhs_boundary.Type = BoundaryType.Dirichlet;
292      rhs_boundary.Value = c_max;
293
294      integration_method = IntegrationMethod();
295      integration_method.type = IntegrationType.Trapezoidal;
296      integration_method.gauss_points = 0; % not used for trapezoidal
297
298      l2_errors = [];
299
300      thetas = [0.0, 1.0, 0.5]; % Explicit Euler, Implicit Euler, Crank-Nicholson
301      method_names = {"Forward Euler", "Crank-Nicolson", "Backward Euler"};
302
303      for i = 1:length(thetas)
304          theta = thetas(i);
305
306              % solve numerically
307              numeric_solution = NumericSolver.SolveNumeric(
308                  mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
309                  integration_method);
310
311                  % compute L2 error
312                  l2_error = L2Error(analytical_solution, numeric_solution);
313                  l2_errors = [l2_errors, l2_error];
314      end
315
316      Plotter.PlotL2Errors(l2_errors, "L2 Error over Time", ...
317          "cw2/report/resources/part2/L2ErrorTimeIntegration", ...
318          method_names);
```

```

319
320      % perform stability analysis
321
322      tmax = 1.0;
323      element_count = 50;
324      dt_list = [0.0001, 0.001, 0.01, 0.1, 0.25];
325
326      % generate mesh
327      mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
328      mesh.Generate();
329
330      l2_errors_stability = [];
331
332      for i = 1:length(thetas)
333          theta = thetas(i);
334
335          l2_errors_dt = [];
336
337          for j = 1:length(dt_list)
338              dt = dt_list(j);
339
340              try
341
342                  fprintf("Testing %s with dt = %.4f...\n", method_names{i}, dt);
343
344                  numeric_solution = NumericSolver.SolveNumeric(...,
345                      mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
346                      integration_method);
347
348                  % CHECK FOR NaN/Inf at each timestep
349                  for t_idx = 1:length(numeric_solution.time)
350                      vals = numeric_solution.values(:, t_idx);
351                      if any(isnan(vals))
352                          fprintf("%s: NaN at step %d (t=%.4f)\n", method_names{i},
353                              t_idx, numeric_solution.time(t_idx));
354                          break;
355                      end
356                      if any(isinf(vals))
357                          fprintf("%s: Inf at step %d (t=%.4f)\n", method_names{i},
358                              t_idx, numeric_solution.time(t_idx));
359                          break;
360                      end
361
362                      if max(abs(vals)) > 1e10
363                          fprintf("%s: Explosion at step %d (t=%.4f), max=%e\n",
364                              method_names{i}, t_idx, numeric_solution.time(t_idx), max(abs(vals)));
365                          break;
366                      end
367                  end
368
369                  analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax,
370                      dt);
371
372                  l2_error = L2Error(analytical_solution, numeric_solution);
373
374                  l2_errors_dt = [l2_errors_dt, l2_error];
375                  catch
376                      l2_errors_dt = [l2_errors_dt, NaN];
377                      fprintf("%s EXPLODED \n", method_names{i});
378                  end
379
380                  l2_errors_stability = [l2_errors_stability; l2_errors_dt];
381
382              end
383
384      end

```

```
381     function Part2GaussianQuadrature()
382 %%%%%%
383 %
384 % Function:      Part2GaussianQuadrature()
385 %
386 % Arguments:    None
387 % Returns:      None
388 %
389 % Description:  Runs a study comparing L2 error with and without
390 %                 Gaussian Quadrature.
391 %
392 %%%%%%
393
394
395     % time parameters
396     tmax = 0.001;
397     dt = 0.0001;
398
399     % mesh parameters
400     xmin = 0.0;
401     xmax = 1.0;
402     element_count = 5;
403     order = 2;
404
405     % diffusion and reaction coefficients
406     D = 0.5;
407     lambda = 0.0;
408
409     % concentrations
410     c_max = 1.0;
411     c_min = 0.0;
412
413     % generate mesh
414     mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
415     mesh.Generate();
416
417     % solve analytically
418     analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
419
420     % solver parameters
421
422     theta = 0.5; % Crank-Nicholson
423
424     lhs_boundary = BoundaryCondition();
425     lhs_boundary.Type = BoundaryType.Dirichlet;
426     lhs_boundary.Value = c_min;
427
428     rhs_boundary = BoundaryCondition();
429     rhs_boundary.Type = BoundaryType.Dirichlet;
430     rhs_boundary.Value = c_max;
431
432     % trapezoidal method
433     trapezoidal_method = IntegrationMethod();
434     trapezoidal_method.type = IntegrationType.Trapezoidal;
435     trapezoidal_method.gauss_points = 0; % not used for trapezoidal
436
437     trapezoidal_solution = NumericSolver.SolveNumeric(
438         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
439         trapezoidal_method);
440
441     % gaussian quadrature method
442     gaussian_method = IntegrationMethod();
443     gaussian_method.type = IntegrationType.Gaussian;
444     gaussian_method.gauss_points = 3; % 3-point Gaussian quadrature
445
446     gaussian_solution = NumericSolver.SolveNumeric(...
```

```
446         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
447 gaussian_method);
448
449     % compute L2 error
450     l2_error_trapezoidal = L2Error(analytical_solution, trapezoidal_solution);
451     l2_error_gaussian = L2Error(analytical_solution, gaussian_solution);
452     l2_errors = [l2_error_trapezoidal, l2_error_gaussian];
453
454     method_names = {"2-point Trapezoidal Integration", "3-point Gaussian
455 Quadrature"};
456
457     Plotter.PlotL2Errors(l2_errors, "Gaussian vs Trapezoidal Integration", ...
458                         "cw2/report/resources/part2/L2ErrorGaussianTrapezoidal", ...
459                         method_names);
460 end
461
462 function Part3InitialResults()
463 %%%%%%
464 %
465 % Function:      Part2GaussianQuadrature()
466 %
467 % Arguments:    None
468 % Returns:      None
469 %
470 % Description:   Runs a study comparing L2 error with and without
471 %                  Gaussian Quadrature.
472 %
473 %%%%%%
474 %
475 % Generate mesh
476 xmin = 0;
477 xmax = 0.01;
478 element_count = 50;
479 order = 2;
480
481 theta = 0.5; % Crank-Nicholson
482 D = 1;
483 lambda = 0;
484
485 epidermis_layer = MeshLayer(0.0, 4e-6, 0.0, 0.02, 2.0);
486 dermis_layer = MeshLayer(0.00166667, 5e-6, 0.01, 0.02, 1.0);
487 sub_cutaneous_layer = MeshLayer(0.005, 2e-6, 0.01, 0.02, 1.0);
488
489 layers = [epidermis_layer, dermis_layer, sub_cutaneous_layer];
490
491 mesh = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers);
492 mesh.Generate();
493
494 tmax = 30.0;
495 dt = 0.01; % works well with element_count = 50
496
497 % concentrations
498 c_max = 30.0;
499 c_min = 0.0;
500
501 lhs_boundary = BoundaryCondition();
502 lhs_boundary.Type = BoundaryType.Dirichlet;
503 lhs_boundary.Value = c_max;
504
505 rhs_boundary = BoundaryCondition();
506 rhs_boundary.Type = BoundaryType.Dirichlet;
507 rhs_boundary.Value = c_min;
508
509 integration_method = IntegrationMethod();
510 integration_method.type = IntegrationType.Gaussian;
511 integration_method.gauss_points = order + 1;
```

