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).

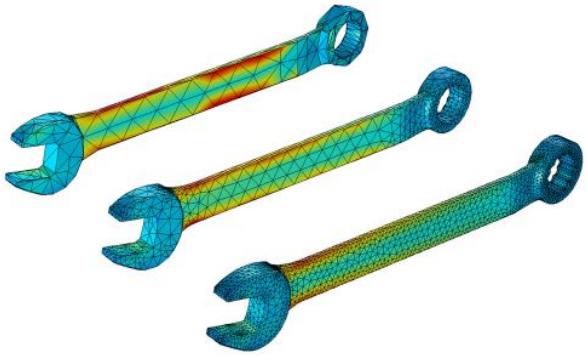

Figure 1: Finite Element Modelling of a Wrench under a Test Load Scenario [2]

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

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

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:

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

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

2.2. Software Architecture

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

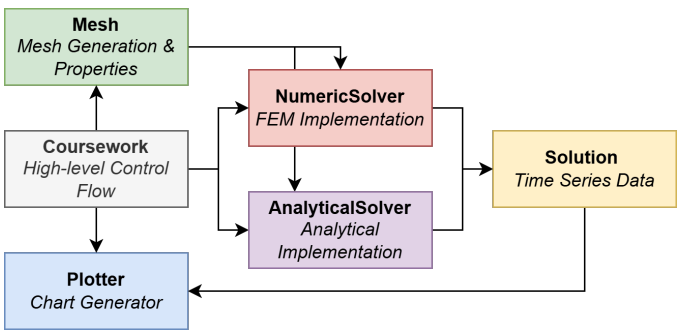

Figure 2: High-Level Software Architecture of the FEM Solver

2.3. Results

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

