

ME40064 System Modelling and Simulation - Coursework 1

Seb Hall samh25@bath.ac.uk, 4th November 2025

Department of Mechanical Engineering, University of Bath

1. Q1: Local Element Matrix Functions

1.1. Q1a: MATLAB 2x2 Element Matrix for the Diffusion Operator

1.1.1. Unit Test Result

Properties	Duration	Details	Name	Passed	Failed	Incomplete
1	0.1107	1x1 struct	'FormativeAssignmentUnitTest/Test1_TestSymmetryOfTheMatrix'	1	0	0
2	0.0128	1x1 struct	'FormativeAssignmentUnitTest/Test2_Test2DifferentElementsOfTheSameSizeProduceSameMatrix'	1	0	0
3	0.0084	1x1 struct	'FormativeAssignmentUnitTest/Test3_TestThatOneMatrixIsEvaluatedCorrectly'	1	0	0
4	0.0253	1x1 struct	'FormativeAssignmentUnitTest/Test4_TestThatDifferentSizedElementsInAMeshAreEvaluatedCorrectly_Element1'	1	0	0
5	0.0097	1x1 struct	'FormativeAssignmentUnitTest/Test5_TestThatDifferentSizedElementsInAMeshAreEvaluatedCorrectly_Element4'	1	0	0

Figure 1: Diffusion Operator Unit Test Results

1.1.2. Function Code

```
function matrix = DiffusionElemMatrix(D, eID, msh)
%% DiffusionElemMatrix - calculates a 2x2 element matrix for the diffusion
%% operator in a 1d finite element mesh.

% create base matrix
matrix = [1, -1; -1, 1];

% calculate element size
elemSize = msh.elem(eID).x(2) - msh.elem(eID).x(1);

% apply matrix scaling
matrix = matrix * (D / elemSize);

end
```

Code Snippet 1: Diffusion Operator Element Matrix Function

1.2. Q1b: MATLAB 2x2 Element Matrix for the Linear Reaction Operator

1.2.1. Unit Test Result

Properties	Duration	Details	Name	Passed	Failed	Incomplete
1	0.2603	1x1 struct	'ReactionUnitTest/Test1_TestSymmetryOfTheMatrix'	1	0	0
2	0.0324	1x1 struct	'ReactionUnitTest/Test2_Test2DifferentElementsOfTheSameSizeProduceSameMatrix'	1	0	0
3	0.0224	1x1 struct	'ReactionUnitTest/Test3_TestThatOneMatrixIsEvaluatedCorrectly'	1	0	0
4	0.0643	1x1 struct	'ReactionUnitTest/Test4_TestThatDifferentSizedElementsInAMeshAreEvaluatedCorrectly_...'	1	0	0
5	0.0247	1x1 struct	'ReactionUnitTest/Test5_TestThatDifferentSizedElementsInAMeshAreEvaluatedCorrectly_...'	1	0	0

Figure 2: Linear Reaction Operator Unit Test Results

1.2.2. Function Code

```
function matrix = ReactionElemMatrix(lambda, eID, msh)
%% ReactionElemMatrix - calculates a 2x2 element matrix for the linear
%% reaction operator in a 1d finite element mesh.

% create base matrix
matrix = [2, 1; 1, 2];

% calculate element size
elemSize = msh.elem(eID).x(2) - msh.elem(eID).x(1);

% apply matrix scaling
matrix = matrix * (lambda * elemSize / 6);
```

```
end
```

Code Snippet 2: Linear Reaction Operator Element Matrix Function

1.2.3. Unit Test 1

```
%% Test 1: test symmetry of the matrix
% % Test that this matrix is symmetric

tol = 1e-14; % test tolerance
lambda = 2; % reaction coefficient
eID = 1; % element ID

xmin = 0;
xmax = 1;
Ne = 10;
msh = OneDimLinearMeshGen(xmin, xmax, Ne);

elemat = ReactionElemMatrix(lambda, eID, msh);

assert(abs(elemat(1,2) - elemat(2,1)) <= tol)
```