```
511     numeric_solution = NumericSolver.SolveNumeric(...  
512         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,  
513         integration_method);  
514  
515     kappa = DoseEvaluator.EvaluateSolution(numeric_solution, 0.005, 4.0, dt);  
516  
517     fprintf('Kappa: %.2f\n', kappa);  
518  
519     Plotter.PlotHeatMap(numeric_solution, "Drug Concentration Heatmap", 'cw2/  
report/resources/part3/InitialNumericHeatmap', c_max);  
520     end  
521  
522 end  
523  
524 end  
525 end  
526
```

11. Mesh

11.1. Mesh.m

```
1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Mesh.m
6 % Author    : samh25
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
15 classdef Mesh < handle
16     % inherit from handle to allow pass-by-reference
17
18     properties
19
20         xmin double
21         xmax double
22         dx double
23
24         order double
25
26         D double % diffusion coefficient
27         lambda double % reaction coefficient
28
29         node_count uint64
30         node_coords double % coordinates of global nodes
31
32         element_count uint64
33         elements MeshElement % array of mesh elements
34
35     end
36
37     methods
38
39         %% Mesh constructor
40         function obj = Mesh(xmin, xmax, element_count, order, D, lambda)
41
42             obj.xmin = xmin;
43             obj.xmax = xmax;
44             obj.dx = (xmax - xmin) / element_count;
45             obj.D = D;
46             obj.lambda = lambda;
47
48             obj.order = order;
49
50             % total number of nodes
51             obj.node_count = (element_count * order) + 1;
52             obj.node_coords = zeros(1, obj.node_count);
53
54             obj.element_count = element_count;
55             obj.elements = MeshElement.empty(element_count, 0);
56
57         end
58
59         function obj = Generate(obj)
60
61             disp('Generating normal mesh... ');
62
63             % generate uniform node coordinates
```

```

64         obj.node_coords = linspace(obj.xmin, obj xmax, obj.node_count);
65
66         % generate elements
67         for e = 1:obj.element_count
68
69             % determine global node IDs for this element
70             node_start = (e - 1) * obj.order + 1;
71             node_ids = node_start:(node_start + obj.order);
72
73             % coordinates for this element
74             coords = obj.node_coords(node_ids);
75
76             % create MeshElement object
77             obj.elements(e) = MeshElement(node_ids, coords, obj.order, obj.D,
78             obj.lambda);
79         end
80     end
81 end
82 end
83

```

11.2. MeshElement.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : MeshElement.m
6 % Author    : samh25
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         %% MeshElement constructor
28         function obj = MeshElement(ids, coords, order, D, lambda)
29
30             % assign properties
31             obj.node_ids = ids;
32             obj.node_coords = coords;
33             obj.order = order;
34             obj.D = D;
35             obj.lambda = lambda;
36
37             % linear mapping from [-1, 1] to [x1, x2]
38             % jacobian = dx/dxi = (x2 - x1) / 2
39             obj.jacobian = (coords(end) - coords(1)) / 2;
40
41         end

```

```

42     end
43 end
44
45

```

11.3. MultilayerMesh.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Mesh.m
6 % Author    : samh25
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
15 classdef MultilayerMesh < Mesh
16     % inherit from handle to allow pass-by-reference
17
18     properties
19         layers    MeshLayer % array of layer properties
20         total_density    double
21     end
22
23     methods
24
25         %% Mesh constructor
26         function obj = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers)
27
28             obj = obj@Mesh(xmin, xmax, element_count, order, D, lambda);
29             obj.layers = layers;
30
31
32             % recalculate nodes and element counts based on layer densities
33             obj.total_density = 0.0;
34
35             for l = 1:length(layers)
36                 obj.total_density = obj.total_density + layers(l).density_ratio;
37             end
38
39             obj.element_count = 0;
40
41             for l = 1:length(layers)
42
43                 layer_density = obj.layers(l).density_ratio;
44                 layer_element_count = round((layer_density / obj.total_density) *
element_count);
45
46                 obj.layers(l).element_count = layer_element_count;
47                 obj.layers(l).layer_offset = obj.element_count + 1; % starting index for
this layer
48
49                 obj.element_count = obj.element_count + layer_element_count;
50
51                 disp(['Layer ' num2str(l) ' Element Count: '
num2str(layer_element_count)]);
52                 disp(obj.layers(l));
53             end
54
55             obj.node_count = (obj.element_count * order) + 1;

```

```
56     obj.node_coords = zeros(1, obj.node_count);
57
58     obj.elements = MeshElement.empty(obj.element_count, 0);
59
60 end
61
62 function obj = Generate(obj)
63
64 disp('Generating multilayer mesh... ');
65
66 % generate per-layer uniform node coordinates
67
68 current_node = 1;
69
70 for l = 1:length(obj.layers)
71
72     % calculate layer xmin and xmax
73
74     layer_xmin = obj.layers(l).x;
75
76     if l < length(obj.layers)
77         layer_xmax = obj.layers(l + 1).x;
78     else
79         layer_xmax = obj.xmax;
80     end
81
82     layer_nodes = obj.layers(l).element_count * obj.order + 1;
83
84     layer_coords = linspace(layer_xmin, layer_xmax, layer_nodes);
85
86
87     if l == 1
88         nodes_to_add = layer_coords;
89     else
90         nodes_to_add = layer_coords(2:end); % Skip duplicate boundary node
91     end
92
93     % Add nodes to global coordinate array
94     num_new_nodes = length(nodes_to_add);
95     obj.node_coords(current_node : current_node + num_new_nodes - 1) =
96     nodes_to_add;
97     current_node = current_node + num_new_nodes;
98
99 end
100
101 % Generate elements
102 for e = 1:obj.element_count
103
104     % Determine global node IDs for this element
105     node_start = (e - 1) * obj.order + 1;
106     node_ids = node_start:(node_start + obj.order);
107
108     % Coordinates for this element
109     coords = obj.node_coords(node_ids);
110
111     midpoint = (coords(1) + coords(end)) / 2;
112
113     % Determine which layer this element is in
114     layer_index = 1;
115     for l = 1:length(obj.layers)
116         if midpoint >= obj.layers(l).x
117             layer_index = l;
118         end
119     end
120
121     D = obj.layers(layer_index).D;
```

