ME40064 System Modelling and Simulation - Coursework 2

1252 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).

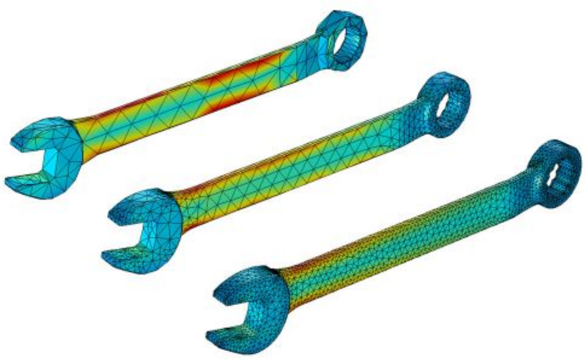

Figure 1: Finite Element Modelling of a Wrench under a Test Load Scenario [2]

This coursework focuses on the implementation and verification of a FEM solver for the transient diffusion-reaction equation, given by [3]:

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

Where:
- c is the concentration level
- D is the diffusion coefficient
- λ is the reaction rate
- f is a source term

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).

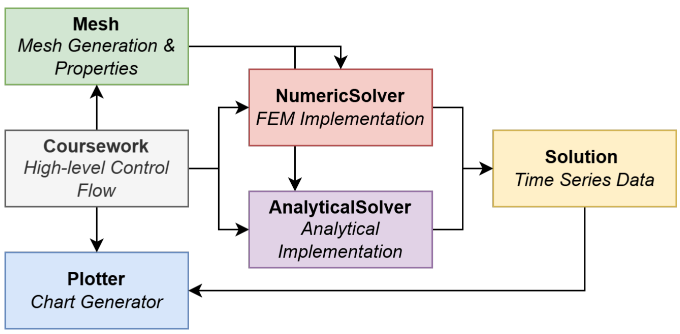

Figure 2: High-Level Software Architecture of the FEM Solver

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).

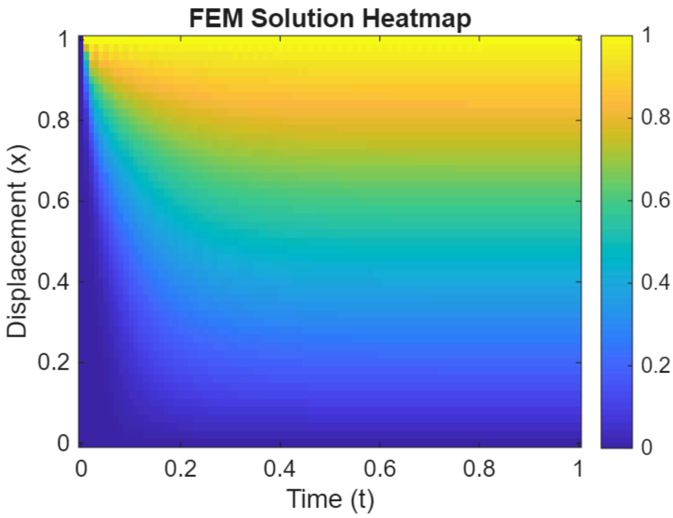

Figure 3: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

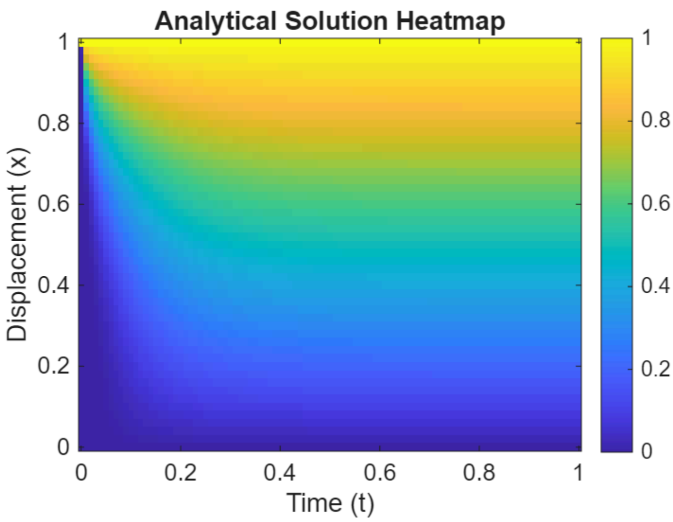

Figure 4: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

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.

