

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).

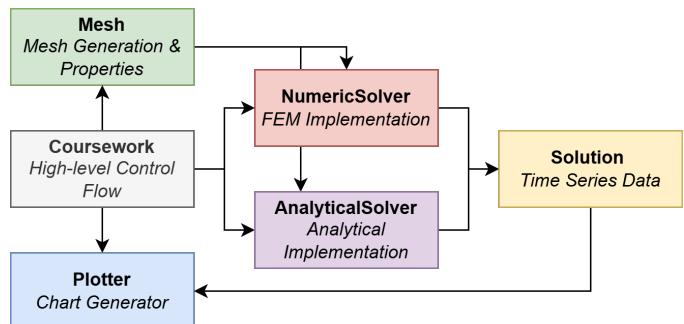


Figure 2: High-Level Software Architecture of the FEM Solver

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).

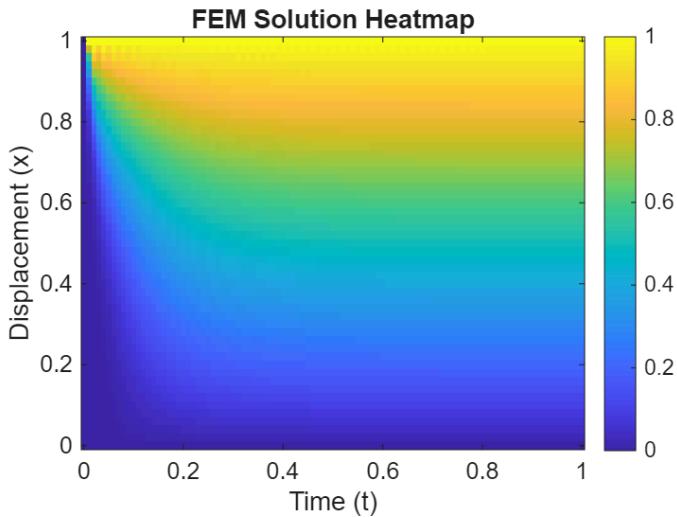


Figure 3: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

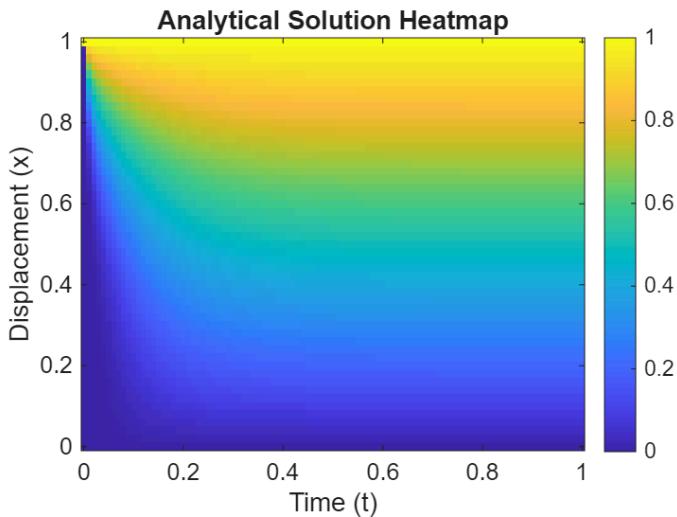


Figure 4: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and $0 \leq t \leq 1s$

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.

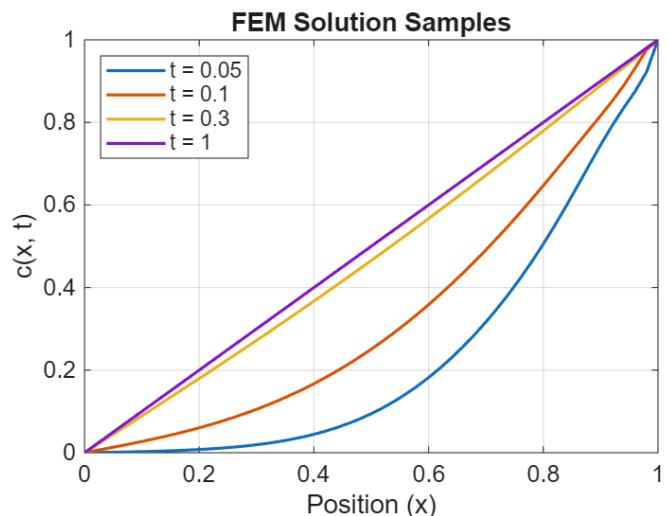


Figure 5: FEM Solution of Diffusion Equation over using the Crank-Nicolson method over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

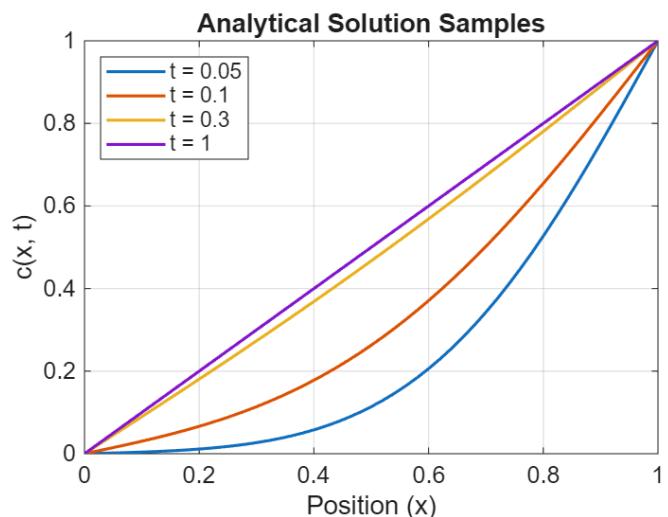


Figure 6: Analytical Solution of Diffusion Equation over $0 \leq x \leq 1$ and at $t = 0.05s, 0.1s, 0.3s, 1.0s$

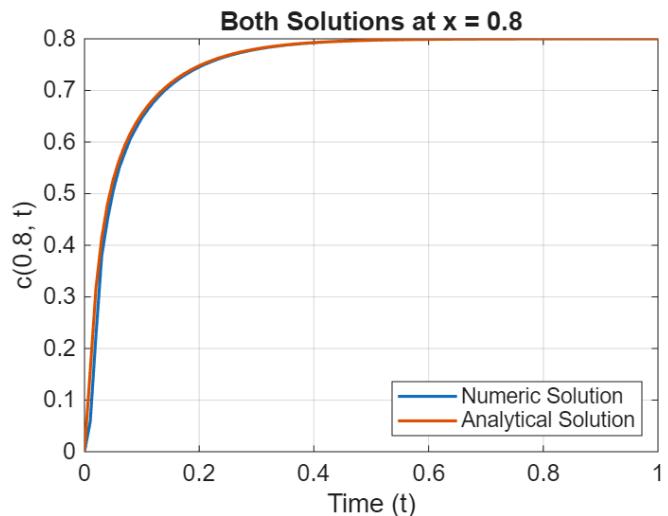


Figure 7: Comparison of Analytical and FEM Solutions at $x = 0.8$ over $0 \leq t \leq 1s$

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.