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

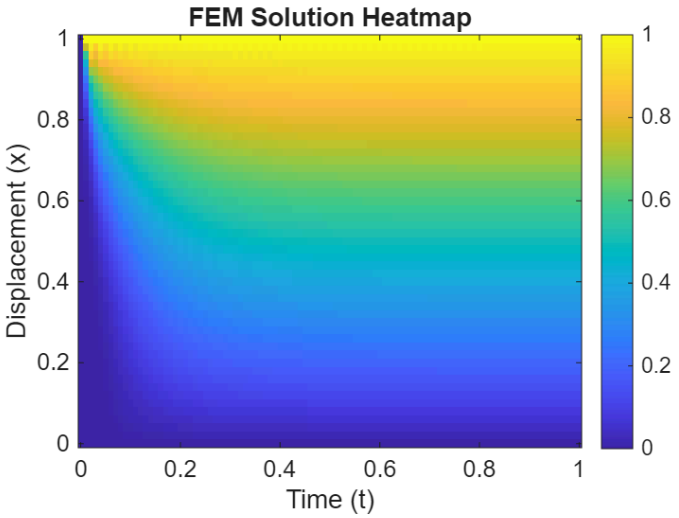

Figure 3: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

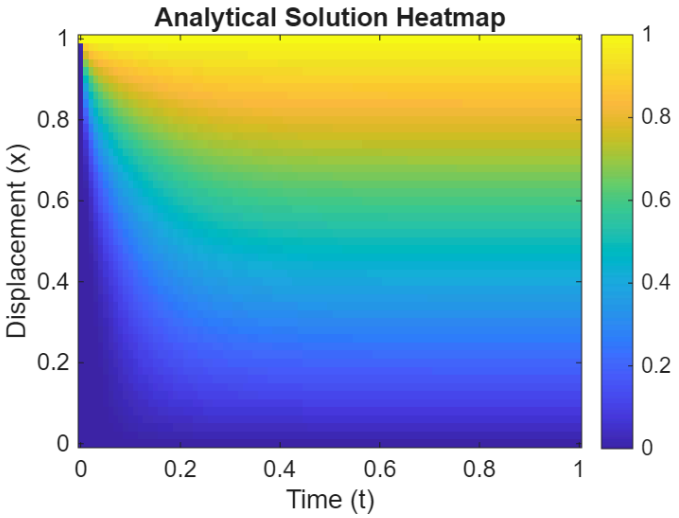

Figure 4: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

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

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

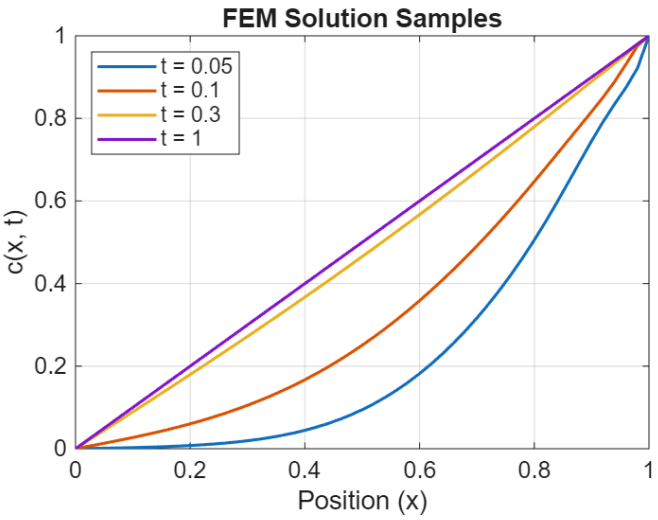

Figure 5: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

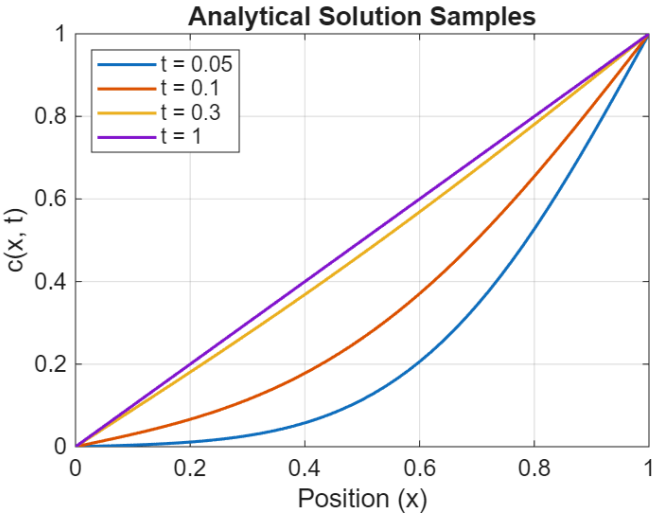

Figure 6: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

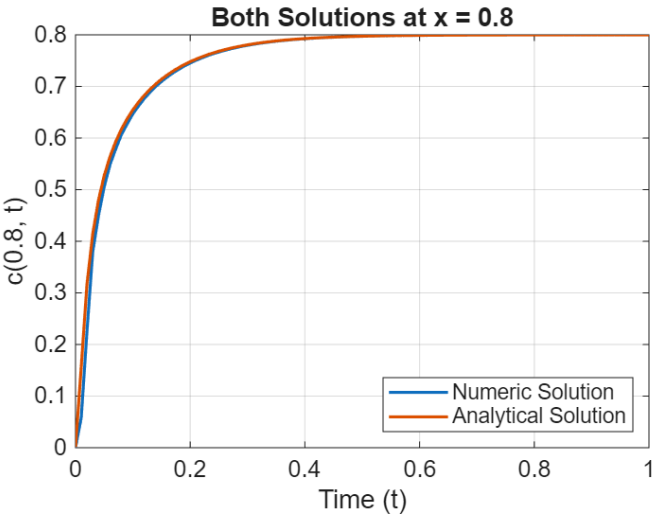

Figure 7: Comparison of Analytical and FEM Solutions at $x = 0.8$ over $0 \leq t \leq 1s$