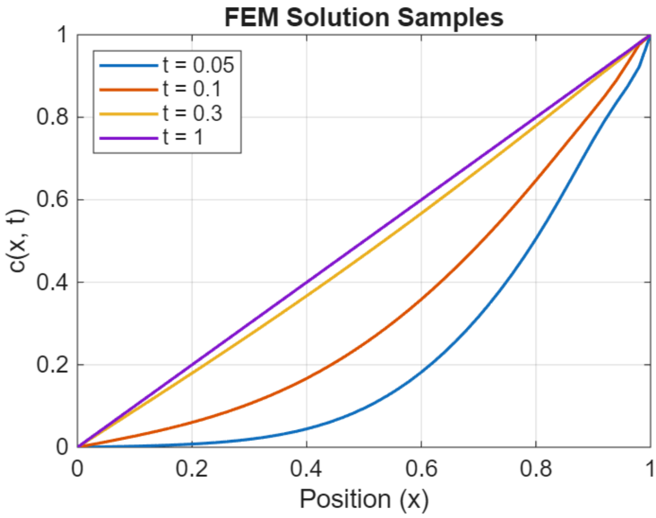

Figure 5: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

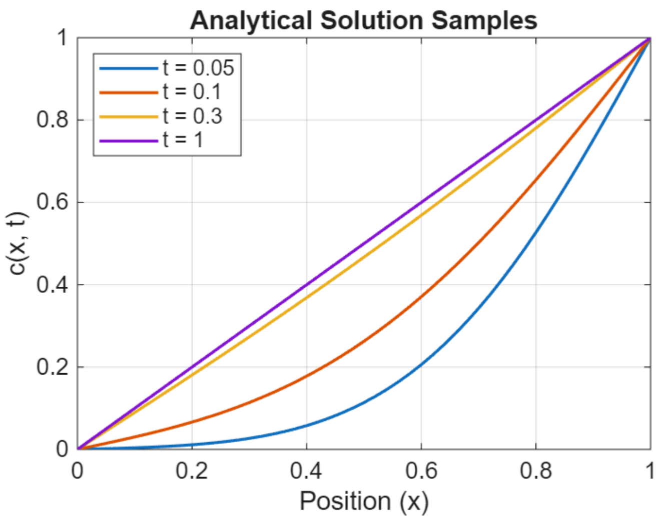

Figure 6: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

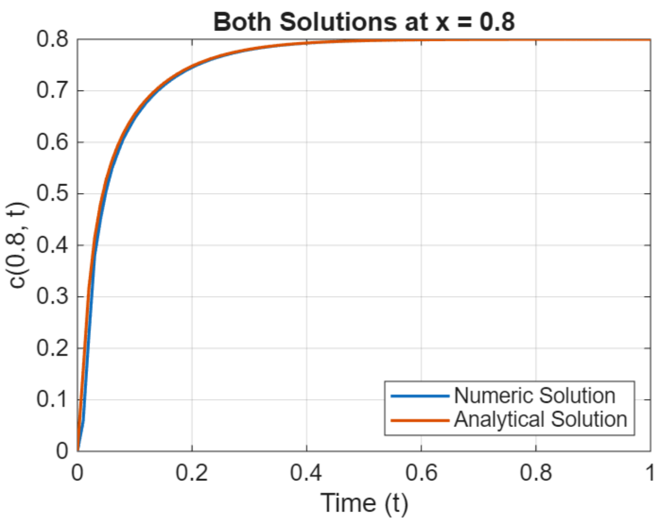

Figure 7: Comparison of Analytical and FEM Solutions at $x = 0.8$ over $0 \leq t \leq 1s$

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.