Code Snippet 3: Linear Reaction Operator Unit Test 1

1.2.4. Unit Test 2

```
%% Test 2: test 2 different elements of the same size produce same matrix
% % Test that for two elements of an equispaced mesh, the element matrices
% % are calculated are the same.

tol = 1e-14; % test tolerance
lambda = 5; % reaction coefficient
eID = 1; % element ID

xmin = 0;
xmax = 1;
Ne = 10;
msh = OneDimLinearMeshGen(xmin, xmax, Ne);

elemat1 = ReactionElemMatrix(lambda, eID, msh);

eID = 2; %element ID
elemat2 = ReactionElemMatrix(lambda, eID, msh);

diff = elemat1 - elemat2;
diffnorm = sum(sum(diff.*diff));
assert(abs(diffnorm) <= tol)
```

Code Snippet 4: Linear Reaction Operator Unit Test 2

1.2.5. Unit Test 3

```
%% Test 3: test that one matrix is evaluated correctly
% % Test that element 1 of the (equispaced) three element mesh
% % problem is evaluated correctly

tol = 1e-14; % test tolerance
lambda = 2.5; % reaction coefficient
eID = 1; % element ID

xmin = 0;
xmax = 1;
Ne = 3;
msh = OneDimLinearMeshGen(xmin, xmax, Ne);
```

```

elemat1 = ReactionElemMatrix(lambda, eID, msh);

matrix = [2, 1; 1, 2];
elemSize = (xmax - xmin) / Ne;
elemat2 = matrix * (lambda * elemSize / 6);

diff = elemat1 - elemat2; % calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); % calculate the total squared error between the matrices
assert(abs(diffnorm) <= tol)

```

Code Snippet 5: Linear Reaction Operator Unit Test 3

1.2.6. Unit Test 4

```

%% Test 4: test that different sized elements in a mesh are evaluated correctly - element 1
% % Test that elements in a non-equally spaced mesh are evaluated correctly

tol = 1e-14; % test tolerance
lambda = 1; % reaction coefficient
eID = 1; % element ID

xmin = 0;
xmax = 1;
Ne = 5;
msh = OneDimSimpleRefinedMeshGen(xmin, xmax, Ne);

elemat1 = ReactionElemMatrix(lambda, eID, msh);

elemSize = msh.elem(eID).x(2) - msh.elem(eID).x(1); % get element size from mesh
elemat2 = [2, 1; 1, 2] * (lambda * elemSize / 6); % calculate resultant matrix

diff = elemat1 - elemat2; % calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); % calculate the total squared error between the matrices
assert(abs(diffnorm) <= tol)

```

Code Snippet 6: Linear Reaction Operator Unit Test 4

1.2.7. Unit Test 5

```

%% Test 5: test that different sized elements in a mesh are evaluated correctly - element 4
% % Test that elements in a non-equally spaced mesh are evaluated correctly

tol = 1e-14; % test tolerance
lambda = 1; % reaction coefficient
eID = 4; % element ID

xmin = 0;
xmax = 1;
Ne = 5;
msh = OneDimSimpleRefinedMeshGen(xmin, xmax, Ne);

elemat1 = ReactionElemMatrix(lambda, eID, msh);

elemSize = msh.elem(eID).x(2) - msh.elem(eID).x(1); % get element size from mesh
elemat2 = [2, 1; 1, 2] * (lambda * elemSize / 6); % calculate resultant matrix

diff = elemat1 - elemat2; % calculate the difference between the two matrices
diffnorm = sum(sum(diff.*diff)); % calculate the total squared error between the matrices
assert(abs(diffnorm) <= tol)

```

Code Snippet 7: Linear Reaction Operator Unit Test 5

2. Q2: Solving Laplace's Equation using FEM

This task was to solve Laplace's equation:

$$\frac{\delta^2 c}{\delta x^2} = 0$$

With the following boundary conditions:

$$\begin{aligned}\frac{\delta c}{\delta x}(x = 0) &= 2 \\ c(x = 1) &= 0\end{aligned}$$

Over a 1D uniform mesh with 4 elements.

