

# Static MATLAB-Based Finite-Element Modelling

## Abstract

$$\frac{\partial^2 c}{\partial x^2} + \lambda c + f = 0$$

*Equation 1: Static diffusion-reaction equation*

An FEM simulation tool has been developed to solve the static diffusion-reaction equation (Equation 1) in a one-dimensional mesh. The tool is designed to be re-usable in the context of a consultancy firm that works on a range of modelling projects.

## Introduction

To numerically solve for  $c$ , Equation 1 was written in matrix form:

$$[Global\ Matrix] \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \end{bmatrix} = [source\ vector] + [boundary\ conditions]$$

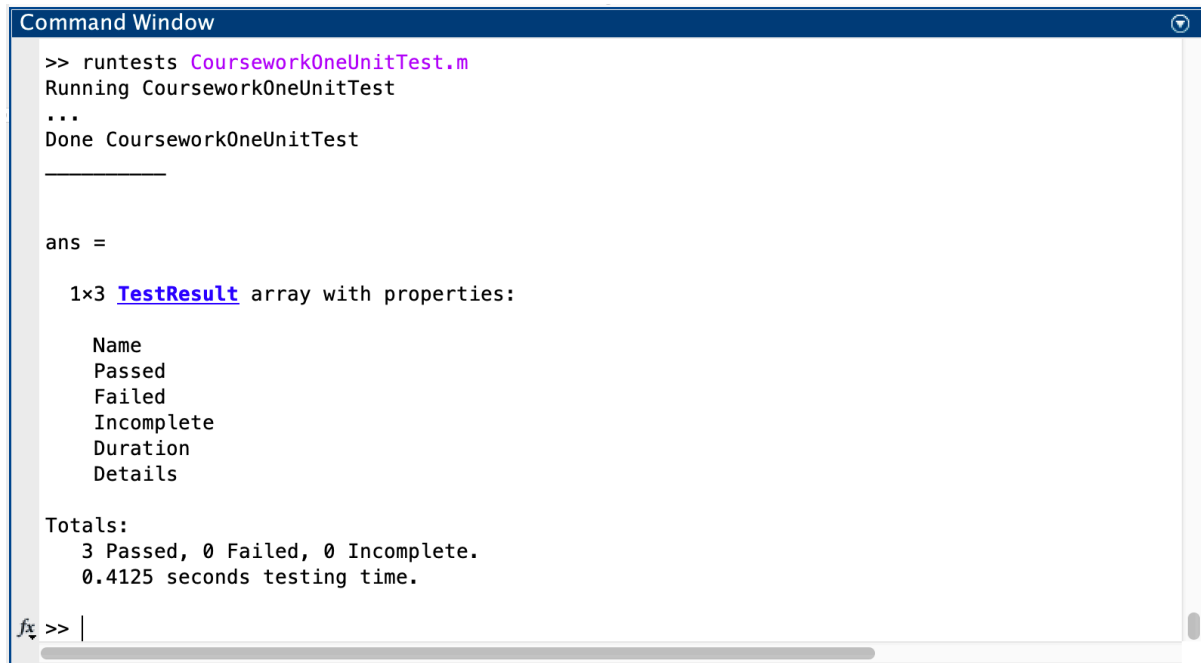
A function, `OneDimLinearMeshGen`, was used to generate a one-dimensional mesh object. A global matrix and global vector were constructed by calculating the terms for each mesh element, before adding them to the global matrix and global source vector.

## Part 1: Software Verification and Analytical Testing

- a. Create a MATLAB function to calculate the local  $2 \times 2$  element matrix for the diffusion operator, for any element in the finite element mesh.

A Matlab function was written to generate a  $2 \times 2$  matrix for each local element representing the diffusion term of the equation. The function, `LaplaceElemMatrix`, is shown in Appendix 1.

The function passed a suite of unit tests as shown in Figure 1.



```
Command Window
>> runtests CourseworkOneUnitTest.m
Running CourseworkOneUnitTest
...
Done CourseworkOneUnitTest

_____

ans =

1x3 TestResult array with properties:

    Name
    Passed
    Failed
    Incomplete
    Duration
    Details

Totals:
  3 Passed, 0 Failed, 0 Incomplete.
  0.4125 seconds testing time.

fx >> |
```

Figure 1: Command window showing successful test results

b. Create a MATLAB function to calculate the local  $2 \times 2$  element matrix for the linear reaction operator, for any element in the finite element mesh.