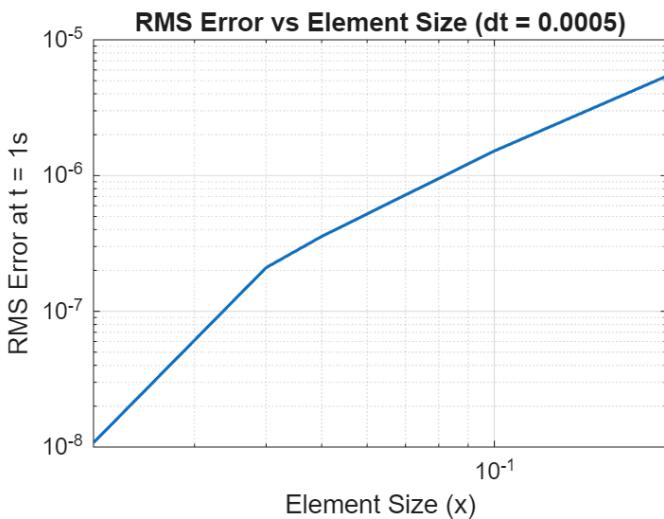


Figure 8: Comparison of RMS errors at $t = 1s$ for Varying Element Sizes

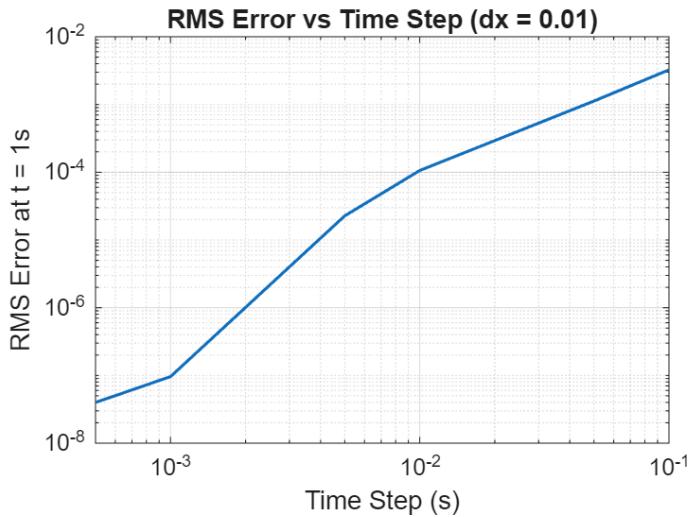


Figure 9: Comparison of RMS errors at $t = 1s$ for Varying Time Steps

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

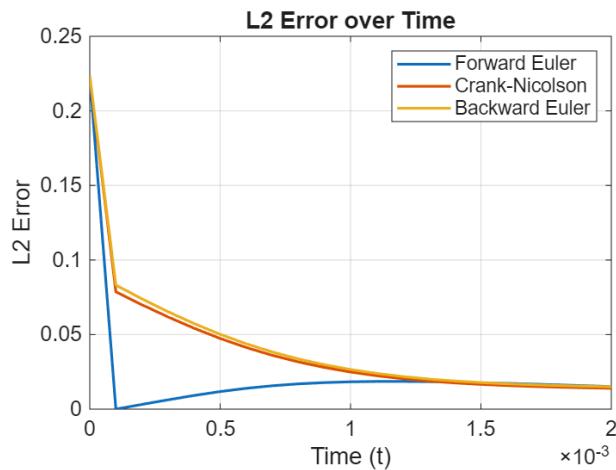


Figure 10: Comparison of L2 Errors for Different Time Integration Methods

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

To illustrate this further, a stability analysis was performed for all three methods, using a larger mesh of 50 elements:

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

Table 2: Integration Method Stability Comparison

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

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

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

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

3.3. Gaussian Quadrature

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

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

3.4. Quadratic Basis Functions

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

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

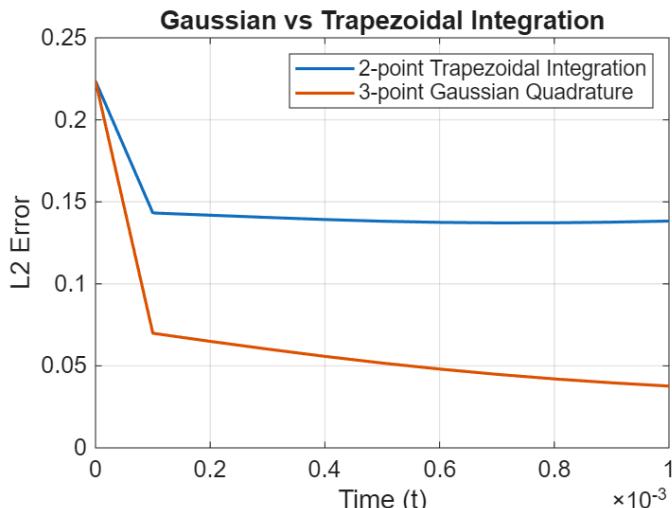


Figure 11: Comparison of L2 Errors for Gaussian Quadrature and Trapezoidal Integration

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

3.5. Summary of Features

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

4. Part 3: Modelling & Simulation Results

4.1. 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

```
1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : main.m
6 % Author    : samh25
7 % Created   : 2025-11-24 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : Main function for solving transient diffusion equation
10 %
11 %%%%%%
12
13 function main()
14     fprintf("ME40064 Coursework 2 Starting...\n");
15
16     Coursework.Part2GaussianQuadrature();
17
18     fprintf("...ME40064 Coursework 2 Complete\n");
19 end
20
```

10. Coursework

10.1. Coursework.m

```
1 %%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Coursework.m
6 % Author    : samh25
7 % Created   : 2025-11-27 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : Static methods for each part of the coursework.
10 %
11 %%%%%%%%%%%%%%
12
13 classdef Coursework
14
15     methods (Static)
16
17         function Part1Plots()
18             %%%%%%
19             %
20             % Function:      Part1Plots()
21             %
22             % Arguments:    None
23             % Returns:      None
24             %
25             % Description: Runs the start Part 1 of the coursework,
26             %                 generating a simple mesh, running numeric and
27             %                 analytical solvers, and plotting the results.
28             %
29 %%%%%%
30
31         % time parameters
32         tmax = 1.0;
33         dt = 0.01;
34
35         % mesh parameters
36         xmin = 0.0;
37         xmax = 1.0;
38         element_count = 50;
39         order = 1;
40
41         % Crank-Nicholson method
42         theta = 0.5;
43
44         % diffusion and reaction coefficients
45         D = 1.0;
46         lambda = 0.0;
47
48         % concentrations
49         c_max = 1.0;
50         c_min = 0.0;
51
52
53         % generate mesh
54         mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
55         mesh.Generate();
56
57         % solver parameters
58         lhs_boundary = BoundaryCondition();
59         lhs_boundary.Type = BoundaryType.Dirichlet;
60         lhs_boundary.Value = c_min;
61
62         rhs_boundary = BoundaryCondition();
63         rhs_boundary.Type = BoundaryType.Dirichlet;
```

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

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

```

190         fprintf("Elements: %d, dt: %.4f, dx: %.4f, RMS Error: %.6f\n", ...
191                 elem_count, dt, (xmax-xmin)/elem_count, rms_error);
192     end
193
194     k = 1;
195
196     % vary time step with fixed element count
197     for j = 1:length(time_steps)
198
199         elem_count = 1 / 0.01;
200         dt = time_steps(j);
201
202         % generate mesh
203         mesh = Mesh(xmin, xmax, elem_count, order, D, lambda);
204         mesh.Generate();
205
206         % solve numerically
207         numeric_solution = NumericSolver.SolveNumeric(... ...
208                         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
209             integration_method);
210
211         % solve analytically
212         analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
213
214         % compute RMS error
215         [~, final_time] = min(abs(analytical_solution.time - tmax));
216
217         c_numeric = numeric_solution.values(:, final_time);
218         c_analytical = analytical_solution.values(:, final_time);
219
220         error = c_numeric - c_analytical;
221         rms_error = sqrt(mean(error.^2));
222
223         rms_errr_table_time_step(k, :) = [elem_count, dt, (xmax-xmin)/elem_count,
224                                         rms_error];
225         k = k + 1;
226
227         fprintf("Elements: %d, dt: %.4f, dx: %.4f, RMS Error: %.6f\n", ...
228                 elem_count, dt, (xmax-xmin)/elem_count, rms_error);
229     end
230
231     % plot element counts
232     dx = rms_errr_table_elem_count(:, 3); % element size
233     err_spatial = rms_errr_table_elem_count(:, 4);
234
235     Plotter.PlotConvergenceError(dx, err_spatial, ...
236         "RMS Error vs Element Size (dt = 0.0005)", ...
237         "cw2/report/resources/part1/ElementSizeConvergence", "Element Size (x)",
238         "RMS Error at t = 1s");
239
240     % plot time steps
241     dt_vals = rms_errr_table_time_step(:, 2); % time steps
242     err_temporal = rms_errr_table_time_step(:, 4);
243
244     Plotter.PlotConvergenceError(dt_vals, err_temporal, ...
245         "RMS Error vs Time Step (dx = 0.01)", ...
246         "cw2/report/resources/part1/TimeStepConvergence", "Time Step (s)", "RMS
247     Error at t = 1s");
248
249     function Part2TimeIntegrationComparison()
250         % 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      l2_errors = [];
299
300      thetas = [0.0, 1.0, 0.5]; % Explicit Euler, Implicit Euler, Crank-Nicholson
301      method_names = {"Forward Euler", "Crank-Nicolson", "Backward Euler"};
302
303      for i = 1:length(thetas)
304          theta = thetas(i);
305
306              % solve numerically
307              numeric_solution = NumericSolver.SolveNumeric(
308                  mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
309                  integration_method);
310
311                  % compute L2 error
312                  l2_error = L2Error(analytical_solution, numeric_solution);
313                  l2_errors = [l2_errors, l2_error];
314      end
315
316      Plotter.PlotL2Errors(l2_errors, "L2 Error over Time", ...
317          "cw2/report/resources/part2/L2ErrorTimeIntegration", ...
318          method_names);
```