2.1. Results

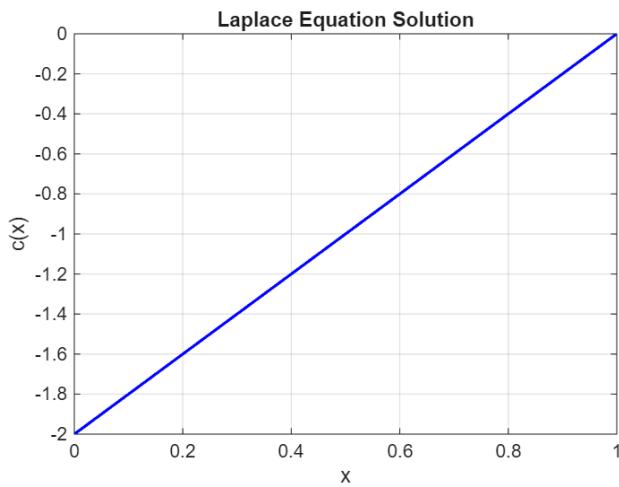


Figure 3: Solution of the Laplace Equation using FEM Solver

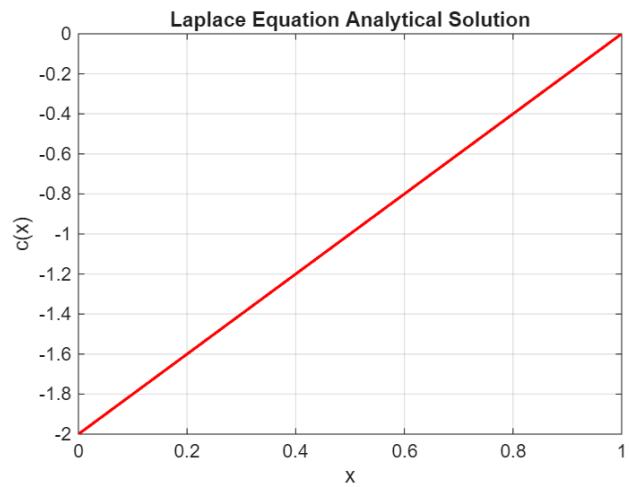


Figure 4: Solution of the Laplace Equation using an Analytical Method, $y = 2(x + 1)$

These results show that for this example, the solver is able to produce a perfect solution for the Laplace equation. Integrating the Laplace equation (the gradient of the function) is constant, and is therefore representable perfectly by discrete sample points.

2.2. Source Code

2.2.1. Solver

```
classdef BoundaryType
enumeration
    Dirichlet, Neumann
end
end
```

Code Snippet 8: BoundaryType.m

```
classdef BoundaryCondition
properties
    Type % Boundary condition type (Dirichlet or Neumann)
    Value % Boundary condition value
end
end
```

Code Snippet 9: BoundaryCondition.m

```

function solution = DiffusionReactionSolver(mesh, D, lambda, leftBoundary, rightBoundary)
%% DiffusionReactionSolver - solves the steady-state diffusion-reaction equation
%% for a 1D mesh
% % Inputs:
%   mesh - 1D finite element mesh structure
%   D - diffusion coefficient
%   lambda - reaction rate
%   leftBoundary - left boundary condition (BoundaryCondition object)
%   rightBoundary - right boundary condition (BoundaryCondition object)
%
% % Outputs:
%   solution - solution vector at mesh nodes

% calculate number of nodes and elements
Ne = mesh.ne;
Nn = Ne + 1;

% initialise global matrix
globalMatrix = zeros(Nn, Nn);

% assemble global matrix
for eID = 1:Ne

    % get element matrices
    diffusionElementMatrix = DiffusionElemMatrix(D, eID, mesh);
    reactionElementMatrix = ReactionElemMatrix(lambda, eID, mesh);

    % combine element matrices
    elemMatrix = diffusionElementMatrix - reactionElementMatrix;

    % insert into global matrix
    globalMatrix(eID, eID) = globalMatrix(eID, eID) + elemMatrix(1, 1);
    globalMatrix(eID, eID + 1) = globalMatrix(eID, eID + 1) + elemMatrix(1, 2);
    globalMatrix(eID + 1, eID) = globalMatrix(eID + 1, eID) + elemMatrix(2, 1);
    globalMatrix(eID + 1, eID + 1) = globalMatrix(eID + 1, eID + 1) + elemMatrix(2, 2);

end

% initialise source vector
sourceVector = zeros(Nn, 1);

% here we would assemble the source vector if there were any source terms
% however, for now we assume there are none, so it remains zero

% Apply left boundary condition
switch leftBoundary.Type

    case BoundaryType.Neumann

        % directly modify source vector for Neumann
        sourceVector(1) = sourceVector(1) - leftBoundary.Value;

    case BoundaryType.Dirichlet

        % apply Dirichlet condition to source vector
        for j = 2:Nn
            sourceVector(j) = sourceVector(j) - globalMatrix(j, 1) *
leftBoundary.Value;
        end

        % modify global matrix
        globalMatrix(1, :) = 0;
        globalMatrix(:, 1) = 0;
        globalMatrix(1, 1) = 1;

    end

    % set value at first node