```
122         lambda = -(obj.layers(layer_index).beta + obj.layers(layer_index).gamma);  
123  
124             % Create MeshElement object  
125             obj.elements(e) = MeshElement(node_ids, coords, obj.order, D, lambda);  
126         end  
127     end  
128  
129 end  
130 end  
131  
132
```

11.4. LayerProperties.m

```
1 classdef LayerProperties  
2     properties  
3         x           double    % min x coordinate for this layer  
4         density_ratio double    % density ratio for this layer  
5         D           double    % diffusion coefficient  
6         beta        double    % extra-vascular diffusivity  
7         gamma       double    % drug degradation rate  
8     end  
9  
10    methods  
11        function obj = LayerProperties(x, D, beta, gamma, density_ratio)  
12            obj.x = x;  
13            obj.D = D;  
14            obj.beta = beta;  
15            obj.gamma = gamma;  
16            obj.density_ratio = density_ratio;  
17        end  
18    end  
19 end
```

12. Analytical

12.1. AnalyticalSolver.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : AnalyticalSolver.m
6 % Author    : samh25
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             % time vector
21             time_vector = 0:dt:tmax;
22             solution = Solution(mesh, time_vector);
23
24             % loop over time steps
25             for step = 1:length(time_vector)
26
27                 t = time_vector(step);
28                 timestep_results = zeros(1, mesh.node_count);
29
30                 % loop over nodes
31                 for i = 1:mesh.node_count
32                     x = mesh.node_coords(i);
33                     timestep_results(i) = TransientAnalyticSoln(x, t);
34                 end
35
36                 solution.SetValues(timestep_results, step);
37             end
38
39         end
40     end
41 end

```

12.2. 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 %
11 %%%%%%
12
13 function [ c ] = TransientAnalyticSoln(x,t)
14 %TransientAnalyticSoln Analytical solution to transient diffusion equation
15 % Computes the analytical solution to the transient diffusion equation for
16 % the domain x=[0,1], subject to initial condition: c(x,0) = 0, and Dirichlet
17 % boundary conditions: c(0,t) = 0, and c(1,t) = 1.
18 % Input Arguments:

```

```
19 % x is the point in space to evaluate the solution at
20 % t is the point in time to evaluate the solution at
21 % Output Argument:
22 % c is the value of concentration at point x and time t, i.e. c(x,t)
23
24 trans = 0.0;
25
26 for k=1:1000
27     trans = trans + ((((-1)^k)/k) * exp(-k^2*pi^2*t)*sin(k*pi*x));
28 end
29
30 c = x + (2/pi)*trans;
31
32 end
```

13. Plotter

13.1. Plotter.m

```
1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Plotter.m
6 % Author    : samh25
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A collection of static methods for plotting
10 %           results for the coursework.
11 %
12 %%%%%%
13
14 classdef Plotter
15
16     methods (Static)
17
18         %% Plot entire solution as a heatmap - time as x-axis, position as y-axis and
19         %% solution value as color
20         function PlotHeatMap(solution, title_str, name, c_max)
21
22             set(0, "DefaultAxesFontSize", 12);
23             set(0, "DefaultTextFontSize", 12);
24
25             % Plot a heat map of the solution values over time
26             figure;
27             imagesc(solution.time, solution.mesh.node_coords, solution.values);
28             colorbar;
29             xlabel("Time (t)");
30             ylabel("Displacement (x)");
31             caxis([0 c_max]) % lock color axis for consistency
32             axis xy; % ensure y-axis is oriented correctly
33             title(title_str);
34             grid off;
35
36             set(gcf, 'Position', [0, 0, 500, 350]);
37
38             % Save figure
39             saveas(gcf, name, "png");
40             saveas(gcf, name, "fig");
41             openfig(name + ".fig");
42
43         end
44
45         %% Plot full solution at specified time samples
46         function PlotTimeSamples(solution, dt, time_samples, title_str, name)
47
48             set(0, "DefaultAxesFontSize", 12);
49             set(0, "DefaultTextFontSize", 12);
50
51             figure;
52             plot_handle = 0;
53
54             for i = 1:length(time_samples)
55                 t_sample = time_samples(i);
56
57                 step_index = round(t_sample / dt) + 1; % +1 for MATLAB indexing
58
59                 plot_handle = plot(solution.mesh.node_coords, solution.values(:, step_index));
60                 set(plot_handle, "LineWidth", 1.5);
61
62                 hold on;
```

```
62         end
63
64         xlabel("Position (x)");
65         ylabel("c(x, t)");
66         title(title_str);
67
68         grid on;
69
70         legend_strings = cell(1, length(time_samples));
71         for i = 1:length(time_samples)
72             legend_strings{i} = ['t = ', num2str(time_samples(i))];
73         end
74
75         legend(legend_strings, "Location", "northwest");
76
77         set(gcf, 'Position', [0, 0, 500, 350]);
78
79         % Save figure
80         saveas(gcf, name, "png");
81         saveas(gcf, name, "fig");
82         openfig(name + ".fig");
83
84     end
85
86 %% Plot two solutions at a specific position over time
87 function PlotSampleOverTime(solution_1, solution_2, x_sample, title_str, name,
88                             legend_strings)
89
90     set(0, "DefaultAxesFontSize", 12);
91     set(0, "DefaultTextFontSize", 12);
92
93     % find x index
94     x_index = round((x_sample - solution_1.mesh.xmin) / (solution_1.mesh.xmax -
95                     solution_1.mesh.xmin) * solution_1.mesh.element_count) + 1; % +1 for MATLAB indexing
96
97     figure;
98     plot_handle = plot(solution_1.time, solution_1.values(x_index, :));
99     set(plot_handle, "LineWidth", 1.5);
100
101    hold on;
102
103    plot_handle = plot(solution_2.time, solution_2.values(x_index, :));
104    set(plot_handle, "LineWidth", 1.5);
105
106    xlabel("Time (t)");
107
108    ylabel("c(" + num2str(x_sample) + ", t)");
109    title(title_str);
110
111    grid on;
112
113    legend(legend_strings, "Location", "southeast");
114    set(gcf, 'Position', [0, 0, 500, 350]);
115
116    % Save figure
117    saveas(gcf, name, "png");
118    saveas(gcf, name, "fig");
119    openfig(name + ".fig");
120
121    function PlotConvergenceError(x_values, y_values, title_str, name, x_label,
122                                 y_label)
123
124        set(0, "DefaultAxesFontSize", 12);
125        set(0, "DefaultTextFontSize", 12);
126
127        figure;
```