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

```
381     function Part2GaussianQuadrature()
382 %%%%%%
383 %
384 % Function:      Part2GaussianQuadrature()
385 %
386 % Arguments:    None
387 % Returns:      None
388 %
389 % Description:  Runs a study comparing L2 error with and without
390 %                 Gaussian Quadrature.
391 %
392 %%%%%%
393
394
395     % time parameters
396     tmax = 0.001;
397     dt = 0.0001;
398
399     % mesh parameters
400     xmin = 0.0;
401     xmax = 1.0;
402     element_count = 5;
403     order = 2;
404
405     % diffusion and reaction coefficients
406     D = 0.5;
407     lambda = 0.0;
408
409     % concentrations
410     c_max = 1.0;
411     c_min = 0.0;
412
413     % generate mesh
414     mesh = Mesh(xmin, xmax, element_count, order, D, lambda);
415     mesh.Generate();
416
417     % solve analytically
418     analytical_solution = AnalyticalSolver.SolveAnalytical(mesh, tmax, dt);
419
420     % solver parameters
421
422     theta = 0.5; % Crank-Nicholson
423
424     lhs_boundary = BoundaryCondition();
425     lhs_boundary.Type = BoundaryType.Dirichlet;
426     lhs_boundary.Value = c_min;
427
428     rhs_boundary = BoundaryCondition();
429     rhs_boundary.Type = BoundaryType.Dirichlet;
430     rhs_boundary.Value = c_max;
431
432     l2_errors = [];
433
434     % trapezoidal method
435     trapezoidal_method = IntegrationMethod();
436     trapezoidal_method.type = IntegrationType.Trapezoidal;
437     trapezoidal_method.gauss_points = 0; % not used for trapezoidal
438
439     trapezoidal_solution = NumericSolver.SolveNumeric(
440         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,
441         trapezoidal_method);
442
443     % gaussian quadrature method
444     gaussian_method = IntegrationMethod();
445     gaussian_method.type = IntegrationType.Gaussian;
446     gaussian_method.gauss_points = 3; % 3-point Gaussian quadrature
```

```
447     gaussian_solution = NumericSolver.SolveNumeric(...  
448         mesh, tmax, dt, theta, lhs_boundary, rhs_boundary, @(~, ~) 0.0,  
449         gaussian_method);  
450  
451         % compute L2 error  
452         l2_error_trapezoidal = L2Error(analytical_solution, trapezoidal_solution);  
453         l2_error_gaussian = L2Error(analytical_solution, gaussian_solution);  
454         l2_errors = [l2_error_trapezoidal, l2_error_gaussian];  
455  
456         method_names = {"2-point Trapezoidal Integration", "3-point Gaussian  
457         Quadrature"};  
458  
459         Plotter.PlotL2Errors(l2_errors, "Gaussian vs Trapezoidal Integration", ...  
460             "cw2/report/resources/part2/L2ErrorGaussianTrapezoidal", ...  
461             method_names);  
462     end  
463  
464     function Part2QuadraticBasisFunctions()  
465         %  
466         % Function:      Part2TimeIntegrationComparison()  
467         %  
468         % Arguments:    None  
469         % Returns:       None  
470         %  
471         % Description:   Runs a study comparing different time integration  
472             % methods for Part 2 of the coursework.  
473         %  
474  
475  
476     end  
477  
478 end  
479  
480 end  
481 end  
482
```

11. Mesh

11.1. Mesh.m

```
1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Mesh.m
6 % Author    : samh25
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A class defining a one-dimensional mesh for
10 %                finite element analysis.
11 %
12 %%%%%%
13
14
15 classdef Mesh < handle
16     % inherit from handle to allow pass-by-reference
17
18     properties
19
20         xmin double
21         xmax double
22         dx double
23
24         order double
25
26         D double % diffusion coefficient
27         lambda double % reaction coefficient
28
29         node_count uint64
30         node_coords double % coordinates of global nodes
31
32         element_count uint64
33         elements MeshElement % array of mesh elements
34
35     end
36
37     methods
38
39         %% Mesh constructor
40         function obj = Mesh(xmin, xmax, element_count, order, D, lambda)
41
42             obj.xmin = xmin;
43             obj.xmax = xmax;
44             obj.dx = (xmax - xmin) / element_count;
45             obj.D = D;
46             obj.lambda = lambda;
47
48             obj.order = order;
49
50             % total number of nodes
51             obj.node_count = (element_count * order) + 1;
52             obj.node_coords = zeros(1, obj.node_count);
53
54             obj.element_count = element_count;
55             obj.elements = MeshElement.empty(element_count, 0);
56
57         end
58
59         function obj = Generate(obj)
60
61             disp('Generating normal mesh... ');
62
63             % generate uniform node coordinates
```

```

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

```

11.2. MeshElement.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : MeshElement.m
6 % Author    : samh25
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A class defining a one-dimensional mesh element
10 %
11 %%%%%%
12
13 classdef MeshElement
14
15     properties
16
17         order      uint8    % polynomial order (1 = linear, 2 = quadratic)
18         node_ids   uint64   % global node IDs
19         node_coords double   % node coordinates
20         jacobian   double   % element jacobian d(x)/d(xi)
21         D          double   % diffusion coefficient
22         lambda     double   % reaction coefficient
23     end
24
25     methods
26
27         %% MeshElement constructor
28         function obj = MeshElement(ids, coords, order, D, lambda)
29
30             % assign properties
31             obj.node_ids = ids;
32             obj.node_coords = coords;
33             obj.order = order;
34             obj.D = D;
35             obj.lambda = lambda;
36
37             % linear mapping from [-1, 1] to [x1, x2]
38             % jacobian = dx/dxi = (x2 - x1) / 2
39             obj.jacobian = (coords(end) - coords(1)) / 2;
40
41         end

```