2.4. Spacial and Temporal Convergence

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

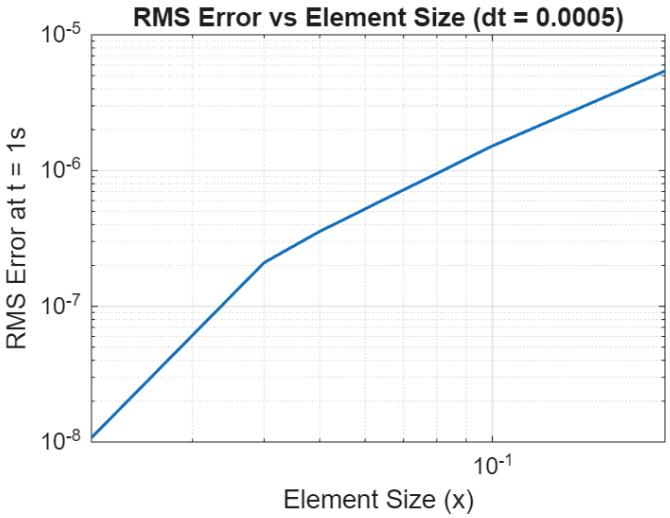

Figure 8: Comparison of RMS errors at $t = 1s$ for Varying Element Sizes

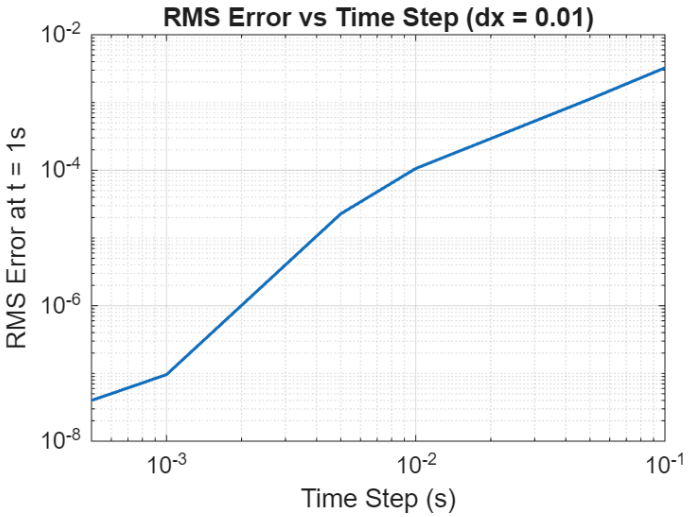

Figure 9: Comparison of RMS errors at $t = 1s$ for Varying Time Steps

2.5. Testing and Validation

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

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

3. Part 2: Software features

3.1. Error Evaluation

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

3.2. Integration Methods

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

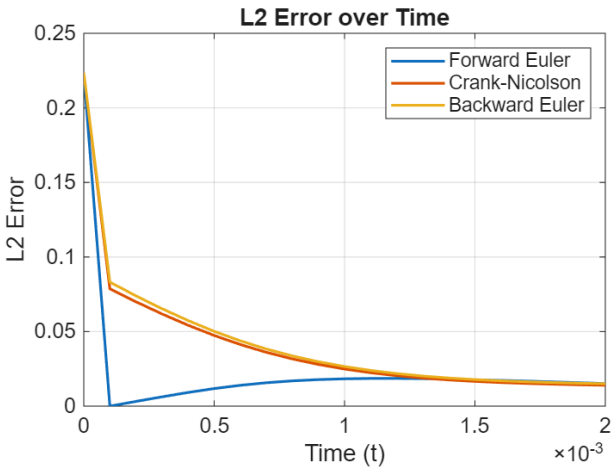

Figure 10: Comparison of L2 Errors for Different Time Integration Methods

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]:

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

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

3.3. Gaussian Quadrature

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

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

3.4. Quadratic Basis Functions

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

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

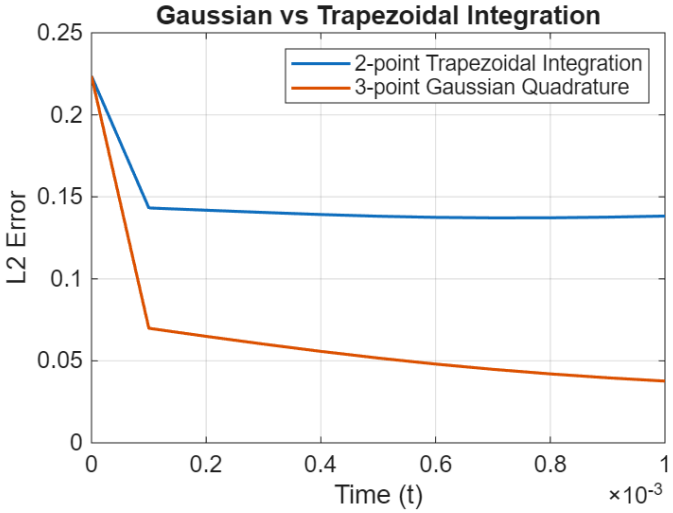

Figure 11: Comparison of L2 Errors for Gaussian Quadrature and Trapezoidal Integration

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

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:

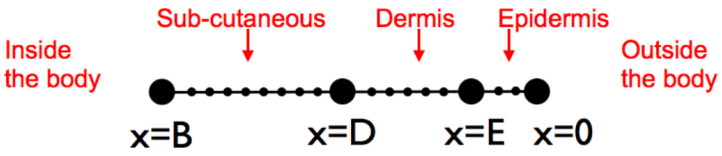

Figure 12: 1D Multilayer Finite Element Mesh of Skin Tissue Layers [8]

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

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

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

4.2. Solver Modification

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

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

density ratio, resulting in a local mesh with higher resolution, and improved solution accuracy.

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

4.3. Simulation Results

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

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

Table 3: Drug Concentration Problem Conditions

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

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

Table 4: Mesh Layer Parameters

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

The results were plotted as a heatmap (Figure 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.