```
126
127     plot_handle = loglog(x_values, y_values);
128     set(plot_handle, "LineWidth", 1.5);
129
130     xlabel(x_label);
131     ylabel(y_label);
132     title(title_str);
133     grid on;
134
135     set(gcf, 'Position', [0, 0, 500, 350]);
136
137 % Save figure
138 saveas(gcf, name, "png");
139 saveas(gcf, name, "fig");
140 openfig(name + ".fig");
141
142 end
143
144 function PlotL2Errors(l2_errors, title_str, name, legend_strings)
145
146     set(0, "DefaultAxesFontSize", 12);
147     set(0, "DefaultTextFontSize", 12);
148
149     figure;
150
151     for i = 1:length(l2_errors)
152         l2_error = l2_errors(i);
153         plot_handle = plot(l2_error.time, l2_error.l2_error);
154         set(plot_handle, "LineWidth", 1.5);
155         hold on;
156     end
157
158     xlabel("Time (t)");
159     ylabel("L2 Error");
160     title(title_str);
161
162     grid on;
163
164     legend(legend_strings, "Location", "northeast");
165     set(gcf, 'Position', [0, 0, 500, 350]);
166
167 % Save figure
168 saveas(gcf, name, "png");
169 saveas(gcf, name, "fig");
170
171 openfig(name + ".fig");
172
173 end
174
175 function PlotTwoConvergenceLines(x_values, y1_values, y2_values, title_str, name,
176 x_label, y_label, legend_strings)
177     set(0, "DefaultAxesFontSize", 12);
178     set(0, "DefaultTextFontSize", 12);
179
180     figure;
181
182     loglog(x_values, y1_values, '-o', 'LineWidth', 1.5, 'MarkerSize', 8);
183     hold on;
184     loglog(x_values, y2_values, '-s', 'LineWidth', 1.5, 'MarkerSize', 8);
185
186     xlabel(x_label);
187     ylabel(y_label);
188     title(title_str);
189     legend(legend_strings, 'Location', 'best');
190     grid on;
191
192     set(gcf, 'Position', [0, 0, 500, 350]);
```

```
192     saveas(gcf, name, "png");
193     saveas(gcf, name, "fig");
194     openfig(name + ".fig");
195 end
196
197
198 end
199 end
```

14. Solution

14.1. Solution.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Solution.m
6 % Author    : samh25
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         %% Solution constructor
28         function obj = Solution(mesh, time_vector)
29
30             % assign properties
31             obj.mesh = mesh;
32             obj.time = time_vector;
33             obj.values = zeros(mesh.node_count, length(time_vector));
34
35         end
36
37         %% Set solution values at given time step
38         function SetValues(obj, values, step)
39             % set solution values at given time step
40             obj.values(:, step) = values(:, );
41         end
42     end
43 end
44
45

```

14.2. L2Error.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : L2Error.m
6 % Author    : samh25
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 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     %% Solution constructor
30     function obj = L2Error(ref_solution, num_solution)
31
32         % assign properties
33         obj.ref_solution = ref_solution;
34         obj.num_solution = num_solution;
35         obj.time = ref_solution.time;
36
37         if ref_solution.mesh.node_count ~= num_solution.mesh.node_count
38             error('Reference and error solutions must have the same number of nodes');
39         end
40
41         if length(ref_solution.time) ~= length(num_solution.time)
42             error('Reference and error solutions must have the same number of time
43 steps');
44         end
45
46         step_count = length(ref_solution.time);
47
48         obj.l2_error = zeros(1, step_count);
49
50         for step = 1:step_count
51             c_ref = ref_solution.values(:, step);
52             c_num = num_solution.values(:, step);
53             x = ref_solution.mesh.node_coords;
54
55             integrand = (c_ref - c_num).^2;
56             obj.l2_error(step) = sqrt(trapz(x, integrand));
57         end
58     end
59
60 end
61 end
62
63

```

14.3. DoseEvaluator.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Evaluation.m
6 % Author    : samh25
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A static class defining a dose evaluator of
10 %                a solution
11 %
12 %%%%%%

```

```
13
14 classdef DoseEvaluator
15
16 methods (Static)
17
18     function K = EvaluateSolution(solution, target_x, c_threshold, dt)
19
20         % first, find the closest node to the target
21         node_index = 0;
22
23         for i = 1:solution.mesh.node_count
24             x = solution.mesh.node_coords(i);
25             if x >= target_x
26                 node_index = i;
27                 break;
28             end
29         end
30
31         fprintf("node index %d\n", node_index);
32         c = solution.values(node_index, :);
33
34         effective_t_index = 0;
35
36         for i = 1:length(c)
37             if c(i) > c_threshold
38                 effective_t_index = i;
39                 break
40             end
41         end
42
43         fprintf("effective t index %d\n", effective_t_index);
44
45         if effective_t_index == 0
46             K = 0; % never exceeds threshold
47             return;
48         end
49
50         % integrate concentration over time until effective_t_index
51         time_range = effective_t_index:length(solution.time);
52         K = trapz(c(time_range)) * dt;
53     end
54 end
55 end
```

15. Solver

15.1. NumericSolver.m

```

1 %%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : NumericSolver.m
6 % Author    : samh25
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             % time vector
23             time_vector = 0:dt:tmax;
24             solution = Solution(mesh, time_vector);
25
26             % === SET INITIAL CONDITION EXPLICITLY ===
27             c0 = zeros(mesh.node_count, 1);
28             solution.SetValues(c0, 1); % column 1 = t=0
29
30             [K, M] = NumericSolver.CreateGlobalMatrices(mesh, theta, integration_method);
31
32             % loop over time steps
33             for step = 1:length(time_vector) - 1
34
35                 c_next = NumericSolver.SolveStep(mesh, solution, step, dt, theta, K, M,
36                     left_boundary, right_boundary, source_fn, integration_method);
37                 solution.SetValues(c_next, step + 1);
38
39             end
40
41         end
42
43         function c = SolveStep(mesh, solution, step, dt, theta, K, M, left_boundary,
44             right_boundary, source_fn, integration_method)
45
46             c_current = solution.values(:, step);
47
48             t = (step - 1) * dt; % current time, converted to 0-based index
49
50             % assemble system matrix and rhs vector
51             system_matrix = M + theta * dt * K;
52             rhs_vector = (M - (1 - theta) * dt * K) * c_current;
53
54             % add source term
55             f_current = NumericSolver.CreateSourceVector(mesh, t, source_fn,
56                 integration_method);
57             f_next = NumericSolver.CreateSourceVector(mesh, t + dt, source_fn,
58                 integration_method);
59             rhs_vector = rhs_vector + dt * (theta * f_next + (1 - theta) * f_current);
60
61             % apply boundary conditions