```
42     end
43 end
44
45
```

11.3. MultilayerMesh.m

```
1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Mesh.m
6 % Author    : samh25
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A class defining a one-dimensional mesh for
10 %                finite element analysis.
11 %
12 %%%%%%
13
14
15 classdef MultilayerMesh < Mesh
16     % inherit from handle to allow pass-by-reference
17
18     properties
19         layer_properties LayerProperties % array of layer properties
20     end
21
22     methods
23
24         %% Mesh constructor
25         function obj = MultilayerMesh(xmin, xmax, element_count, order, D, lambda,
26             layer_properties)
27
28             obj = obj@Mesh(xmin, xmax, element_count, order, D, lambda);
29             obj.layer_properties = layer_properties;
30
31         end
32
33         function obj = Generate(obj)
34
35             disp('Generating multilayer mesh... ');
36
37             % generate uniform node coordinates
38             obj.node_coords = linspace(obj.xmin, obj.xmax, obj.node_count);
39
40             % generate elements
41             for e = 1:obj.element_count
42
43                 % determine global node IDs for this element
44                 node_start = (e - 1) * obj.order + 1;
45                 node_ids = node_start:(node_start + obj.order);
46
47                 % coordinates for this element
48                 coords = obj.node_coords(node_ids);
49
50                 midpoint = (coords(1) + coords(end)) / 2;
51
52                 % determine which layer this element is in
53                 layer_index = 1;
54
55                 for l = 1:length(obj.layer_properties)
56                     if midpoint >= obj.layer_properties(l).x
57                         layer_index = l;
58                     end
59                 end
60             end
61         end
62     end
63
64     function obj = setLayerProperties(obj, layer_properties)
65
66         % Set the layer properties for the mesh
67         obj.layer_properties = layer_properties;
68
69     end
70
71     function [x, y] = getCoordinates(obj)
72
73         % Get the coordinates of the mesh
74         x = obj.node_coords;
75         y = obj.layer_properties(layer_index).y;
76
77     end
78
79     function obj = plotMesh(obj)
80
81         % Plot the mesh
82         figure;
83         plot(obj.node_coords, obj.layer_properties(layer_index).y);
84         title('Multilayer Mesh');
85
86     end
87
88     function obj = saveMesh(obj, filename)
89
90         % Save the mesh to a file
91         save(filename, 'obj');
92
93     end
94
95     function obj = loadMesh(filename)
96
97         % Load a saved mesh from a file
98         obj = load(filename);
99
100    end
101
102    function obj = copy(obj)
103
104        % Create a copy of the mesh
105        obj_copy = obj;
106
107        % Set the layer properties for the copied mesh
108        obj_copy.layer_properties = obj.layer_properties;
109
110    end
111
112    function obj = clear(obj)
113
114        % Clear the mesh
115        obj = obj@Mesh();
116
117    end
118
119    function obj = delete(obj)
120
121        % Delete the mesh
122        delete(obj);
123
124    end
125
126    function obj = isMesh(obj)
127
128        % Check if the object is a mesh
129        if isobject(obj, 'Mesh')
130            return true;
131        else
132            return false;
133        end
134
135    end
136
137    function obj = isMultilayerMesh(obj)
138
139        % Check if the object is a multilayer mesh
140        if isobject(obj, 'MultilayerMesh')
141            return true;
142        else
143            return false;
144        end
145
146    end
147
148    function obj = isLayerProperties(obj)
149
150        % Check if the object is a layer properties
151        if isobject(obj, 'LayerProperties')
152            return true;
153        else
154            return false;
155        end
156
157    end
158
159    function obj = isLinspace(obj)
160
161        % Check if the object is a linspace
162        if isobject(obj, 'linspace')
163            return true;
164        else
165            return false;
166        end
167
168    end
169
170    function obj = isFigure(obj)
171
172        % Check if the object is a figure
173        if isobject(obj, 'Figure')
174            return true;
175        else
176            return false;
177        end
178
179    end
180
181    function obj = isPlot(obj)
182
183        % Check if the object is a plot
184        if isobject(obj, 'Plot')
185            return true;
186        else
187            return false;
188        end
189
190    end
191
192    function obj = isSave(obj)
193
194        % Check if the object is a save
195        if isobject(obj, 'Save')
196            return true;
197        else
198            return false;
199        end
200
201    end
202
203    function obj = isLoad(obj)
204
205        % Check if the object is a load
206        if isobject(obj, 'Load')
207            return true;
208        else
209            return false;
210        end
211
212    end
213
214    function obj = isDelete(obj)
215
216        % Check if the object is a delete
217        if isobject(obj, 'Delete')
218            return true;
219        else
220            return false;
221        end
222
223    end
224
225    function obj = isCopy(obj)
226
227        % Check if the object is a copy
228        if isobject(obj, 'Copy')
229            return true;
230        else
231            return false;
232        end
233
234    end
235
236    function obj = isClear(obj)
237
238        % Check if the object is a clear
239        if isobject(obj, 'Clear')
240            return true;
241        else
242            return false;
243        end
244
245    end
246
247    function obj = isIsMesh(obj)
248
249        % Check if the object is an ismesh
250        if isobject(obj, 'IsMesh')
251            return true;
252        else
253            return false;
254        end
255
256    end
257
258    function obj = isIsMultilayerMesh(obj)
259
260        % Check if the object is an ismultilayermesh
261        if isobject(obj, 'IsMultilayerMesh')
262            return true;
263        else
264            return false;
265        end
266
267    end
268
269    function obj = isIsLayerProperties(obj)
270
271        % Check if the object is an islayerproperties
272        if isobject(obj, 'IsLayerProperties')
273            return true;
274        else
275            return false;
276        end
277
278    end
279
280    function obj = isIsLinspace(obj)
281
282        % Check if the object is an isolinspace
283        if isobject(obj, 'IsLinspace')
284            return true;
285        else
286            return false;
287        end
288
289    end
290
291    function obj = isIsFigure(obj)
292
293        % Check if the object is an isfigure
294        if isobject(obj, 'IsFigure')
295            return true;
296        else
297            return false;
298        end
299
300    end
301
302    function obj = isIsPlot(obj)
303
304        % Check if the object is an isplot
305        if isobject(obj, 'IsPlot')
306            return true;
307        else
308            return false;
309        end
310
311    end
312
313    function obj = isIsSave(obj)
314
315        % Check if the object is an issave
316        if isobject(obj, 'IsSave')
317            return true;
318        else
319            return false;
320        end
321
322    end
323
324    function obj = isIsLoad(obj)
325
326        % Check if the object is an isload
327        if isobject(obj, 'IsLoad')
328            return true;
329        else
330            return false;
331        end
332
333    end
334
335    function obj = isIsDelete(obj)
336
337        % Check if the object is an isdelete
338        if isobject(obj, 'IsDelete')
339            return true;
340        else
341            return false;
342        end
343
344    end
345
346    function obj = isIsCopy(obj)
347
348        % Check if the object is an iscopy
349        if isobject(obj, 'IsCopy')
350            return true;
351        else
352            return false;
353        end
354
355    end
356
357    function obj = isIsClear(obj)
358
359        % Check if the object is an isclear
360        if isobject(obj, 'IsClear')
361            return true;
362        else
363            return false;
364        end
365
366    end
367
368    function obj = isIsIsMesh(obj)
369
370        % Check if the object is an isismesh
371        if isobject(obj, 'IsIsMesh')
372            return true;
373        else
374            return false;
375        end
376
377    end
378
379    function obj = isIsIsMultilayerMesh(obj)
380
381        % Check if the object is an isismultilayermesh
382        if isobject(obj, 'IsIsMultilayerMesh')
383            return true;
384        else
385            return false;
386        end
387
388    end
389
390    function obj = isIsIsLayerProperties(obj)
391
392        % Check if the object is an isislayerproperties
393        if isobject(obj, 'IsIsLayerProperties')
394            return true;
395        else
396            return false;
397        end
398
399    end
400
401    function obj = isIsIsLinspace(obj)
402
403        % Check if the object is an isolinspace
404        if isobject(obj, 'IsIsLinspace')
405            return true;
406        else
407            return false;
408        end
409
410    end
411
412    function obj = isIsIsFigure(obj)
413
414        % Check if the object is an isisfigure
415        if isobject(obj, 'IsIsFigure')
416            return true;
417        else
418            return false;
419        end
420
421    end
422
423    function obj = isIsIsPlot(obj)
424
425        % Check if the object is an isisplot
426        if isobject(obj, 'IsIsPlot')
427            return true;
428        else
429            return false;
429        end
430
431    end
432
433    function obj = isIsIsSave(obj)
434
435        % Check if the object is an isissave
436        if isobject(obj, 'IsIsSave')
437            return true;
438        else
439            return false;
439        end
440
441    end
442
443    function obj = isIsIsLoad(obj)
444
445        % Check if the object is an isisload
446        if isobject(obj, 'IsIsLoad')
447            return true;
448        else
449            return false;
449        end
450
451    end
452
453    function obj = isIsIsDelete(obj)
454
455        % Check if the object is an isisdelete
456        if isobject(obj, 'IsIsDelete')
457            return true;
458        else
459            return false;
459        end
460
461    end
462
463    function obj = isIsIsCopy(obj)
464
465        % Check if the object is an isiscopy
466        if isobject(obj, 'IsIsCopy')
467            return true;
468        else
469            return false;
469        end
470
471    end
472
473    function obj = isIsIsClear(obj)
474
475        % Check if the object is an isisclear
476        if isobject(obj, 'IsIsClear')
477            return true;
478        else
479            return false;
479        end
480
481    end
482
483    function obj = isIsIsIsMesh(obj)
484
485        % Check if the object is an isisismesh
486        if isobject(obj, 'IsIsIsMesh')
487            return true;
488        else
489            return false;
489        end
490
491    end
492
493    function obj = isIsIsIsMultilayerMesh(obj)
494
495        % Check if the object is an isisismultilayermesh
496        if isobject(obj, 'IsIsIsMultilayerMesh')
497            return true;
498        else
499            return false;
499        end
500
501    end
502
503    function obj = isIsIsIsLayerProperties(obj)
504
505        % Check if the object is an isisislayerproperties
506        if isobject(obj, 'IsIsIsLayerProperties')
507            return true;
508        else
509            return false;
509        end
510
511    end
512
513    function obj = isIsIsIsLinspace(obj)
514
515        % Check if the object is an isolinspace
516        if isobject(obj, 'IsIsIsLinspace')
517            return true;
518        else
519            return false;
519        end
520
521    end
522
523    function obj = isIsIsIsFigure(obj)
524
525        % Check if the object is an isisisfigure
526        if isobject(obj, 'IsIsIsFigure')
527            return true;
528        else
529            return false;
529        end
530
531    end
532
533    function obj = isIsIsIsPlot(obj)
534
535        % Check if the object is an isisisplot
536        if isobject(obj, 'IsIsIsPlot')
537            return true;
538        else
539            return false;
539        end
540
541    end
542
543    function obj = isIsIsIsSave(obj)
544
545        % Check if the object is an isisisave
546        if isobject(obj, 'IsIsIsSave')
547            return true;
548        else
549            return false;
549        end
550
551    end
552
553    function obj = isIsIsIsLoad(obj)
554
555        % Check if the object is an isisisload
556        if isobject(obj, 'IsIsIsLoad')
557            return true;
558        else
559            return false;
559        end
560
561    end
562
563    function obj = isIsIsIsDelete(obj)
564
565        % Check if the object is an isisisdelete
566        if isobject(obj, 'IsIsIsDelete')
567            return true;
568        else
569            return false;
569        end
570
571    end
572
573    function obj = isIsIsIsCopy(obj)
574
575        % Check if the object is an isisiscopy
576        if isobject(obj, 'IsIsIsCopy')
577            return true;
578        else
579            return false;
579        end
580
581    end
582
583    function obj = isIsIsIsClear(obj)
584
585        % Check if the object is an isisisclear
586        if isobject(obj, 'IsIsIsClear')
587            return true;
588        else
589            return false;
589        end
590
591    end
592
593    function obj = isIsIsIsIsMesh(obj)
594
595        % Check if the object is an isisisisismesh
596        if isobject(obj, 'IsIsIsIsMesh')
597            return true;
598        else
599            return false;
599        end
600
601    end
602
603    function obj = isIsIsIsIsMultilayerMesh(obj)
604
605        % Check if the object is an isisisisismultilayermesh
606        if isobject(obj, 'IsIsIsIsMultilayerMesh')
607            return true;
608        else
609            return false;
609        end
610
611    end
612
613    function obj = isIsIsIsIsLayerProperties(obj)
614
615        % Check if the object is an isisisisislayerproperties
616        if isobject(obj, 'IsIsIsIsLayerProperties')
617            return true;
618        else
619            return false;
619        end
620
621    end
622
623    function obj = isIsIsIsIsLinspace(obj)
624
625        % Check if the object is an isolinspace
626        if isobject(obj, 'IsIsIsIsLinspace')
627            return true;
628        else
629            return false;
629        end
630
631    end
632
633    function obj = isIsIsIsIsFigure(obj)
634
635        % Check if the object is an isisisisisfigure
636        if isobject(obj, 'IsIsIsIsFigure')
637            return true;
638        else
639            return false;
639        end
640
641    end
642
643    function obj = isIsIsIsIsPlot(obj)
644
645        % Check if the object is an isisisisisplot
646        if isobject(obj, 'IsIsIsIsPlot')
647            return true;
648        else
649            return false;
649        end
650
651    end
652
653    function obj = isIsIsIsIsSave(obj)
654