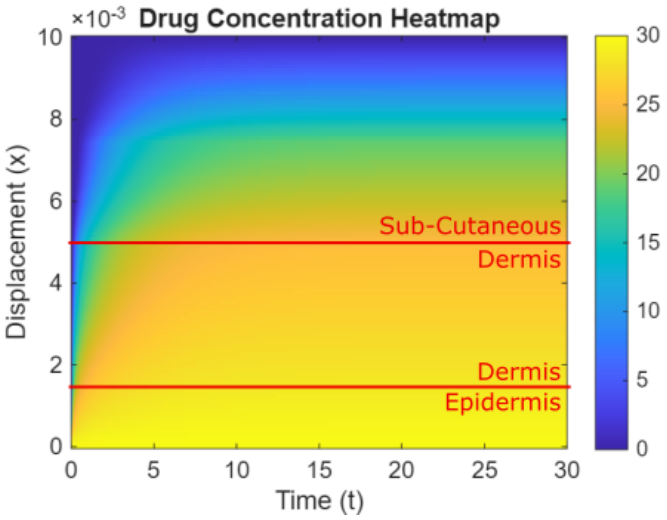

Figure 13: FEM Solution of Drug Diffusion through Multilayer Skin Structure over $0 \leq x \leq 0.01$ and $0 \leq t \leq 30s$

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

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File          :  main.m
% Author        :  samh25
% Created       :  2025-11-24 (YYYY-MM-DD)
% License       :  MIT
% Description   :  Main function for solving transient diffusion equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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

    Coursework.Part3InitialResults();

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

```

10. Coursework

10.1. Coursework.m

```m
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File          :  Coursework.m
% Author        :  samh25
% Created       :  2025-11-27 (YYYY-MM-DD)
% License       :  MIT
% Description   :  Static methods for each part of the coursework.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef Coursework

    methods (Static)

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

            % time parameters
            tmax = 1.0;
            dt = 0.01;

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

            % Crank-Nicholson method
            theta = 0.5;

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

            % concentrations
            c_max = 1.0;
            c_min = 0.0;


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

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

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

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

```
            lambda = 0.0;

            % concentrations
            c_max = 1.0;
            c_min = 0.0;


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

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

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

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

            % calculate RMS error with varying mesh sizes and time steps

            element_counts = [5, 10, 20, 25, 50];
            time_steps = [0.1, 0.05, 0.01, 0.005, 0.001, 0.0005];

            num_cases = length(element_counts) * length(time_steps);
            rms_errr_table_elem_count = zeros(num_cases, 4);   % columns: elem_count, dt,
dx, RMS error
            rms_errr_table_time_step = zeros(num_cases, 4);   % columns: elem_count, dt,
dx, RMS error

            k = 1;

            % vary element count with fixed time step
            for i = 1:length(element_counts)
                elem_count = element_counts(i);
                dt = 0.0005;

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

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

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

                % compute RMS error
                [~, final_time] = min(abs(analytical_solution.time - tmax));

                c_numeric = numeric_solution.values(:, final_time);
                c_analytical = analytical_solution.values(:, final_time);

                error = c_numeric - c_analytical;
                rms_error = sqrt(mean(error.^2));

                rms_errr_table_elem_count(k,:) = [elem_count, dt, (xmax-xmin)/elem_count,
rms_error];
                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,
    integration_method);
209 
210                 % solve analytically
211                 analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
212 
213                 % compute RMS error
214                 [~, final_time] = min(abs(analytical_solution.time - tmax));
215 
216                 c_numeric = numeric_solution.values(:, final_time);
217                 c_analytical = analytical_solution.values(:, final_time);
218 
219                 error = c_numeric - c_analytical;
220                 rms_error = sqrt(mean(error.^2));
221 
222                 rms_errr_table_time_step(k,:) = [elem_count, dt, (xmax-xmin)/elem_count,
    rms_error];
223                 k = k + 1;
224 
225                 fprintf("Elements: %d, dt: %.4f, dx: %.4f, RMS Error: %.6f\n", ...
226                     elem_count, dt, (xmax-xmin)/elem_count, rms_error);
227 
228             end
229 
230             % plot element counts
231             dx = rms_errr_table_elem_count(:, 3);  % element size
232             err_spatial = rms_errr_table_elem_count(:, 4);
233 
234             Plotter.PlotConvergenceError(dx, err_spatial, ...
235                 "RMS Error vs Element Size (dt = 0.0005)", ...
236                 "cw2/report/resources/part1/ElementSizeConvergence", "Element Size (x)",
    "RMS Error at t = 1s");
237 
238             % plot time steps
239             dt_vals = rms_errr_table_time_step(:, 2);  % time steps
240             err_temporal = rms_errr_table_time_step(:, 4);
241 
242             Plotter.PlotConvergenceError(dt_vals, err_temporal, ...
243                 "RMS Error vs Time Step (dx = 0.01)", ...
244                 "cw2/report/resources/part1/TimeStepConvergence", "Time Step (s)", "RMS
    Error at t = 1s");
245         end
246 
247         function Part2TimeIntegrationComparison()
248         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
249         %
250         % Function:     Part2TimeIntegrationComparison()
251         %
252         % 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
299             l2_errors = [];
300
301             thetas = [0.0, 1.0, 0.5]; % Explicit Euler, Implicit Euler, Crank-Nicholson
302             method_names = {"Forward Euler", "Crank-Nicolson", "Backward Euler"};
303
304             for i = 1:length(thetas)
305                 theta = thetas(i);
306
307                 % solve numerically
308                 numeric_solution = NumericSolver.SolveNumeric(...
309                     mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
    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,
    integration_method);
346
347                             % CHECK FOR NaN/Inf at each timestep
348                         for t_idx = 1:length(numeric_solution.time)
349                             vals = numeric_solution.values(:, t_idx);
350                             if any(isnan(vals))
351                                 fprintf("%s: NaN at step %d (t=%.4f)\n", method_names{i},
    t_idx, numeric_solution.time(t_idx));
352                                 break;
353                             end
354                             if any(isinf(vals))
355                                 fprintf("%s: Inf at step %d (t=%.4f)\n", method_names{i},
    t_idx, numeric_solution.time(t_idx));
356                                 break;
357                             end
358
359                             if max(abs(vals)) > 1e10
360                                 fprintf("%s: Explosion at step %d (t=%.4f), max=%.2e\n",
    method_names{i}, t_idx, numeric_solution.time(t_idx), max(abs(vals)));
361                                 break;
362                             end
363                         end
364
365                         analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax,
    dt);
366
367                         l2_error = L2Error(analytical_solution, numeric_solution);
368
369                         l2_errors_dt = [l2_errors_dt, l2_error];
370                     catch
371                         l2_errors_dt = [l2_errors_dt, NaN];
372                         fprintf("%s EXPLODED \n", method_names{i});
373                     end
374                 end
375
376                 l2_errors_stability = [l2_errors_stability; l2_errors_dt];
377             end
378
379         end
380
```

```
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,
    trapezoidal_method);
439 
440             % gaussian quadrature method
441             gaussian_method = IntegrationMethod();
442             gaussian_method.type = IntegrationType.Gaussian;
443             gaussian_method.gauss_points = 3; % 3-point Gaussian quadrature
444 
445             gaussian_solution = NumericSolver.SolveNumeric(...
```