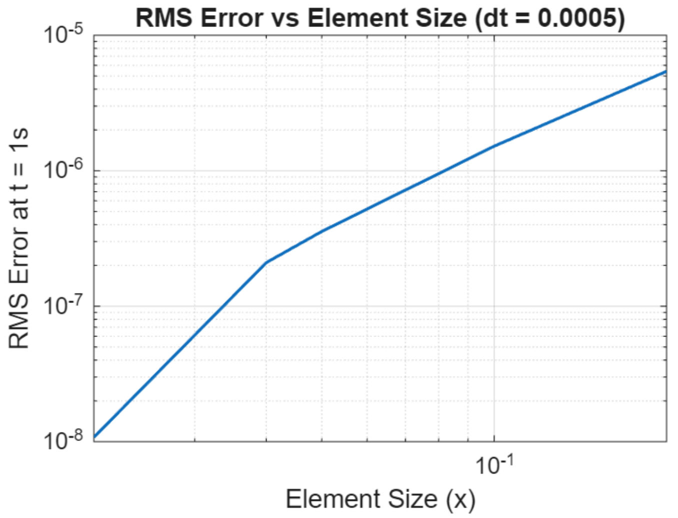

Figure 8: Comparison of RMS errors at $t = 1s$ for Varying Element Sizes

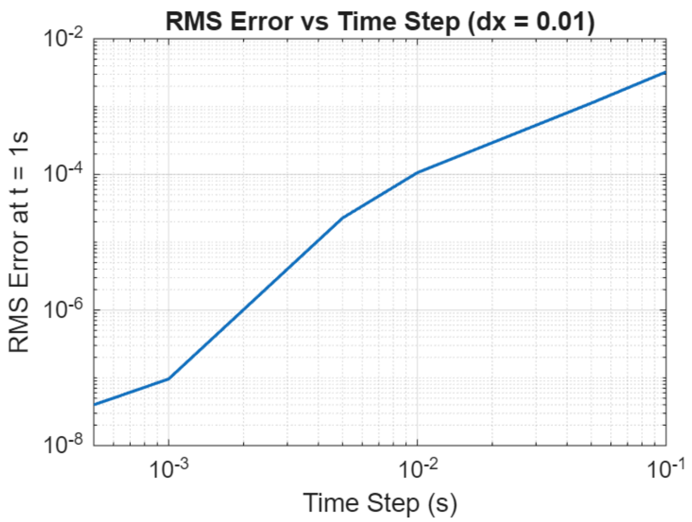

Figure 9: Comparison of RMS errors at $t = 1s$ for Varying Time Steps

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 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.

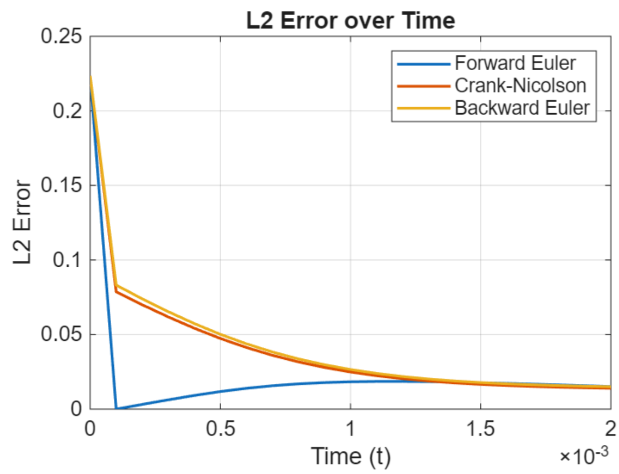

Figure 10: Comparison of L2 Errors for Different Time Integration Methods

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]:

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

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:

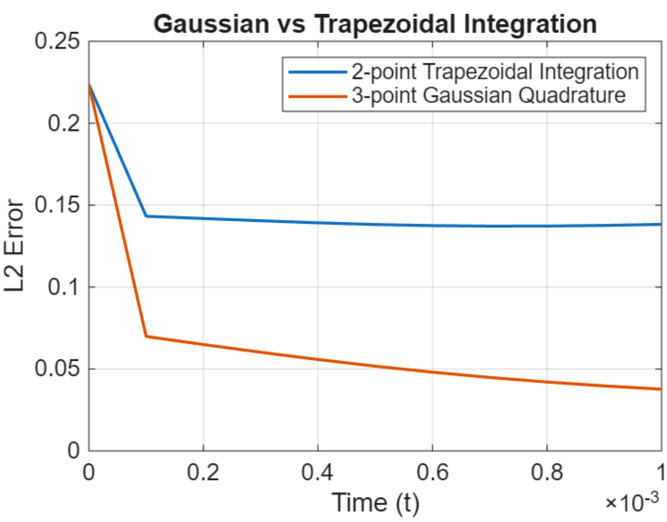

Figure 11: Comparison of L2 Errors for Gaussian Quadrature and Trapezoidal Integration

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. Mesh Refining

NEED TO REFINE MESH!