655        % Check if the object is an isisisisisave
656        if isobject(obj, 'IsIsIsIsSave')
657            return true;
658        else
659            return false;
659        end
660
661    end
662
663    function obj = isIsIsIsIsLoad(obj)
664
665        % Check if the object is an isisisisisload
666        if isobject(obj, 'IsIsIsIsLoad')
667            return true;
668        else
669            return false;
669        end
670
671    end
672
673    function obj = isIsIsIsIsDelete(obj)
674
675        % Check if the object is an isisisisisdelete
676        if isobject(obj, 'IsIsIsIsDelete')
677            return true;
678        else
679            return false;
679        end
680
681    end
682
683    function obj = isIsIsIsIsCopy(obj)
684
685        % Check if the object is an isisisiscopy
686        if isobject(obj, 'IsIsIsIsCopy')
687            return true;
688        else
689            return false;
689        end
690
691    end
692
693    function obj = isIsIsIsIsClear(obj)
694
695        % Check if the object is an isisisisclear
696        if isobject(obj, 'IsIsIsIsClear')
697            return true;
698        else
699            return false;
699        end
700
701    end
702
703    function obj = isIsIsIsIsIsMesh(obj)
704
705        % Check if the object is an isisisisisisismesh
706        if isobject(obj, 'IsIsIsIsIsMesh')
707            return true;
708        else
709            return false;
709        end
710
711    end
712
713    function obj = isIsIsIsIsIsMultilayerMesh(obj)
714
715        % Check if the object is an isisisisisisismultilayermesh
716        if isobject(obj, 'IsIsIsIsIsMultilayerMesh')
717            return true;
718        else
719            return false;
719        end
720
721    end
722
723    function obj = isIsIsIsIsIsLayerProperties(obj)
724
725        % Check if the object is an isisisisisisislayerproperties
726        if isobject(obj, 'IsIsIsIsIsLayerProperties')
727            return true;
728        else
729            return false;
729        end
730
731    end
732
733    function obj = isIsIsIsIsIsLinspace(obj)
734
735        % Check if the object is an isolinspace
736        if isobject(obj, 'IsIsIsIsIsLinspace')
737            return true;
738        else
739            return false;
739        end
740
741    end
742
743    function obj = isIsIsIsIsIsFigure(obj)
744
745        % Check if the object is an isisisisisisisfigure
746        if isobject(obj, 'IsIsIsIsIsFigure')
747            return true;
748        else
749            return false;
749        end
750
751    end
752
753    function obj = isIsIsIsIsIsPlot(obj)
754
755        % Check if the object is an isisisisisisisplot
756        if isobject(obj, 'IsIsIsIsIsPlot')
757            return true;
758        else
759            return false;
759        end
760
761    end
762
763    function obj = isIsIsIsIsIsSave(obj)
764
765        % Check if the object is an isisisisisisisave
766        if isobject(obj, 'IsIsIsIsIsSave')
767            return true;
768        else
769            return false;
769        end
770
771    end
772
773    function obj = isIsIsIsIsIsLoad(obj)
774
775        % Check if the object is an isisisisisisisload
776        if isobject(obj, 'IsIsIsIsIsLoad')
777            return true;
778        else
779            return false;
779        end
780
781    end
782
783    function obj = isIsIsIsIsIsDelete(obj)
784
785        % Check if the object is an isisisisisisisdelete
786        if isobject(obj, 'IsIsIsIsIsDelete')
787            return true;
788        else
789            return false;
789        end
790
791    end
792
793    function obj = isIsIsIsIsIsCopy(obj)
794
795        % Check if the object is an isisisisisiscopy
796        if isobject(obj, 'IsIsIsIsIsCopy')
797            return true;
798        else
799            return false;
799        end
800
801    end
802
803    function obj = isIsIsIsIsIsClear(obj)
804
805        % Check if the object is an isisisisisisclear
806        if isobject(obj, 'IsIsIsIsIsClear')
807            return true;
808        else
809            return false;
809        end
810
811    end
812
813    function obj = isIsIsIsIsIsIsMesh(obj)
814
815        % Check if the object is an isisisisisisisisismesh
816        if isobject(obj, 'IsIsIsIsIsIsMesh')
817            return true;
818        else
819            return false;
819        end
820
821    end
822
823    function obj = isIsIsIsIsIsIsMultilayerMesh(obj)
824
825        % Check if the object is an isisisisisisisisismultilayermesh
826        if isobject(obj, 'IsIsIsIsIsIsMultilayerMesh')
827            return true;
828        else
829            return false;
829        end
830
831    end
832
833    function obj = isIsIsIsIsIsIsLayerProperties(obj)
834
835        % Check if the object is an isisisisisisisisislayerproperties
836        if isobject(obj, 'IsIsIsIsIsIsLayerProperties')
837            return true;
838        else
839            return false;
839        end
840
841    end
842
843    function obj = isIsIsIsIsIsIsLinspace(obj)
844
845        % Check if the object is an isolinspace
846        if isobject(obj, 'IsIsIsIsIsIsLinspace')
847            return true;
848        else
849            return false;
849        end
850
851    end
852
853    function obj = isIsIsIsIsIsIsFigure(obj)
854
855        % Check if the object is an isisisisisisisisisfigure
856        if isobject(obj, 'IsIsIsIsIsIsFigure')
857            return true;
858        else
859            return false;
859        end
860
861    end
862
863    function obj = isIsIsIsIsIsIsPlot(obj)
864
865        % Check if the object is an isisisisisisisisisplot
866        if isobject(obj, 'IsIsIsIsIsIsPlot')
867            return true;
868        else
869            return false;
869        end
870
871    end
872
873    function obj = isIsIsIsIsIsIsSave(obj)
874
875        % Check if the object is an isisisisisisisisisave
876        if isobject(obj, 'IsIsIsIsIsIsSave')
877            return true;
878        else
879            return false;
879        end
880
881    end
882
883    function obj = isIsIsIsIsIsIsLoad(obj)
884
885        % Check if the object is an isisisisisisisisisload
886        if isobject(obj, 'IsIsIsIsIsIsLoad')
887            return true;
888        else
889            return false;
889        end
890
891    end
892
893    function obj = isIsIsIsIsIsIsDelete(obj)
894
895        % Check if the object is an isisisisisisisisisdelete
896        if isobject(obj, 'IsIsIsIsIsIsDelete')
897            return true;
898        else
899            return false;
899        end
900
901    end
902
903    function obj = isIsIsIsIsIsIsCopy(obj)
904
905        % Check if the object is an isisisisisisisiscopy
906        if isobject(obj, 'IsIsIsIsIsIsCopy')
907            return true;
908        else
909            return false;
909        end
910
911    end
912
913    function obj = isIsIsIsIsIsIsClear(obj)
914
915        % Check if the object is an isisisisisisisisclear
916        if isobject(obj, 'IsIsIsIsIsIsClear')
917            return true;
918        else
919            return false;
919        end
920
921    end
922
923    function obj = isIsIsIsIsIsIsIsMesh(obj)
924
925        % Check if the object is an isisisisisisisisisisismesh
926        if isobject(obj, 'IsIsIsIsIsIsIsMesh')
927            return true;
928        else
929            return false;
929        end
930
931    end
932
933    function obj = isIsIsIsIsIsIsIsMultilayerMesh(obj)
934
935        % Check if the object is an isisisisisisisisisisismultilayermesh
936        if isobject(obj, 'IsIsIsIsIsIsIsMultilayerMesh')
937            return true;
938        else
939            return false;
939        end
940
941    end
942
943    function obj = isIsIsIsIsIsIsIsLayerProperties(obj)
944
945        % Check if the object is an isisisisisisisisisisislayerproperties
946        if isobject(obj, 'IsIsIsIsIsIsIsLayerProperties')
947            return true;
948        else
949            return false;
949        end
950
951    end
952
953    function obj = isIsIsIsIsIsIsIsLinspace(obj)
954
955        % Check if the object is an isolinspace
956        if isobject(obj, 'IsIsIsIsIsIsIsLinspace')
957            return true;
958        else
959            return false;
959        end
960
961    end
962
963    function obj = isIsIsIsIsIsIsIsFigure(obj)
964
965        % Check if the object is an isisisisisisisisisisisfigure
966        if isobject(obj, 'IsIsIsIsIsIsIsFigure')
967            return true;
968        else
969            return false;
969        end
970
971    end
972
973    function obj = isIsIsIsIsIsIsIsPlot(obj)
974
975        % Check if the object is an isisisisisisisisisisisplot
976        if isobject(obj, 'IsIsIsIsIsIsIsPlot')
977            return true;
978        else
979            return false;
979        end
980
981    end
982
983    function obj = isIsIsIsIsIsIsIsSave(obj)
984
985        % Check if the object is an isisisisisisisisisisisave
986        if isobject(obj, 'IsIsIsIsIsIsIsSave')
987            return true;
988        else
989            return false;
989        end
990
991    end
992
993    function obj = isIsIsIsIsIsIsIsLoad(obj)
994
995        % Check if the object is an isisisisisisisisisisisload
996        if isobject(obj, 'IsIsIsIsIsIsIsLoad')
997            return true;
998        else
999            return false;
999        end
1000
1001    end
1002
1003    function obj = isIsIsIsIsIsIsIsDelete(obj)
1004
1005        % Check if the object is an isisisisisisisisisisisdelete
1006        if isobject(obj, 'IsIsIsIsIsIsIsDelete')
1007            return true;
1008        else
1009            return false;
1009        end
1010
1011    end
1012
1013    function obj = isIsIsIsIsIsIsIsCopy(obj)
1014
1015        % Check if the object is an isisisisisisisisisiscopy
1016        if isobject(obj, 'IsIsIsIsIsIsIsCopy')
1017            return true;
1018        else
1019            return false;
1019        end
1020
1021    end
1022
1023    function obj = isIsIsIsIsIsIsIsClear(obj)
1024
1025        % Check if the object is an isisisisisisisisisisclear
1026        if isobject(obj, 'IsIsIsIsIsIsIsClear')
1027            return true;
1028        else
1029            return false;
1029        end
1030
1031    end
1032
1033    function obj = isIsIsIsIsIsIsIsIsMesh(obj)
1034
1035        % Check if the object is an isisisisisisisisisisisisismesh
1036        if isobject(obj, 'IsIsIsIsIsIsIsIsMesh')
1037            return true;
1038        else
1039            return false;
1039        end
1040
1041    end
1042
1043    function obj = isIsIsIsIsIsIsIsIsMultilayerMesh(obj)
1044
1045        % Check if the object is an isisisisisisisisisisisisismultilayermesh
1046        if isobject(obj, 'IsIsIsIsIsIsIsIsMultilayerMesh')
1047            return true;
1048        else
1049            return false;
1049        end
1050
1051    end
1052
1053    function obj = isIsIsIsIsIsIsIsIsLayerProperties(obj)
1054
1055        % Check if the object is an isisisisisisisisisisisisislayerproperties
1056        if isobject(obj, 'IsIsIsIsIsIsIsIsLayerProperties')
1057            return true;
1058        else
1059            return false;
1059        end
1060
1061    end
1062
1063    function obj = isIsIsIsIsIsIsIsIsLinspace(obj)
1064
1065        % Check if the object is an isolinspace
1066        if isobject(obj, 'IsIsIsIsIsIsIsIsLinspace')
1067            return true;
1068        else
1069            return false;
1069        end
1070
1071    end
1072
1073    function obj = isIsIsIsIsIsIsIsIsFigure(obj)
1074
1075        % Check if the object is an isisisisisisisisisisisisisfigure
1076        if isobject(obj, 'IsIsIsIsIsIsIsIsFigure')
1077            return true;
1078        else
1079            return false;
1079        end
1080
1081    end
1082
1083    function obj = isIsIsIsIsIsIsIsIsPlot(obj)
1084
1085        % Check if the object is an isisisisisisisisisisisisisplot
1086        if isobject(obj, 'IsIsIsIsIsIsIsIsPlot')
1087            return true;
1088        else
1089            return false;
1089        end
1090
1091    end
1092
1093    function obj = isIsIsIsIsIsIsIsIsSave(obj)
1094
1095        % Check if the object is an isisisisisisisisisisisisisave
1096        if isobject(obj, 'IsIsIsIsIsIsIsIsSave')
1097            return true;
1098        else
1099            return false;
1099        end
1100
1101    end
1102
1103    function obj = isIsIsIsIsIsIsIsIsLoad(obj)
1104
1105        % Check if the object is an isisisisisisisisisisisisisload
1106        if isobject(obj, 'IsIsIsIsIsIsIsIsLoad')
1107            return true;
1108        else
1109            return false;
1109        end
1110
1111    end
1112
1113    function obj = isIsIsIsIsIsIsIsIsDelete(obj)
1114
1115        % Check if the object is an isisisisisisisisisisisisisdelete
1116        if isobject(obj, 'IsIsIsIsIsIsIsIsDelete')
1117            return true;
1118        else
1119            return false;
1119        end
1120
1121    end
1122
1123    function obj = isIsIsIsIsIsIsIsIsCopy(obj)
1124
1125        % Check if the object is an isisisisisisisisisisisiscopy
1126        if isobject(obj, 'IsIsIsIsIsIsIsIsCopy')
1127            return true;
1128        else
1129            return false;
1129        end
1130
1131    end
1132
1133    function obj = isIsIsIsIsIsIsIsIsClear(obj)
1134
1135        % Check if the object is an isisisisisisisisisisisisclear
1136        if isobject(obj, 'IsIsIsIsIsIsIsIsClear')
1137            return true;
1138        else
1139            return false;
1139        end
1140
1141    end
1142
1143    function obj = isIsIsIsIsIsIsIsIsIsMesh(obj)
1144
1145        % Check if the object is an isisisisisisisisisisisisisisismesh
1146        if isobject(obj, 'IsIsIsIsIsIsIsIsIsMesh')
1147            return true;
1148        else
1149            return false;
1149        end
1150
1151    end
1152
1153    function obj = isIsIsIsIsIsIsIsIsIsMultilayerMesh(obj)
1154
1155        % Check if the object is an isisisisisisisisisisisisisisismultilayermesh
1156        if isobject(obj, 'IsIsIsIsIsIsIsIsIsMultilayerMesh')
1157            return true;
1158        else
1159            return false;
1159        end
1160
1161    end
1162
1163    function obj = isIsIsIsIsIsIsIsIsIsLayerProperties(obj)
1164
1165        % Check if the object is an isisisisisisisisisisisisisisislayerproperties
1166        if isobject(obj, 'IsIsIsIsIsIsIsIsIsLayerProperties')
1167            return true;
1168        else
1169            return false;
1169        end
1170
1171    end
1172
1173    function obj = isIsIsIsIsIsIsIsIsIsLinspace(obj)
1174
1175        % Check if the object is an isolinspace
1176        if isobject(obj, 'IsIsIsIsIsIsIsIsIsLinspace')
1177            return true;
1178        else
1179            return false;
1179        end
1180
1181    end
1182
1183    function obj = isIsIsIsIsIsIsIsIsIsFigure(obj)
1184
1185        % Check if the object is an isisisisisisisisisisisisisisisfigure
1186        if isobject(obj, 'IsIsIsIsIsIsIsIsIsFigure')
1187            return true;
1188        else
1189            return false;
1189        end
1190
1191    end
1192
1193    function obj = isIsIsIsIsIsIsIsIsIsPlot(obj)
1194
1195        % Check if the object is an isisisisisisisisisisisisisisisplot
1196        if isobject(obj, 'IsIsIsIsIsIsIsIsIsPlot')
1197            return true;
1198        else
1199            return false;
1199        end
1200
1201    end
1202
1203    function obj = isIsIsIsIsIsIsIsIsIsSave(obj)
1204
```