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

```
511             numeric_solution = NumericSolver.SolveNumeric(...
512                 mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
    integration_method);
513 
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 
523     end
524 
525 end
526 
```

11. Mesh

11.1. Mesh.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  Mesh.m
% Author       :  samh25
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a one-dimensional mesh for
%                 finite element analysis.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


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

    properties

        xmin double
        xmax double
        dx double

        order double

        D double % diffusion coefficient
        lambda double % reaction coefficient

        node_count uint64
        node_coords double % coordinates of global nodes

        element_count uint64
        elements MeshElement % array of mesh elements

    end

    methods

        %% Mesh constructor
        function obj = Mesh(xmin, xmax, element_count, order, D, lambda)

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

            obj.order = order;

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

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

        end

        function obj = Generate(obj)

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

            % generate uniform node coordinates
```

```
            obj.node_coords = linspace(obj.xmin, obj.xmax, obj.node_count);

            % generate elements
            for e = 1:obj.element_count

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

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

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

    end
end

```

11.2. MeshElement.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  MeshElement.m
% Author       :  samh25
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a one-dimensional mesh element
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef MeshElement

    properties

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

    methods

        %% MeshElement constructor
        function obj = MeshElement(ids, coords, order, D, lambda)

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

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

        end
```

```
    end
end


```

11.3. MultilayerMesh.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  Mesh.m
% Author       :  samh25
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a one-dimensional mesh for
%                 finite element analysis.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


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

    properties
        layers    MeshLayer % array of layer properties
        total_density       double
    end

    methods

        %% Mesh constructor
        function obj = MultilayerMesh(xmin, xmax, element_count, order, D, lambda, layers)

            obj = obj@Mesh(xmin, xmax, element_count, order, D, lambda);
            obj.layers = layers;


            % recalculate nodes and element counts based on layer densities
            obj.total_density = 0.0;

            for l = 1:length(layers)
                obj.total_density = obj.total_density + layers(l).density_ratio;
            end

            obj.element_count = 0;

            for l = 1:length(layers)

                layer_density = obj.layers(l).density_ratio;
                layer_element_count = round((layer_density / obj.total_density) *
element_count);

                obj.layers(l).element_count = layer_element_count;
                obj.layers(l).layer_offset = obj.element_count + 1; % starting index for
this layer

                obj.element_count = obj.element_count + layer_element_count;

                disp(['Layer ' num2str(l) ' Element Count: '
num2str(layer_element_count)]);
                disp(obj.layers(l));
            end

            obj.node_count = (obj.element_count * order) + 1;
```

```
            obj.node_coords = zeros(1, obj.node_count);

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

        end

        function obj = Generate(obj)