```

```

        sourceVector(1) = leftBoundary.Value;
    end

    % Apply right boundary condition
    switch rightBoundary.Type

        case BoundaryType.Neumann

            % directly modify source vector for Neumann
            sourceVector(Nn) = sourceVector(Nn) - rightBoundary.Value;

        case BoundaryType.Dirichlet

            % apply Dirichlet condition to source vector
            for j = 2:(Nn-1)
                sourceVector(j) = sourceVector(j) - globalMatrix(j, Nn) *
rightBoundary.Value;
            end

            % modify global matrix
            globalMatrix(Nn, :) = 0;
            globalMatrix(:, Nn) = 0;
            globalMatrix(Nn, Nn) = 1;

            % set value at last node
            sourceVector(Nn) = rightBoundary.Value;
        end

    % solve system of equations
    solution = globalMatrix \ sourceVector;
end

```

Code Snippet 10: DiffusionReactionSolver.m

2.2.2. Implementation

```

function RunLaplace()
%% RunLaplace: solves a 1D mesh for Laplace's equation as required for Q2.

fprintf("Running Laplace Solver...\n");

% create 1D uniform mesh
xmin = 0;
xmax = 1;
Ne = 4;
mesh = OneDimLinearMeshGen(xmin, xmax, Ne);

% set parameters for Laplace equation
lambda = 0;
D = 1;

% set boundary conditions
leftBoundary = BoundaryCondition;
leftBoundary.Type = BoundaryType.Neumann;
leftBoundary.Value = 2; % dc/dx(c=0) = 2

rightBoundary = BoundaryCondition;
rightBoundary.Type = BoundaryType.Dirichlet;
rightBoundary.Value = 0; % c(1) = 0

% run diffusion reaction solver
solution = DiffusionReactionSolver(mesh, D, lambda, leftBoundary, rightBoundary);

% plot solution (assume uniform 1D mesh)

```

```
x = linspace(xmin, xmax, Ne + 1);
plotHandle = plot(x, solution);

% set chart title and axes
title('Laplace Equation Solution');
xlabel('x');
ylabel('c(x)');
grid on;
set(gcf, 'Position', [0, 0, 500, 350]);
set(plotHandle, 'LineWidth', 1.5);
set(plotHandle, 'Color', [0, 0, 1]);

% save and open figure
saveas(gcf, 'LaplaceEquationSolution.fig');
saveas(gcf, 'cwl/report/resources/LaplaceEquationSolution.png');
openfig('LaplaceEquationSolution.fig');

% plot real solution for comparison
clf;
realSolution = 2*(x - 1);
plotHandle = plot(x, realSolution);

% set chart title and axes
title('Laplace Equation Analytical Solution');
xlabel('x');
ylabel('c(x)');
grid on;
set(gcf, 'Position', [0, 0, 500, 350]);
set(plotHandle, 'LineWidth', 1.5);
set(plotHandle, 'Color', [1, 0, 0]);

% save and open figure
saveas(gcf, 'LaplaceEquationAnalyticalSolution.fig');
saveas(gcf, 'cwl/report/resources/LaplaceEquationAnalyticalSolution.png');
openfig('LaplaceEquationAnalyticalSolution.fig');

end
```