```

```
57     [system_matrix, rhs_vector] =
58     NumericSolver.ApplyBoundaryConditions(system_matrix, rhs_vector, t + dt, left_boundary,
59                                         right_boundary);
60
61     % solve system
62     c = system_matrix \ rhs_vector;
63
64 %% Create global stiffness and mass matrices
65 function [K, M] = CreateGlobalMatrices(mesh, theta, integration_method)
66
67     num_elements = mesh.element_count;
68     num_nodes = mesh.node_count;
69
70     % initialise global matrix (use sparse for efficiency with large systems)
71     K = sparse(num_nodes, num_nodes);
72     M = sparse(num_nodes, num_nodes);
73
74     for element_id = 1:num_elements
75
76         element = mesh.elements(element_id);
77         nodes = element.node_ids;
78         local_size = length(nodes);
79
80         diff_matrix = ElementMatrices.DiffusionElemMatrix(element,
81 integration_method);
82         react_matrix = ElementMatrices.ReactionElemMatrix(element,
83 integration_method);
84
85         k_matrix = diff_matrix - react_matrix;
86
87         elem_size = element.node_coords(end) - element.node_coords(1);
88         m_matrix = ElementMatrices.MassElemMatrix(element, integration_method);
89
90         % assemble into global matrices
91         for i = 1:local_size
92             for j = 1:local_size
93                 gi = nodes(i); gj = nodes(j);
94                 K(gi, gj) = K(gi, gj) + k_matrix(i, j);
95                 M(gi, gj) = M(gi, gj) + m_matrix(i, j);
96             end
97         end
98
99     end
100
101 %% Create source vector for given time
102 function F = CreateSourceVector(mesh, t, source_fn, integration_method)
103
104     F = zeros(mesh.node_count, 1);
105
106     % return if no source function defined
107     if (isempty(source_fn))
108         return;
109     end
110
111     for element_id = 1:mesh.element_count
112
113         element = mesh.elements(element_id);
114
115         elem_size = element.node_coords(end) - element.node_coords(1);
116         midpoint = (element.node_coords(1) + element.node_coords(end)) / 2;
117
118         f_val = source_fn(midpoint, t);
119
120     end
121
122 %% Create boundary conditions
123 function bc = CreateBoundaryConditions(mesh, t, left_boundary, right_boundary)
124
125     bc = zeros(mesh.node_count, 1);
126
127     % set boundary values
128     bc(left_boundary) = 1;
129     bc(right_boundary) = 1;
130
131     % set interior values to zero
132     for i = 1:mesh.node_count
133         if (i < left_boundary || i > right_boundary)
134             bc(i) = 0;
135         end
136     end
137
138 %% Create initial conditions
139 function IC = CreateInitialConditions(mesh, t, initial_fn)
140
141     IC = zeros(mesh.node_count, 1);
142
143     % evaluate initial function at each node
144     for i = 1:mesh.node_count
145         IC(i) = initial_fn(mesh.nodes(i));
146     end
147
148 %% Create element matrices
149 function [K, M] = CreateElementMatrices(element, integration_method)
150
151     % get element properties
152     n = element.node_ids;
153     x = element.node_coords;
154
155     % calculate element size
156     h = x(end) - x(1);
157
158     % calculate element shape functions
159     N1 = 1 - (x - x(1)) / h;
160     N2 = (x - x(1)) / h;
161
162     % calculate element Jacobian
163     J = [1, 0; 0, 1];
164
165     % calculate element area
166     A = det(J);
167
168     % calculate element mass matrix
169     M = A * diag(N1 * N1 * h, 0) + A * diag(N2 * N2 * h, 1);
170
171     % calculate element diffusion matrix
172     K = A * diag(N1 * N1 * h, 0) + A * diag(N2 * N2 * h, 1);
173
174     % calculate element reaction matrix
175     R = A * diag(N1 * N1 * h, 0) + A * diag(N2 * N2 * h, 1);
176
177     % assemble element matrices into global matrices
178     for i = 1:element.node_ids
179         for j = 1:element.node_ids
180             K(i, j) = K(i, j) + K;
181             M(i, j) = M(i, j) + M;
182             R(i, j) = R(i, j) + R;
183         end
184     end
185
186     % return element matrices
187     K = K / A;
188     M = M / A;
189     R = R / A;
190
191 %% Create element function
192 function f = CreateElementFunction(element, integration_method)
193
194     % get element properties
195     n = element.node_ids;
196     x = element.node_coords;
197
198     % calculate element size
199     h = x(end) - x(1);
200
201     % calculate element shape functions
202     N1 = 1 - (x - x(1)) / h;
203     N2 = (x - x(1)) / h;
204
205     % calculate element Jacobian
206     J = [1, 0; 0, 1];
207
208     % calculate element area
209     A = det(J);
210
211     % calculate element function
212     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
213         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
214         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
215         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
216
217 %% Create element function
218 function f = CreateElementFunction(element, integration_method)
219
220     % get element properties
221     n = element.node_ids;
222     x = element.node_coords;
223
224     % calculate element size
225     h = x(end) - x(1);
226
227     % calculate element shape functions
228     N1 = 1 - (x - x(1)) / h;
229     N2 = (x - x(1)) / h;
230
231     % calculate element Jacobian
232     J = [1, 0; 0, 1];
233
234     % calculate element area
235     A = det(J);
236
237     % calculate element function
238     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
239         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
240         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
241         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
242
243 %% Create element function
244 function f = CreateElementFunction(element, integration_method)
245
246     % get element properties
247     n = element.node_ids;
248     x = element.node_coords;
249
250     % calculate element size
251     h = x(end) - x(1);
252
253     % calculate element shape functions
254     N1 = 1 - (x - x(1)) / h;
255     N2 = (x - x(1)) / h;
256
257     % calculate element Jacobian
258     J = [1, 0; 0, 1];
259
260     % calculate element area
261     A = det(J);
262
263     % calculate element function
264     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
265         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
266         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
267         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
268
269 %% Create element function
270 function f = CreateElementFunction(element, integration_method)
271
272     % get element properties
273     n = element.node_ids;
274     x = element.node_coords;
275
276     % calculate element size
277     h = x(end) - x(1);
278
279     % calculate element shape functions
280     N1 = 1 - (x - x(1)) / h;
281     N2 = (x - x(1)) / h;
282
283     % calculate element Jacobian
284     J = [1, 0; 0, 1];
285
286     % calculate element area
287     A = det(J);
288
289     % calculate element function
290     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
291         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
292         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
293         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
294
295 %% Create element function
296 function f = CreateElementFunction(element, integration_method)
297
298     % get element properties
299     n = element.node_ids;
300     x = element.node_coords;
301
302     % calculate element size
303     h = x(end) - x(1);
304
305     % calculate element shape functions
306     N1 = 1 - (x - x(1)) / h;
307     N2 = (x - x(1)) / h;
308
309     % calculate element Jacobian
310     J = [1, 0; 0, 1];
311
312     % calculate element area
313     A = det(J);
314
315     % calculate element function
316     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
317         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
318         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
319         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
320
321 %% Create element function
322 function f = CreateElementFunction(element, integration_method)
323
324     % get element properties
325     n = element.node_ids;