```
58         end
59
60         D = obj.layer_properties(layer_index).D;
61         lambda = -(obj.layer_properties(layer_index).beta +
62             obj.layer_properties(layer_index).gamma);
63
64         % create MeshElement object
65         obj.elements(e) = MeshElement(node_ids, coords, obj.order, D, lambda);
66     end
67
68 end
69 end
70
71
```

11.4. LayerProperties.m

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

12. Analytical

12.1. AnalyticalSolver.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : AnalyticalSolver.m
6 % Author    : samh25
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A static class defining an analytical solver
10 %           for the transient diffusion equation
11 %
12 %%%%%%
13
14 classdef AnalyticalSolver
15
16     methods (Static)
17
18         function solution = SolveAnalytical(mesh, tmax, dt)
19
20             % time vector
21             time_vector = 0:dt:tmax;
22             solution = Solution(mesh, time_vector);
23
24             % loop over time steps
25             for step = 1:length(time_vector)
26
27                 t = time_vector(step);
28                 timestep_results = zeros(1, mesh.node_count);
29
30                 % loop over nodes
31                 for i = 1:mesh.node_count
32                     x = mesh.node_coords(i);
33                     timestep_results(i) = TransientAnalyticSoln(x, t);
34                 end
35
36                 solution.SetValues(timestep_results, step);
37             end
38
39         end
40     end
41 end