Code Snippet 11: RunLaplace.m

3. Q3: Verifying the FEM Solver for the Diffusion-Reaction Equation

3.1. Results

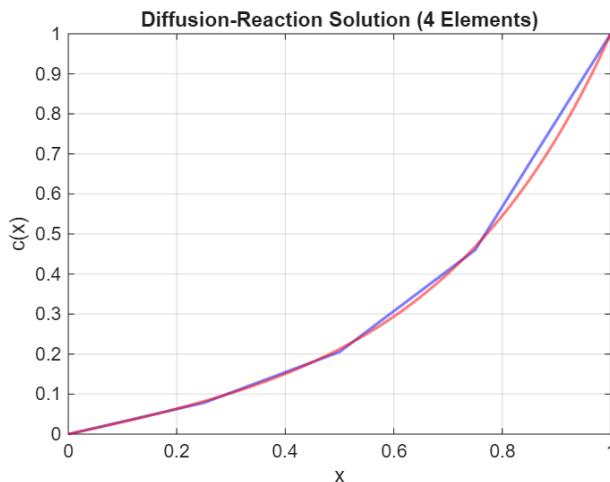


Figure 5: Solution of the Diffusion-Reaction Equation using FEM Solver with 4 Elements (Blue), and Analytical Solution (Red)

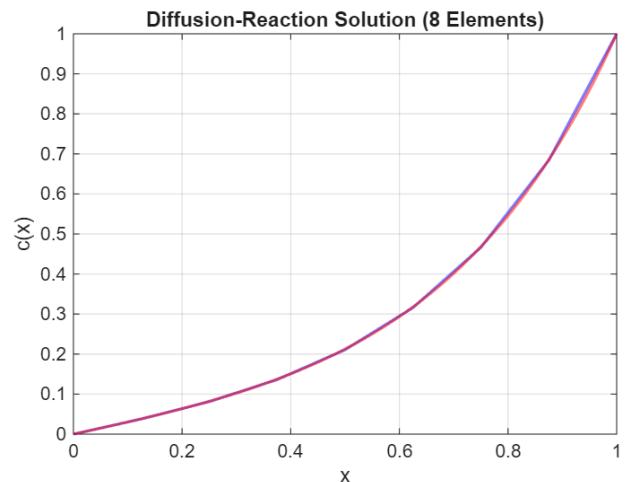


Figure 6: Solution of the Diffusion-Reaction Equation using FEM Solver with 8 Elements (Blue), and Analytical Solution (Red)