326     x = element.node_coords;
327
328     % calculate element size
329     h = x(end) - x(1);
330
331     % calculate element shape functions
332     N1 = 1 - (x - x(1)) / h;
333     N2 = (x - x(1)) / h;
334
335     % calculate element Jacobian
336     J = [1, 0; 0, 1];
337
338     % calculate element area
339     A = det(J);
340
341     % calculate element function
342     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
343         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
344         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
345         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
346
347 %% Create element function
348 function f = CreateElementFunction(element, integration_method)
349
350     % get element properties
351     n = element.node_ids;
352     x = element.node_coords;
353
354     % calculate element size
355     h = x(end) - x(1);
356
357     % calculate element shape functions
358     N1 = 1 - (x - x(1)) / h;
359     N2 = (x - x(1)) / h;
360
361     % calculate element Jacobian
362     J = [1, 0; 0, 1];
363
364     % calculate element area
365     A = det(J);
366
367     % calculate element function
368     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
369         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
370         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
371         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
372
373 %% Create element function
374 function f = CreateElementFunction(element, integration_method)
375
376     % get element properties
377     n = element.node_ids;
378     x = element.node_coords;
379
380     % calculate element size
381     h = x(end) - x(1);
382
383     % calculate element shape functions
384     N1 = 1 - (x - x(1)) / h;
385     N2 = (x - x(1)) / h;
386
387     % calculate element Jacobian
388     J = [1, 0; 0, 1];
389
390     % calculate element area
391     A = det(J);
392
393     % calculate element function
394     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
395         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
396         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
397         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
398
399 %% Create element function
400 function f = CreateElementFunction(element, integration_method)
401
402     % get element properties
403     n = element.node_ids;
404     x = element.node_coords;
405
406     % calculate element size
407     h = x(end) - x(1);
408
409     % calculate element shape functions
410     N1 = 1 - (x - x(1)) / h;
411     N2 = (x - x(1)) / h;
412
413     % calculate element Jacobian
414     J = [1, 0; 0, 1];
415
416     % calculate element area
417     A = det(J);
418
419     % calculate element function
420     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
421         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
422         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
423         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
424
425 %% Create element function
426 function f = CreateElementFunction(element, integration_method)
427
428     % get element properties
429     n = element.node_ids;
430     x = element.node_coords;
431
432     % calculate element size
433     h = x(end) - x(1);
434
435     % calculate element shape functions
436     N1 = 1 - (x - x(1)) / h;
437     N2 = (x - x(1)) / h;
438
439     % calculate element Jacobian
440     J = [1, 0; 0, 1];
441
442     % calculate element area
443     A = det(J);
444
445     % calculate element function
446     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
447         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
448         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
449         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
450
451 %% Create element function
452 function f = CreateElementFunction(element, integration_method)
453
454     % get element properties
455     n = element.node_ids;
456     x = element.node_coords;
457
458     % calculate element size
459     h = x(end) - x(1);
460
461     % calculate element shape functions
462     N1 = 1 - (x - x(1)) / h;
463     N2 = (x - x(1)) / h;
464
465     % calculate element Jacobian
466     J = [1, 0; 0, 1];
467
468     % calculate element area
469     A = det(J);
470
471     % calculate element function
472     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
473         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
474         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
475         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
476
477 %% Create element function
478 function f = CreateElementFunction(element, integration_method)
479
480     % get element properties
481     n = element.node_ids;
482     x = element.node_coords;
483
484     % calculate element size
485     h = x(end) - x(1);
486
487     % calculate element shape functions
488     N1 = 1 - (x - x(1)) / h;
489     N2 = (x - x(1)) / h;
490
491     % calculate element Jacobian
492     J = [1, 0; 0, 1];
493
494     % calculate element area
495     A = det(J);
496
497     % calculate element function
498     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
499         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
500         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
501         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
502
503 %% Create element function
504 function f = CreateElementFunction(element, integration_method)
505
506     % get element properties
507     n = element.node_ids;
508     x = element.node_coords;
509
510     % calculate element size
511     h = x(end) - x(1);
512
513     % calculate element shape functions
514     N1 = 1 - (x - x(1)) / h;
515     N2 = (x - x(1)) / h;
516
517     % calculate element Jacobian
518     J = [1, 0; 0, 1];
519
520     % calculate element area
521     A = det(J);
522
523     % calculate element function
524     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
525         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
526         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
527         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
528
529 %% Create element function
530 function f = CreateElementFunction(element, integration_method)
531
532     % get element properties
533     n = element.node_ids;
534     x = element.node_coords;
535
536     % calculate element size
537     h = x(end) - x(1);
538
539     % calculate element shape functions
540     N1 = 1 - (x - x(1)) / h;
541     N2 = (x - x(1)) / h;
542
543     % calculate element Jacobian
544     J = [1, 0; 0, 1];
545
546     % calculate element area
547     A = det(J);
548
549     % calculate element function
550     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
551         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
552         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
553         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
554
555 %% Create element function
556 function f = CreateElementFunction(element, integration_method)
557
558     % get element properties
559     n = element.node_ids;
560     x = element.node_coords;
561
562     % calculate element size
563     h = x(end) - x(1);
564
565     % calculate element shape functions
566     N1 = 1 - (x - x(1)) / h;
567     N2 = (x - x(1)) / h;
568
569     % calculate element Jacobian
570     J = [1, 0; 0, 1];
571
572     % calculate element area
573     A = det(J);
574
575     % calculate element function
576     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
577         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
578         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
579         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
580
581 %% Create element function
582 function f = CreateElementFunction(element, integration_method)
583
584     % get element properties
585     n = element.node_ids;
586     x = element.node_coords;
587
588     % calculate element size
589     h = x(end) - x(1);
590
591     % calculate element shape functions
592     N1 = 1 - (x - x(1)) / h;
593     N2 = (x - x(1)) / h;
594
595     % calculate element Jacobian
596     J = [1, 0; 0, 1];