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

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

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 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.Part2GaussianQuadrature();

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

```

10. Coursework

10.1. Coursework.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;
```

```
            rhs_boundary.Value = c_max;

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

            % 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);

            % plot solutions as a heatmaps
            Plotter.PlotHeatMap(numeric_solution, "FEM Solution Heatmap", ...
                "cw2/report/resources/part1/NumericHeatmap", c_max);
            Plotter.PlotHeatMap(analytical_solution, "Analytical Solution Heatmap", ...
                "cw2/report/resources/part1/AnalyticalHeatmap", c_max);

            % plot solution samples at specified times
            sample_times = [0.05, 0.1, 0.3, 1.0];
            Plotter.PlotTimeSamples(numeric_solution, dt, sample_times, "FEM Solution Samples", ...
                "cw2/report/resources/part1/NumericSamples");
            Plotter.PlotTimeSamples(analytical_solution, dt, sample_times, "Analytical Solution Samples", ...
                "cw2/report/resources/part1/AnalyticalSamples");

            % plot both solutions at a specific position over time
            sample_x = 0.8;
            legend_strings = {"Numeric Solution", "Analytical Solution"};
            Plotter.PlotSampleOverTime(numeric_solution, analytical_solution, ...
                sample_x, "Both Solutions at x = 0.8", "cw2/report/resources/part1/BothX08", legend_strings);

        end

        function Part1Convergence()
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     Part1Convergence()
        %
        % Arguments:    None
        % Returns:      None
        %
        % Description:  Runs a convergence study for Part 1 of the
        %               coursework, calculating RMS error between numeric
        %               and analytical solutions over a range of element
        %               counts and time steps.
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % time parameters
            tmax = 1.0;

            % 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;
            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;

```

```
                fprintf("Elements: %d, dt: %.4f, dx: %.4f, RMS Error: %.6f\n", ...
                    elem_count, dt, (xmax-xmin)/elem_count, rms_error);
            end

            k = 1;

            % vary time step with fixed element count
            for j = 1:length(time_steps)

                elem_count = 1 / 0.01;
                dt = time_steps(j);

                % 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_time_step(k,:) = [elem_count, dt, (xmax-xmin)/elem_count, rms_error];
                k = k + 1;

                fprintf("Elements: %d, dt: %.4f, dx: %.4f, RMS Error: %.6f\n", ...
                    elem_count, dt, (xmax-xmin)/elem_count, rms_error);

            end

            % plot element counts
            dx = rms_errr_table_elem_count(:, 3);  % element size
            err_spatial = rms_errr_table_elem_count(:, 4);

            Plotter.PlotConvergenceError(dx, err_spatial, ...
                "RMS Error vs Element Size (dt = 0.0005)", ...
                "cw2/report/resources/part1/ElementSizeConvergence", "Element Size (x)", "RMS Error at t = 1s");

            % plot time steps
            dt_vals = rms_errr_table_time_step(:, 2);  % time steps
            err_temporal = rms_errr_table_time_step(:, 4);

            Plotter.PlotConvergenceError(dt_vals, err_temporal, ...
                "RMS Error vs Time Step (dx = 0.01)", ...
                "cw2/report/resources/part1/TimeStepConvergence", "Time Step (s)", "RMS Error at t = 1s");
        end

        function Part2TimeIntegrationComparison()
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     Part2TimeIntegrationComparison()
        %
        % Arguments:    None
```

```
        % Returns:      None
        %
        % Description:  Runs a study comparing different time integration
        %               methods for Part 2 of the coursework.
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            % time parameters
            tmax = 0.002;
            dt = 0.0001;

            % mesh parameters
            xmin = 0.0;
            xmax = 1.0;
            element_count = 10;
            order = 1;

            % 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();

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

            % 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


            l2_errors = [];

            thetas = [0.0, 1.0, 0.5]; % Explicit Euler, Implicit Euler, Crank-Nicholson
            method_names = {"Forward Euler", "Crank-Nicolson", "Backward Euler"};

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

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

                % compute L2 error
                l2_error = L2Error(analytical_solution, numeric_solution);
                l2_errors = [l2_errors, l2_error];
            end

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

```
            % perform stability analysis

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

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

            l2_errors_stability = [];

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

                l2_errors_dt = [];

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

                    try

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

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

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

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

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

                        l2_error = L2Error(analytical_solution, numeric_solution);

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

                l2_errors_stability = [l2_errors_stability; l2_errors_dt];
            end

        end
```

```
        function Part2GaussianQuadrature()
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     Part2GaussianQuadrature()
        %
        % Arguments:    None
        % Returns:      None
        %
        % Description:  Runs a study comparing L2 error with and without
        %               Gaussian Quadrature.
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


            % time parameters
            tmax = 0.001;
            dt = 0.0001;

            % mesh parameters
            xmin = 0.0;
            xmax = 1.0;
            element_count = 5;
            order = 2;

            % diffusion and reaction coefficients
            D = 0.5;
            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();

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

            % solver parameters

            theta = 0.5; % Crank-Nicholson

            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;

            l2_errors = [];