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

            % generate per-layer uniform node coordinates

            current_node = 1;

            for l = 1:length(obj.layers)

                % calculate layer xmin and xmax

                layer_xmin = obj.layers(l).x;

                if l < length(obj.layers)
                    layer_xmax = obj.layers(l + 1).x;
                else
                    layer_xmax = obj.xmax;
                end

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

                layer_coords = linspace(layer_xmin, layer_xmax, layer_nodes);


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

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


            end

            % Generate elements
            for e = 1:obj.element_count

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

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

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

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

                D = obj.layers(layer_index).D;
```

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

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

    end
end


```

11.4. LayerProperties.m

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

    methods
        function obj = LayerProperties(x, D, beta, gamma, density_ratio)
            obj.x = x;
            obj.D = D;
            obj.beta = beta;
            obj.gamma = gamma;
            obj.density_ratio = density_ratio;
        end
    end
end
```

12. Analytical

12.1. AnalyticalSolver.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  AnalyticalSolver.m
% Author       :  samh25
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A static class defining an analytical solver
%                 for the transient diffusion equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef AnalyticalSolver

    methods (Static)

        function solution = SolveAnalytical(mesh, tmax, dt)

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

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

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

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

                solution.SetValues(timestep_results, step);
            end

        end
    end
end
```

12.2. TransientAnalyticSoln.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  TransientAnalyticSoln.m
% Author       :  A. N. Cookson
% Created      :  2025-11-11 (YYYY-MM-DD)
% License      :  -
% Description  :  Analytical solution to transient diffusion equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [ c ] = TransientAnalyticSoln(x,t)
%TransientAnalyticSonl Analytical solution to transient diffusion equation
%   Computes the analytical solution to the transient diffusion equation for
%   the domain x=[0,1], subject to initial condition: c(x,0) = 0, and Dirichlet
%   boundary conditions: c(0,t) = 0, and c(1,t) = 1.
%   Input Arguments:
```

```
%   x is the point in space to evaluate the solution at
%   t is the point in time to evaluate the solution at
%   Output Argument:
%   c is the value of concentration at point x and time t, i.e. c(x,t)

trans = 0.0;

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

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

end
```

13. Plotter

13.1. Plotter.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  Plotter.m
% Author       :  samh25
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A collection of static methods for plotting
%                 results for the coursework.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef Plotter

    methods (Static)

        %% Plot entire solution as a heatmap - time as x-axis, position as y-axis and solution value as color
        function PlotHeatMap(solution, title_str, name, c_max)

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

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

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

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

        end

        %% Plot full solution at specified time samples
        function PlotTimeSamples(solution, dt, time_samples, title_str, name)

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

            figure;
            plot_handle = 0;

            for i = 1:length(time_samples)
                t_sample = time_samples(i);

                step_index = round(t_sample / dt) + 1; % +1 for MATLAB indexing

                plot_handle = plot(solution.mesh.node_coords, solution.values(:, step_index));
                set(plot_handle, "LineWidth", 1.5);

                hold on;
```

```
            end

            xlabel("Position (x)");
            ylabel("c(x, t)");
            title(title_str);

            grid on;

            legend_strings = cell(1, length(time_samples));
            for i = 1:length(time_samples)
                legend_strings{i} = ['t = ', num2str(time_samples(i))];
            end

            legend(legend_strings, "Location", "northwest");

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

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

        end

        %% Plot two solutions at a specific position over time
        function PlotSampleOverTime(solution_1, solution_2, x_sample, title_str, name,
legend_strings)

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

            % find x index
            x_index = round((x_sample - solution_1.mesh.xmin) / (solution_1.mesh.xmax -
solution_1.mesh.xmin) * solution_1.mesh.element_count) + 1; % +1 for MATLAB indexing

            figure;
            plot_handle = plot(solution_1.time, solution_1.values(x_index, :));
            set(plot_handle, "LineWidth", 1.5);

            hold on;

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

            xlabel("Time (t)");

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

            grid on;

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

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

        end

        function PlotConvergenceError(x_values, y_values, title_str, name, x_label,
y_label)
            set(0, "DefaultAxesFontSize", 12);
            set(0, "DefaultTextFontSize", 12);

            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,
    x_label, y_label, legend_strings)
176             set(0, "DefaultAxesFontSize", 12);
177             set(0, "DefaultTextFontSize", 12);
178
179             figure;
180
181             loglog(x_values, y1_values, '-o', 'LineWidth', 1.5, 'MarkerSize', 8);
182             hold on;
183             loglog(x_values, y2_values, '-s', 'LineWidth', 1.5, 'MarkerSize', 8);
184
185             xlabel(x_label);
186             ylabel(y_label);
187             title(title_str);
188             legend(legend_strings, 'Location', 'best');
189             grid on;
190
191             set(gcf, 'Position', [0, 0, 500, 350]);
```