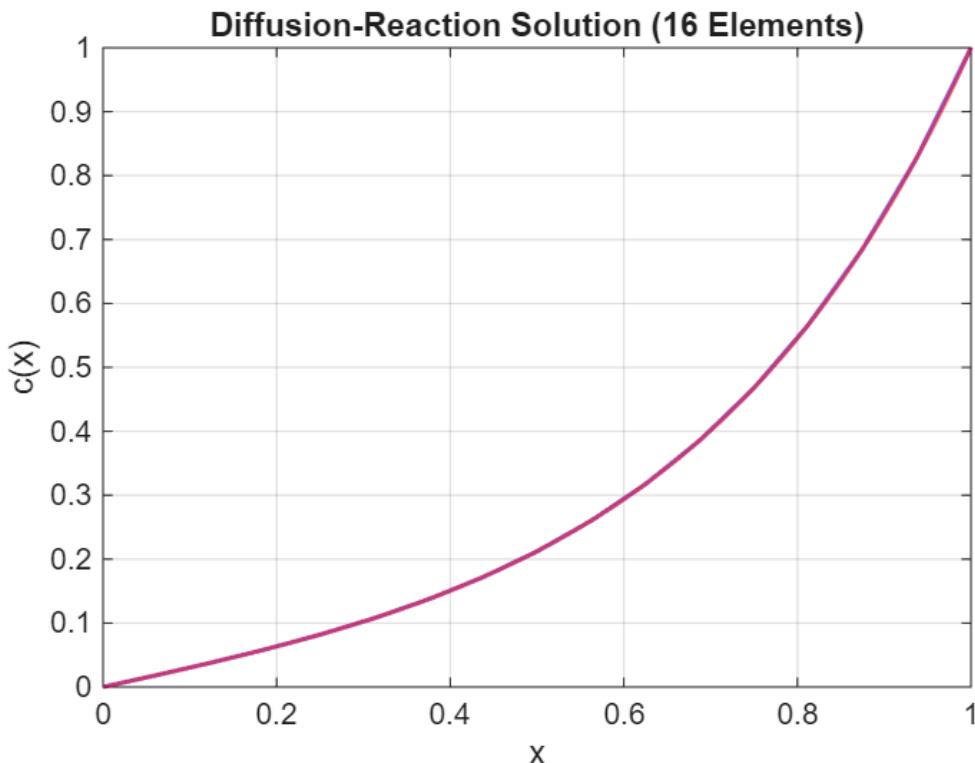


Figure 7: Solution of the Diffusion-Reaction Equation using FEM Solver with 16 Elements (Blue), and Analytical Solution (Red)

3.2. Source Code

The solver was unchanged from Q2 (it was developed to handle diffusion-reaction problems in a generalised form). The implementation code was as follows:

```
function RunDiffusionReaction()
    %% RunDiffusionReaction: solves a 1D mesh for the Diffusion/Reaction
    %% equation as required for Q3.

    fprintf("Running Diffusion/Reaction Solver...\n");
```

```
% array of different element counts
elementCounts = [4, 8, 16];

% iterate over each element count
for Ne = elementCounts
    xmin = 0;
    xmax = 1;

    % create 1D uniform mesh
    mesh = OneDimLinearMeshGen(xmin, xmax, Ne);

    % set parameters for Laplace equation
    lambda = -9;
    D = 1;

    leftBoundary = BoundaryCondition;
    leftBoundary.Type = BoundaryType.Dirichlet;
    leftBoundary.Value = 0; % c(0) = 0 -> concentration at right boundary

    rightBoundary = BoundaryCondition;
    rightBoundary.Type = BoundaryType.Dirichlet;
    rightBoundary.Value = 1; % c(1) = 1 -> concentration at right boundary

    % call diffusion reaction solver
    solution = DiffusionReactionSolver(mesh, D, lambda, leftBoundary, rightBoundary);

    % clear current figure
    clf;

    % generate x vectors
    x = linspace(xmin, xmax, Ne + 1);
    xreal = linspace(xmin, xmax, 500);

    % calculate real solution
    realSolution = exp(3)/(exp(6)-1)*(exp(3*xreal)-exp(-3*xreal));

    % plot solver and real solutions
    solverSolution = plot(x, solution);
    hold on;
    realHandle = plot(xreal, realSolution);

    % set chart title and axes
    title('Diffusion-Reaction Solution (' + string(Ne) + ' Elements)');
    xlabel('x');
    ylabel('c(x)');
    grid on;
    set(gcf, 'Position', [0, 0, 500, 350]);

    % set line properties
    set(realHandle, 'LineWidth', 1.5);
    set(realHandle, 'Color', [1, 0, 0, 0.5]);
    set(solverSolution, 'LineWidth', 1.5);
    set(solverSolution, 'Color', [0, 0, 1, 0.5]);

    % save and open figure
    saveas(gcf, 'DiffusionReactionSolution' + string(Ne) + 'Elements.fig');
    saveas(gcf, 'cw1/report/resources/DiffusionReactionSolution',...
        + string(Ne) + 'Elements.png');
    openfig('DiffusionReactionSolution' + string(Ne) + 'Elements.fig');
end
end
```

Code Snippet 12: RunDiffusionReaction.m

4. 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 for this, and provided generative suggestions for code snippets and report phrasing.