597
598     % calculate element area
599     A = det(J);
600
601     % calculate element function
602     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
603         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
604         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
605         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
606
607 %% Create element function
608 function f = CreateElementFunction(element, integration_method)
609
610     % get element properties
611     n = element.node_ids;
612     x = element.node_coords;
613
614     % calculate element size
615     h = x(end) - x(1);
616
617     % calculate element shape functions
618     N1 = 1 - (x - x(1)) / h;
619     N2 = (x - x(1)) / h;
620
621     % calculate element Jacobian
622     J = [1, 0; 0, 1];
623
624     % calculate element area
625     A = det(J);
626
627     % calculate element function
628     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
629         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
630         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
631         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
632
633 %% Create element function
634 function f = CreateElementFunction(element, integration_method)
635
636     % get element properties
637     n = element.node_ids;
638     x = element.node_coords;
639
640     % calculate element size
641     h = x(end) - x(1);
642
643     % calculate element shape functions
644     N1 = 1 - (x - x(1)) / h;
645     N2 = (x - x(1)) / h;
646
647     % calculate element Jacobian
648     J = [1, 0; 0, 1];
649
650     % calculate element area
651     A = det(J);
652
653     % calculate element function
654     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
655         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
656         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
657         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
658
659 %% Create element function
660 function f = CreateElementFunction(element, integration_method)
661
662     % get element properties
663     n = element.node_ids;
664     x = element.node_coords;
665
666     % calculate element size
667     h = x(end) - x(1);
668
669     % calculate element shape functions
670     N1 = 1 - (x - x(1)) / h;
671     N2 = (x - x(1)) / h;
672
673     % calculate element Jacobian
674     J = [1, 0; 0, 1];
675
676     % calculate element area
677     A = det(J);
678
679     % calculate element function
680     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
681         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
682         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
683         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
684
685 %% Create element function
686 function f = CreateElementFunction(element, integration_method)
687
688     % get element properties
689     n = element.node_ids;
690     x = element.node_coords;
691
692     % calculate element size
693     h = x(end) - x(1);
694
695     % calculate element shape functions
696     N1 = 1 - (x - x(1)) / h;
697     N2 = (x - x(1)) / h;
698
699     % calculate element Jacobian
700     J = [1, 0; 0, 1];
701
702     % calculate element area
703     A = det(J);
704
705     % calculate element function
706     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
707         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
708         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
709         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
710
711 %% Create element function
712 function f = CreateElementFunction(element, integration_method)
713
714     % get element properties
715     n = element.node_ids;
716     x = element.node_coords;
717
718     % calculate element size
719     h = x(end) - x(1);
720
721     % calculate element shape functions
722     N1 = 1 - (x - x(1)) / h;
723     N2 = (x - x(1)) / h;
724
725     % calculate element Jacobian
726     J = [1, 0; 0, 1];
727
728     % calculate element area
729     A = det(J);
730
731     % calculate element function
732     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
733         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
734         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
735         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
736
737 %% Create element function
738 function f = CreateElementFunction(element, integration_method)
739
740     % get element properties
741     n = element.node_ids;
742     x = element.node_coords;
743
744     % calculate element size
745     h = x(end) - x(1);
746
747     % calculate element shape functions
748     N1 = 1 - (x - x(1)) / h;
749     N2 = (x - x(1)) / h;
750
751     % calculate element Jacobian
752     J = [1, 0; 0, 1];
753
754     % calculate element area
755     A = det(J);
756
757     % calculate element function
758     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
759         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
760         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
761         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
762
763 %% Create element function
764 function f = CreateElementFunction(element, integration_method)
765
766     % get element properties
767     n = element.node_ids;
768     x = element.node_coords;
769
770     % calculate element size
771     h = x(end) - x(1);
772
773     % calculate element shape functions
774     N1 = 1 - (x - x(1)) / h;
775     N2 = (x - x(1)) / h;
776
777     % calculate element Jacobian
778     J = [1, 0; 0, 1];
779
780     % calculate element area
781     A = det(J);
782
783     % calculate element function
784     f = N1 * (x(1) * (x(1) - x(2)) * (x(1) - x(3)) * (x(1) - x(4))) / (6 * A) +
785         N2 * (x(2) * (x(2) - x(1)) * (x(2) - x(3)) * (x(2) - x(4))) / (6 * A) +
786         N1 * (x(3) * (x(3) - x(1)) * (x(3) - x(2)) * (x(3) - x(4))) / (6 * A) +
787         N2 * (x(4) * (x(4) - x(1)) * (x(4) - x(2)) * (x(4) - x(3))) / (6 * A);
788
789 %% Create element function
790 function f = CreateElementFunction(element, integration_method)
791
792     % get element properties
793     n = element.node_ids;
794     x = element.node_coords;
795
796     % calculate element size
797     h = x(end) - x(1);
798
799     % calculate element shape functions
800     N1 = 1 - (x - x(1)) / h;
801     N2 = (x - x(1)) / h;
802
803     % calculate element Jacobian
804     J = [1, 0; 0, 1];
805
806     % calculate element area
807     A = det(J);
808
809     % calculate element function
810     f = N1 * (x(1
```

```

120             % Local Force Vector for linear element (Int N^T * s dx)
121             f_local = f_val * ElementMatrices.ForceMatrix(element, integration_method);
122
123             nodes = element.node_ids;
124             F(nodes) = F(nodes) + f_local;
125         end
126
127     end
128
129     function [lhs, rhs] = ApplyBoundaryConditions(lhs, rhs, t, left_boundary,
130                                                 right_boundary)
131
132         % Store diagonal values before modification
133         diag_left = lhs(1,1);
134         diag_right = lhs(end,end);
135
136         % apply left boundary condition
137         switch left_boundary.Type
138
139             case BoundaryType.Dirichlet
140                 lhs(1, :) = 0;                                % clear row
141                 lhs(1, 1) = diag_left;                      % keep diagonal
142                 rhs(1) = left_boundary.Value * diag_left;    % scale by diagonal
143
144             case BoundaryType.Neumann
145                 rhs(1) = rhs(1) + left_boundary.ValueFunction(t); % apply flux
146
147         end
148
149
150         % apply right boundary condition
151         switch right_boundary.Type
152
153             case BoundaryType.Dirichlet
154                 lhs(end, :) = 0;                            % clear row
155                 lhs(end, end) = diag_right;               % set diagonal to 1
156                 rhs(end) = right_boundary.Value * diag_right; % set value
157
158             case BoundaryType.Neumann
159                 rhs(end) = rhs(end) + right_boundary.ValueFunction(t); % apply flux
160
161         end
162
163     end
164 end
165