```

12.2. TransientAnalyticSoln.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : TransientAnalyticSoln.m
6 % Author    : A. N. Cookson
7 % Created   : 2025-11-11 (YYYY-MM-DD)
8 % License   : -
9 % Description : Analytical solution to transient diffusion equation
10 %
11 %%%%%%
12
13 function [ c ] = TransientAnalyticSoln(x,t)
14 %TransientAnalyticSoln Analytical solution to transient diffusion equation
15 % Computes the analytical solution to the transient diffusion equation for
16 % the domain x=[0,1], subject to initial condition: c(x,0) = 0, and Dirichlet
17 % boundary conditions: c(0,t) = 0, and c(1,t) = 1.
18 % Input Arguments:

```

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

13. Plotter

13.1. Plotter.m

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

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

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

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

14. Solution

14.1. Solution.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : Solution.m
6 % Author    : samh25
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A class defining a solution to the transient
10 %               diffusion equation
11 %
12 %%%%%%
13
14 classdef Solution < handle
15     % inherit from handle to allow pass-by-reference behaviour
16
17     properties
18
19         mesh      Mesh      % handle to mesh object
20         time     double    % time series - 1 x Nsteps
21         values    double    % solution values - Nnodes x Nsteps
22
23     end
24
25     methods
26
27         %% Solution constructor
28         function obj = Solution(mesh, time_vector)
29
30             % assign properties
31             obj.mesh = mesh;
32             obj.time = time_vector;
33             obj.values = zeros(mesh.node_count, length(time_vector));
34
35         end
36
37         %% Set solution values at given time step
38         function SetValues(obj, values, step)
39             % set solution values at given time step
40             obj.values(:, step) = values(:, );
41         end
42     end
43 end
44
45

```

14.2. L2Error.m

```

1 %%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : L2Error.m
6 % Author    : samh25
7 % Created   : 2025-11-26 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : A class defining a solution to the transient
10 %               diffusion equation
11 %
12 %%%%%%
13
14 classdef L2Error < handle

```

```

15 % inherit from handle to allow pass-by-reference behaviour
16
17 properties
18
19     ref_solution    Solution % handle to reference solution object
20     num_solution    Solution % handle to solution object
21
22     time            double   % time series - 1 x Nsteps
23     l2_error        double   % L2 error at each time step - 1 x Nsteps
24
25 end
26
27 methods
28
29 %% Solution constructor
30 function obj = L2Error(ref_solution, num_solution)
31
32     % assign properties
33     obj.ref_solution = ref_solution;
34     obj.num_solution = num_solution;
35     obj.time = ref_solution.time;
36
37     if ref_solution.mesh.node_count ~= num_solution.mesh.node_count
38         error('Reference and error solutions must have the same number of nodes');
39     end
40
41     if length(ref_solution.time) ~= length(num_solution.time)
42         error('Reference and error solutions must have the same number of time
43 steps');
44     end
45
46     step_count = length(ref_solution.time);
47
48     obj.l2_error = zeros(1, step_count);
49
50     for step = 1:step_count
51         c_ref = ref_solution.values(:, step);
52         c_num = num_solution.values(:, step);
53         x = ref_solution.mesh.node_coords;
54
55         integrand = (c_ref - c_num).^2;
56         obj.l2_error(step) = sqrt(trapz(x, integrand));
57     end
58
59 end
60 end
61 end
62
63