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

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

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

```

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

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

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

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

        function Part2QuadraticBasisFunctions()
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        %
        % Function:     Part2TimeIntegrationComparison()
        %
        % Arguments:    None
        % Returns:      None
        %
        % Description:  Runs a study comparing different time integration
        %               methods for Part 2 of the coursework.
        %
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



        end

    end

end

```

11. Mesh

11.1. Mesh.m

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 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
        layer_properties LayerProperties % array of layer properties
    end

    methods

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

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

        end

        function obj = Generate(obj)

            disp('Generating multilayer 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);

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

                % determine which layer this element is in
                layer_index = 1;

                for l = 1:length(obj.layer_properties)
                    if midpoint >= obj.layer_properties(l).x
                        layer_index = l;
                    end
```

```
                end

                D = obj.layer_properties(layer_index).D;
                lambda = -(obj.layer_properties(layer_index).beta +
obj.layer_properties(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
        D       double  % diffusion coefficient
        beta    double  % extra-vascular diffusivity
        gamma   double  % drug degradation rate
    end

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

```

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

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

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

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

        end

        function PlotL2Errors(l2_errors, title_str, name, legend_strings)

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

            figure;

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

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

            grid on;

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

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

            openfig(name + ".fig");

        end

        function PlotTwoConvergenceLines(x_values, y1_values, y2_values, title_str, name, x_label, y_label, legend_strings)
            set(0, "DefaultAxesFontSize", 12);
            set(0, "DefaultTextFontSize", 12);

            figure;

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

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

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

```

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

    end
end
```

14. Solution

14.1. Solution.m

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 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

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 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
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```matlab

classdef DoseEvaluator

    methods (Static)

        function K = EvaluateSolution(solution, target_x, c_threshold, dt)

            % first, find the closest node to the target
            node_index = 0;

            for i = 1:solution.mesh.node_count
                x = solution.mesh.node_coords(i);
                if x >= target_x
                    node_index = i;
                    break;
                end
            end

            fprintf("node index %d\n", node_index);
            c = solution.values(node_index, :);

            effective_t_index = 0;

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

            fprintf("effective t index %d\n", effective_t_index);

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

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

```
            [system_matrix, rhs_vector] = NumericSolver.ApplyBoundaryConditions(system_matrix, rhs_vector, t + dt, left_boundary, right_boundary);

            % solve system
            c = system_matrix \ rhs_vector;

        end

        %% Create global stiffness and mass matrices
        function [K, M] = CreateGlobalMatrices(mesh, theta, integration_method)

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

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

            for element_id = 1:num_elements

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

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

                k_matrix = diff_matrix - react_matrix;

                elem_size = element.node_coords(end) - element.node_coords(1);
                m_matrix = ElementMatrices.MassElemMatrix(element, integration_method);

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

            end

        end

        %% Create source vector for given time
        function F = CreateSourceVector(mesh, t, source_fn, integration_method)

            F = zeros(mesh.node_count, 1);

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

            for element_id = 1:mesh.element_count

                element = mesh.elements(element_id);

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

                f_val = source_fn(midpoint, t);

```

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

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

        end

        function [lhs, rhs] = ApplyBoundaryConditions(lhs, rhs, t, left_boundary,
right_boundary)

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

            % apply left boundary condition
            switch left_boundary.Type

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

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

            end



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

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

        end

    end
end

```

15.2. BoundaryCondition.m

```
classdef BoundaryCondition
    properties
        Type BoundaryType % Boundary condition type (Dirichlet or Neumann)
        Value double % Boundary condition value for Dirichlet
        ValueFunction function_handle % Boundary condition function for Neumann - parameter
t, return double
    end
end
```

15.3. BoundaryType.m

```
classdef BoundaryType
   enumeration
        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
```

```

                % apply matrix scaling
                matrix = matrix * (element.lambda * elem_size / 6);

            else

                matrix = zeros(element.order + 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 + (element.lambda * (N' * N)) * (wi(i) * J);
                end
            end
        end

        function matrix = MassElemMatrix(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

                % apply matrix scaling
                matrix = matrix * (elem_size / 6);

            else

                matrix = zeros(element.order + 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' * N) * (wi(i) * J);
                end

            end

        end

        function matrix = ForceMatrix(element, method)

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

            if method.type == IntegrationType.Trapezoidal

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

```

```
                % 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
```

```
    end
end
```

15.5. IntegrationMethod.m

```
classdef IntegrationMethod
    properties
        type            IntegrationType
        gauss_points    uint8
    end
end
```

15.6. IntegrationType.m

```
classdef IntegrationType
   enumeration
        Trapezoidal, Gaussian
   end
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
```

```

function s = SourceFunction(x, t)
    s = 10;
end

```