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

14. Solution

14.1. Solution.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  Solution.m
% Author       :  samh25
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a solution to the transient
%                 diffusion equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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

    properties

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

    end

    methods

        %% Solution constructor
        function obj = Solution(mesh, time_vector)

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

        end

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


```

14.2. L2Error.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  L2Error.m
% Author       :  samh25
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A class defining a solution to the transient
%                 diffusion equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

classdef L2Error < handle
```

```
    % inherit from handle to allow pass-by-reference behaviour

    properties

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

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

    end

    methods

        %% Solution constructor
        function obj = L2Error(ref_solution, num_solution)

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

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

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

            step_count = length(ref_solution.time);

            obj.l2_error = zeros(1, step_count);

            for step = 1:step_count
                c_ref = ref_solution.values(:, step);
                c_num = num_solution.values(:, step);
                x = ref_solution.mesh.node_coords;

                integrand = (c_ref - c_num).^2;
                obj.l2_error(step) = sqrt(trapz(x, integrand));
            end

        end

    end
end


```

14.3. DoseEvaluator.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  Evaluation.m
% Author       :  samh25
% Created      :  2025-11-26 (YYYY-MM-DD)
% License      :  MIT
% Description  :  A static class defining a dose evaluator of
%                 a solution
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
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

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File         :  NumericSolver.m
% Author       :  samh25
% Created      :  2025-11-24 (YYYY-MM-DD)
% License      :  MIT
% Description  :  Class definition for generic solver for the
%                 transient diffusion-reaction equation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


classdef NumericSolver

    methods (Static)

        function solution = SolveNumeric(mesh, tmax, dt, theta, left_boundary, right_boundary, source_fn, integration_method)

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

            % === SET INITIAL CONDITION EXPLICITLY ===
            c0 = zeros(mesh.node_count, 1);
            solution.SetValues(c0, 1);  % column 1 = t=0

            [K, M] = NumericSolver.CreateGlobalMatrices(mesh, theta, integration_method);

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

                c_next = NumericSolver.SolveStep(mesh, solution, step, dt, theta, K, M, left_boundary, right_boundary, source_fn, integration_method);
                solution.SetValues(c_next, step + 1);

            end

        end

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

            c_current = solution.values(:, step);

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

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

            % add source term
            f_current = NumericSolver.CreateSourceVector(mesh, t, source_fn, integration_method);
            f_next = NumericSolver.CreateSourceVector(mesh, t + dt, source_fn, integration_method);
            rhs_vector = rhs_vector + dt * (theta * f_next + (1 - theta) * f_current);

            % apply boundary conditions
```

```
57            [system_matrix, rhs_vector] =
   NumericSolver.ApplyBoundaryConditions(system_matrix, rhs_vector, t + dt, left_boundary,
   right_boundary);
58
59            % solve system
60            c = system_matrix \ rhs_vector;
61
62        end
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,
   integration_method);
81                react_matrix = ElementMatrices.ReactionElemMatrix(element,
   integration_method);
82
83                k_matrix = diff_matrix - react_matrix;
84
85                elem_size = element.node_coords(end) - element.node_coords(1);
86                m_matrix = ElementMatrices.MassElemMatrix(element, integration_method);
87
88                % assemble into global matrices
89                for i = 1:local_size
90                    for j = 1:local_size
91                        gi = nodes(i); gj = nodes(j);
92                        K(gi, gj) = K(gi, gj) + k_matrix(i, j);
93                        M(gi, gj) = M(gi, gj) + m_matrix(i, j);
94                    end
95                end
96
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                 % 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,
    right_boundary)
130 
131              % Store diagonal values before modification
132             diag_left = lhs(1,1);
133             diag_right = lhs(end,end);
134 
135             % apply left boundary condition
136             switch left_boundary.Type
137 
138                 case BoundaryType.Dirichlet
139                     lhs(1, :) = 0;                  % clear row
140                     lhs(1, 1) = diag_left;          % keep diagonal
141                     rhs(1) = left_boundary.Value * diag_left;   % scale by diagonal
142 
143                 case BoundaryType.Neumann
144                     rhs(1) = rhs(1) + left_boundary.ValueFunction(t); % apply flux
145 
146             end
147 
148 
149 
150             % apply right boundary condition
151             switch right_boundary.Type
152                  case BoundaryType.Dirichlet
153                     lhs(end, :) = 0;                % clear row
154                     lhs(end, end) = diag_right;     % set diagonal to 1
155                     rhs(end) = right_boundary.Value * diag_right; % set value
156 
157                 case BoundaryType.Neumann
158                     rhs(end) = rhs(end) + right_boundary.ValueFunction(t); % apply flux
159             end
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
  t, return double
6     end
7 end
```

15.3. BoundaryType.m