```

14.3. DoseEvaluator.m

```

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

```

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

15. Solver

15.1. NumericSolver.m

```

1 %%%%%%%%%%%%%%
2 %
3 % ME40064 Coursework 2
4 %
5 % File      : NumericSolver.m
6 % Author    : samh25
7 % Created   : 2025-11-24 (YYYY-MM-DD)
8 % License   : MIT
9 % Description : Class definition for generic solver for the
10 %                transient diffusion-reaction equation
11 %
12 %%%%%%%%%%%%%%
13
14
15 classdef NumericSolver
16
17     methods (Static)
18
19         function solution = SolveNumeric(mesh, tmax, dt, theta, left_boundary,
20             right_boundary, source_fn, integration_method)
21
22             % time vector
23             time_vector = 0:dt:tmax;
24             solution = Solution(mesh, time_vector);
25
26             % === SET INITIAL CONDITION EXPLICITLY ===
27             c0 = zeros(mesh.node_count, 1);
28             solution.SetValues(c0, 1); % column 1 = t=0
29
30             [K, M] = NumericSolver.CreateGlobalMatrices(mesh, theta, integration_method);
31
32             % loop over time steps
33             for step = 1:length(time_vector) - 1
34
35                 c_next = NumericSolver.SolveStep(mesh, solution, step, dt, theta, K, M,
36                     left_boundary, right_boundary, source_fn, integration_method);
37                 solution.SetValues(c_next, step + 1);
38
39             end
40
41         end
42
43         function c = SolveStep(mesh, solution, step, dt, theta, K, M, left_boundary,
44             right_boundary, source_fn, integration_method)
45
46             c_current = solution.values(:, step);
47
48             t = (step - 1) * dt; % current time, converted to 0-based index
49
50             % assemble system matrix and rhs vector
51             system_matrix = M + theta * dt * K;
52             rhs_vector = (M - (1 - theta) * dt * K) * c_current;
53
54             % add source term
55             f_current = NumericSolver.CreateSourceVector(mesh, t, source_fn,
56                 integration_method);
57             f_next = NumericSolver.CreateSourceVector(mesh, t + dt, source_fn,
58                 integration_method);
59             rhs_vector = rhs_vector + dt * (theta * f_next + (1 - theta) * f_current);
60
61             % apply boundary conditions

```

```
57     [system_matrix, rhs_vector] =
58     NumericSolver.ApplyBoundaryConditions(system_matrix, rhs_vector, t + dt, left_boundary,
59                                         right_boundary);
60
61     % solve system
62     c = system_matrix \ rhs_vector;
63
64 %% Create global stiffness and mass matrices
65 function [K, M] = CreateGlobalMatrices(mesh, theta, integration_method)
66
67     num_elements = mesh.element_count;
68     num_nodes = mesh.node_count;
69
70     % initialise global matrix (use sparse for efficiency with large systems)
71     K = sparse(num_nodes, num_nodes);
72     M = sparse(num_nodes, num_nodes);
73
74     for element_id = 1:num_elements
75
76         element = mesh.elements(element_id);
77         nodes = element.node_ids;
78         local_size = length(nodes);
79
80         diff_matrix = ElementMatrices.DiffusionElemMatrix(element,
81 integration_method);
82         react_matrix = ElementMatrices.ReactionElemMatrix(element,
83 integration_method);
84
85         k_matrix = diff_matrix - react_matrix;
86
87         elem_size = element.node_coords(end) - element.node_coords(1);
88         m_matrix = ElementMatrices.MassElemMatrix(element, integration_method);
89
90         % assemble into global matrices
91         for i = 1:local_size
92             for j = 1:local_size
93                 gi = nodes(i); gj = nodes(j);
94                 K(gi, gj) = K(gi, gj) + k_matrix(i, j);
95                 M(gi, gj) = M(gi, gj) + m_matrix(i, j);
96             end
97         end
98
99     end
100
101 %% Create source vector for given time
102 function F = CreateSourceVector(mesh, t, source_fn, integration_method)
103
104     F = zeros(mesh.node_count, 1);
105
106     % return if no source function defined
107     if (isempty(source_fn))
108         return;
109     end
110
111     for element_id = 1:mesh.element_count
112
113         element = mesh.elements(element_id);
114
115         elem_size = element.node_coords(end) - element.node_coords(1);
116         midpoint = (element.node_coords(1) + element.node_coords(end)) / 2;
117
118         f_val = source_fn(midpoint, t);
119
120     end
121
122 %% Create boundary conditions
123 function bc = CreateBoundaryConditions(mesh, t, left_boundary, right_boundary)
124
125     bc = zeros(mesh.node_count, 1);
126
127     % set boundary values
128     bc(left_boundary) = 1;
129     bc(right_boundary) = 1;
130
131     % set interior values to zero
132     for i = 1:mesh.node_count
133         if (i < left_boundary) || (i > right_boundary)
134             bc(i) = 0;
135         end
136     end
137
138 %% Create initial conditions
139 function IC = CreateInitialConditions(mesh, t, initial_value)
140
141     IC = zeros(mesh.node_count, 1);
142
143     % set all nodes to initial value
144     IC = initial_value;
145
146 %% Create element matrices
147 function [K, M] = CreateElementMatrices(element, integration_method)
148
149     % get node coordinates
150     node_coords = element.node_coords;
151
152     % calculate element size
153     elem_size = node_coords(end) - node_coords(1);
154
155     % calculate mid-point
156     midpoint = (node_coords(1) + node_coords(end)) / 2;
157
158     % calculate element area
159     area = elem_size * sqrt(1 + sum((node_coords(2:end) - node_coords(1:end-1)).^2));
160
161     % calculate element shape functions
162     N1 = 1 - (x - node_coords(1)) / elem_size;
163     N2 = 1 - (x - node_coords(2)) / elem_size;
164
165     % calculate element Jacobian
166     J = [1, 0; 0, 1];
167
168     % calculate element stiffness matrix
169     K = zeros(2, 2);
170     for i = 1:2
171         for j = 1:2
172             K(i, j) = int1d(0, 1, N1.*N2, J);
173         end
174     end
175
176     % calculate element mass matrix
177     M = zeros(2, 2);
178     for i = 1:2
179         for j = 1:2
180             M(i, j) = int1d(0, 1, N1.*N2, J);
181         end
182     end
183
184     % return element matrices
185     K = K / area;
186     M = M / area;
187
188 %% Create diffusion matrix
189 function K = CreateDiffusionMatrix(element, integration_method)
190
191     % get element size
192     elem_size = element.node_coords(end) - element.node_coords(1);
193
194     % calculate element Jacobian
195     J = [1, 0; 0, 1];
196
197     % calculate element stiffness matrix
198     K = zeros(2, 2);
199     for i = 1:2
200         for j = 1:2
201             K(i, j) = int1d(0, 1, 1, J);
202         end
203     end
204
205     % return diffusion matrix
206     K = K / elem_size;
207
208 %% Create reaction matrix
209 function R = CreateReactionMatrix(element, integration_method)
210
211     % get element size
212     elem_size = element.node_coords(end) - element.node_coords(1);
213
214     % calculate element Jacobian
215     J = [1, 0; 0, 1];
216
217     % calculate element stiffness matrix
218     R = zeros(2, 2);
219     for i = 1:2
220         for j = 1:2
221             R(i, j) = int1d(0, 1, 1, J);
222         end
223     end
224
225     % return reaction matrix
226     R = R / elem_size;
227
228 %% Create mass matrix
229 function M = CreateMassMatrix(element, integration_method)
230
231     % get element size
232     elem_size = element.node_coords(end) - element.node_coords(1);
233
234     % calculate element Jacobian
235     J = [1, 0; 0, 1];
236
237     % calculate element stiffness matrix
238     M = zeros(2, 2);
239     for i = 1:2
240         for j = 1:2
241             M(i, j) = int1d(0, 1, 1, J);
242         end
243     end
244
245     % return mass matrix
246     M = M / elem_size;
```

```

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

```

15.2. BoundaryCondition.m

```

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

```

15.3. BoundaryType.m

```

1 classdef BoundaryType
2     enumeration
3         Dirichlet, Neumann

```

```

4     end
5 end

```

15.4. ElementMatrices.m

```

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

```

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

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

```
194     end  
195 end
```

15.5. IntegrationMethod.m

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

15.6. IntegrationType.m

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

16. Tests

16.1. NumericSolverTest.m

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

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