```

15.2. BoundaryCondition.m

```

1 classdef BoundaryCondition
2     properties
3         Type BoundaryType % Boundary condition type (Dirichlet or Neumann)
4         Value double % Boundary condition value for Dirichlet
5         ValueFunction function_handle % Boundary condition function for Neumann - parameter
6         t, return double
7     end
7 end

```

15.3. BoundaryType.m

```

1 classdef BoundaryType
2     enumeration
3         Dirichlet, Neumann

```

```

4     end
5 end

```

15.4. ElementMatrices.m

```

1 classdef ElementMatrices
2
3     methods (Static)
4
5         function matrix = DiffusionElemMatrix(element, method)
6
7             elem_size = element.node_coords(end) - element.node_coords(1);
8
9             if method.type == IntegrationType.Trapezoidal
10
11                 % create base matrix
12                 matrix = eye(element.order + 1);
13
14                 for i = 1:(element.order + 1)
15                     for j = 1:(element.order + 1)
16                         if i ~= j
17                             matrix(i, j) = -1;
18                         end
19                     end
20                 end
21
22                 % apply matrix scaling
23                 matrix = matrix * (element.D / elem_size);
24
25             else
26
27                 matrix = zeros(element.order + 1);
28                 [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
29
30                 for i = 1:length(xi)
31                     dN_dx = ElementMatrices.ShapeFunctionDerivatives(element.order,
32                         xi(i));
33
34                     J = element.jacobian;
35                     dN_dx = dN_dx / J;
36
37                     % compute contribution to stiffness matrix
38                     matrix = matrix + (element.D * (dN_dx' * dN_dx)) * (wi(i) * J);
39                 end
40             end
41
42         end
43
44         function matrix = ReactionElemMatrix(element, method)
45
46             elem_size = element.node_coords(end) - element.node_coords(1);
47
48             if method.type == IntegrationType.Trapezoidal
49
50                 % create base matrix
51                 matrix = eye(element.order + 1) * 2;
52
53                 for i = 1:(element.order + 1)
54                     for j = 1:(element.order + 1)
55                         if i ~= j
56                             matrix(i, j) = 1;
57                         end
58                     end
59                 end

```

```
60      % apply matrix scaling
61      matrix = matrix * (element.lambda * elem_size / 6);
62
63  else
64
65      matrix = zeros(element.order + 1);
66      [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
67
68      for i = 1:length(xi)
69          N = ElementMatrices.ShapeFunctions(element.order, xi(i));
70
71          J = element.jacobian;
72
73          % compute contribution to stiffness matrix
74          matrix = matrix + (element.lambda * (N' * N)) * (wi(i) * J);
75      end
76  end
77
78
79 function matrix = MassElemMatrix(element, method)
80
81     elem_size = element.node_coords(end) - element.node_coords(1);
82
83     if method.type == IntegrationType.Trapezoidal
84
85         % create base matrix
86         matrix = eye(element.order + 1) * 2;
87
88         for i = 1:(element.order + 1)
89             for j = 1:(element.order + 1)
90                 if i ~= j
91                     matrix(i, j) = 1;
92                 end
93             end
94         end
95
96         % apply matrix scaling
97         matrix = matrix * (elem_size / 6);
98
99     else
100
101         matrix = zeros(element.order + 1);
102         [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
103
104         for i = 1:length(xi)
105             N = ElementMatrices.ShapeFunctions(element.order, xi(i));
106
107             J = element.jacobian;
108
109             % compute contribution to stiffness matrix
110             matrix = matrix + (N' * N) * (wi(i) * J);
111         end
112
113     end
114
115
116 end
117
118 function matrix = ForceMatrix(element, method)
119
120     elem_size = element.node_coords(end) - element.node_coords(1);
121
122     if method.type == IntegrationType.Trapezoidal
123
124         % create base matrix
125         matrix = ones(element.order + 1, 1);
```

```
127         % apply matrix scaling
128         matrix = matrix * (elem_size / 2);
129
130     else
131
132         matrix = zeros(element.order + 1, 1);
133         [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
134
135         for i = 1:length(xi)
136             N = ElementMatrices.ShapeFunctions(element.order, xi(i));
137
138             J = element.jacobian;
139
140             % compute contribution to stiffness matrix
141             matrix = matrix + N' * (wi(i) * J);
142         end
143
144     end
145
146 end
147
148 methods (Static, Access = private)
149
150     function [xi, wi] = GaussQuadraturePoints(n)
151
152         switch n
153             case 1
154                 xi = 0;
155                 wi = 2;
156             case 2
157                 xi = [-1/sqrt(3), 1/sqrt(3)];
158                 wi = [1, 1];
159             case 3
160                 xi = [-sqrt(3/5), 0, sqrt(3/5)];
161                 wi = [5/9, 8/9, 5/9];
162             otherwise
163                 error('Gauss quadrature for n > 3 not implemented.');
164         end
165
166     end
167
168     function N = ShapeFunctions(order, xi)
169
170         switch order
171             case 1 % linear
172                 N = [(1 - xi) / 2, (1 + xi) / 2];
173             case 2 % quadratic
174                 N = [xi * (xi - 1) / 2, (1 - xi^2), xi * (xi + 1) / 2];
175             otherwise
176                 error('Shape functions for order > 2 not implemented.');
177         end
178
179     end
180
181     function dN_dxi = ShapeFunctionDerivatives(order, xi)
182
183         switch order
184             case 1 % linear
185                 dN_dxi = [-0.5, 0.5];
186             case 2 % quadratic
187                 dN_dxi = [xi - 0.5, -2 * xi, xi + 0.5];
188             otherwise
189                 error('Shape function derivatives for order > 2 not implemented.');
190         end
191
192     end
193
```

```
194     end  
195 end
```

15.5. IntegrationMethod.m

```
1 classdef IntegrationMethod  
2     properties  
3         type          IntegrationType  
4         gauss_points uint8  
5     end  
6 end
```

15.6. IntegrationType.m

```
1 classdef IntegrationType  
2     enumeration  
3         Trapezoidal, Gaussian  
4     end  
5 end
```

16. Tests

16.1. NumericSolverTest.m

```
1 %%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : NumericSolverTest.m
6 % Author    : samh25
7 % Created   : 2025-11-27 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : Test suite for NumericSolver class
10 %
11 %%%%%%%%%%%%%%
12
13 function tests = NumericSolverTest
14     tests = functiontests(localfunctions);
15 end
16
17 function TestSolveNumericReactionOnly(testCase)
18
19     % mesh parameters
20     xmin = 0.0;
21     xmax = 1.0;
22     element_count = 6;
23     order = 1;
24     lambda = -1.0;
25     D = 0.0;
26
27     % time parameters
28     tmax = 0.5;
29     dt = 0.02;
30     theta = 0.5; % Crank-Nicholson
31
32     % generate mesh
33     mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
34     mesh.Generate();
35
36     % solver parameters
37     lhs_boundary = BoundaryCondition();
38     lhs_boundary.Type = BoundaryType.Neumann;
39     lhs_boundary.ValueFunction = @(t) 0.0;
40
41     rhs_boundary = BoundaryCondition();
42     rhs_boundary.Type = BoundaryType.Neumann;
43     rhs_boundary.ValueFunction = @(t) 0.0;
44
45     integration_method = IntegrationMethod();
46     integration_method.type = IntegrationType.Trapezoidal;
47     integration_method.gauss_points = 0; % not used for trapezoidal
48
49     numeric_solution = NumericSolver.SolveNumeric...
50         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @SourceFunction,
51         integration_method);
52
53     % analytical solution
54     t_analytic = 0:dt:tmax;
55     c_exact = 10 * (1 - exp(lambda * t_analytic));
56
57     % chose random node (3 in this case) to compare
58     c_numeric = numeric_solution.values(3,:);
59     error = norm(c_numeric - c_exact) / norm(c_exact);
60
61     tolerance = 1e-3;
62     verifyLessThan(testCase, error, tolerance);
63 end
```

```
63
64 function s = SourceFunction(x, t)
65     s = 10;
66 end
67
```