```
1 classdef BoundaryType
2    enumeration
3         Dirichlet, Neumann
```

```
    end
end
```

15.4. ElementMatrices.m

```
classdef ElementMatrices

    methods (Static)

        function matrix = DiffusionElemMatrix(element, method)

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

            if method.type == IntegrationType.Trapezoidal

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

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

                % apply matrix scaling
                matrix = matrix * (element.D / elem_size);

            else

                matrix = zeros(element.order + 1);
                [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);

                for i = 1:length(xi)
                    dN_dxi = ElementMatrices.ShapeFunctionDerivatives(element.order,
xi(i));

                    J = element.jacobian;
                    dN_dx = dN_dxi / J;

                    % compute contribution to stiffness matrix
                    matrix = matrix + (element.D * (dN_dx' * dN_dx)) * (wi(i) * J);
                end

            end

        end

        function matrix = ReactionElemMatrix(element, method)

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

            if method.type == IntegrationType.Trapezoidal

                % create base matrix
                matrix = eye(element.order + 1) * 2;

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

```
60
61                  % apply matrix scaling
62                  matrix = matrix * (element.lambda * elem_size / 6);
63
64              else
65
66                  matrix = zeros(element.order + 1);
67                  [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
68
69                  for i = 1:length(xi)
70                      N = ElementMatrices.ShapeFunctions(element.order, xi(i));
71
72                      J = element.jacobian;
73
74                      % compute contribution to stiffness matrix
75                      matrix = matrix + (element.lambda * (N' * N)) * (wi(i) * J);
76                  end
77              end
78          end
79
80          function matrix = MassElemMatrix(element, method)
81
82              elem_size = element.node_coords(end) - element.node_coords(1);
83
84              if method.type == IntegrationType.Trapezoidal
85
86                  % create base matrix
87                  matrix = eye(element.order + 1) * 2;
88
89                  for i = 1:(element.order + 1)
90                      for j = 1:(element.order + 1)
91                          if i ~= j
92                              matrix(i, j) = 1;
93                          end
94                      end
95                  end
96
97                  % apply matrix scaling
98                  matrix = matrix * (elem_size / 6);
99
100             else
101
102                 matrix = zeros(element.order + 1);
103                 [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);
104
105                 for i = 1:length(xi)
106                     N = ElementMatrices.ShapeFunctions(element.order, xi(i));
107
108                     J = element.jacobian;
109
110                     % compute contribution to stiffness matrix
111                     matrix = matrix + (N' * N) * (wi(i) * J);
112                 end
113
114             end
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);
126
```

```
                % apply matrix scaling
                matrix = matrix * (elem_size / 2);

            else

                matrix = zeros(element.order + 1, 1);
                [xi, wi] = ElementMatrices.GaussQuadraturePoints(method.gauss_points);

                for i = 1:length(xi)
                    N = ElementMatrices.ShapeFunctions(element.order, xi(i));

                    J = element.jacobian;

                    % compute contribution to stiffness matrix
                    matrix = matrix + N' * (wi(i) * J);
                end

            end

        end
    end

    methods (Static, Access = private)

        function [xi, wi] = GaussQuadraturePoints(n)

            switch n
                case 1
                    xi = 0;
                    wi = 2;
                case 2
                    xi = [-1/sqrt(3), 1/sqrt(3)];
                    wi = [1, 1];
                case 3
                    xi = [-sqrt(3/5), 0, sqrt(3/5)];
                    wi = [5/9, 8/9, 5/9];
                otherwise
                    error('Gauss quadrature for n > 3 not implemented.');
            end

        end

        function N = ShapeFunctions(order, xi)

            switch order
                case 1 % linear
                    N = [(1 - xi) / 2, (1 + xi) / 2];
                case 2 % quadratic
                    N = [xi * (xi - 1) / 2, (1 - xi^2), xi * (xi + 1) / 2];
                otherwise
                    error('Shape functions for order > 2 not implemented.');
            end

        end

        function dN_dxi = ShapeFunctionDerivatives(order, xi)

            switch order
                case 1 % linear
                    dN_dxi = [-0.5, 0.5];
                case 2 % quadratic
                    dN_dxi = [xi - 0.5, -2 * xi, xi + 0.5];
                otherwise
                    error('Shape function derivatives for order > 2 not implemented.');
            end

        end
```

```
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

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ME40064 Coursework 2
%
% File          :  NumericSolverTest.m
% Author        :  samh25
% Created       :  2025-11-27 (YYYY-MM-DD)
% License       :  MIT
% Description   :  Test suite for NumericSolver class
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function tests = NumericSolverTest
    tests = functiontests(localfunctions);
end

function TestSolveNumericReactionOnly(testCase)

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

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

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

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

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

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

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

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

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

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

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