A similar function was written for the reaction term of the equation. Its source code is shown in Appendix 2.

reactionElemMatrix takes the following arguments as inputs:

- lambda, the reaction coefficient
- eID, the identifier of the mesh element
- msh, the mesh object

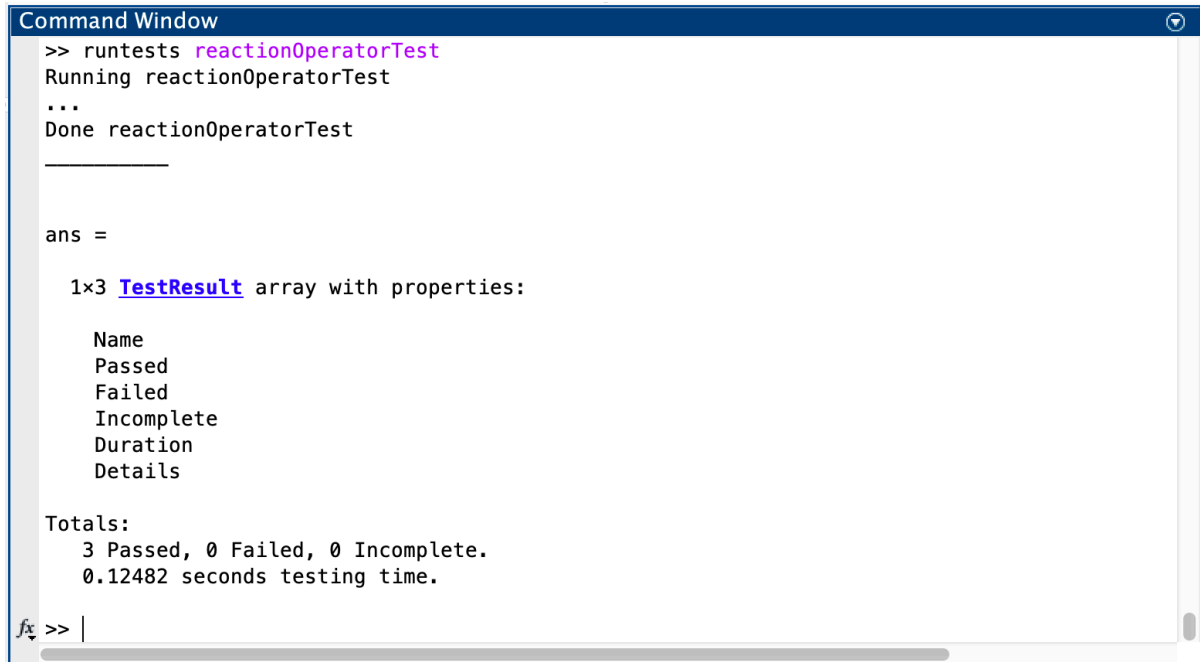
A test suite (Appendix 7) based on Cookson (2019) [1] was written to ensure this function worked as required.

The tests are sufficient to verify the function works correctly because they ensure against the most likely implementation errors:

- inserting matrix elements in the wrong places (by checking a known matrix)
- returning a matrix of the wrong format (by checking symmetry)

- incorrectly using element attributes unrelated to element size (by checking the result is the same for different elements of the same size)

The tests were passed (Figure 2).



```

Command Window
>> runtests reactionOperatorTest
Running reactionOperatorTest
...
Done reactionOperatorTest

-----

ans =

    1x3 TestResult array with properties:

        Name
        Passed
        Failed
        Incomplete
        Duration
        Details

Totals:
    3 Passed, 0 Failed, 0 Incomplete.
    0.12482 seconds testing time.

fx >> |

```

Figure 2: Reaction element test results

### c. Using these functions, develop a finite element solver for the static reaction-diffusion equation.

A simulation tool (the 'solver') was developed using the above functions. The source code of this solver is shown in Appendix 3.

The solver takes the following arguments as inputs:

- mesh, the mesh object on which the solution is calculated
- D, the diffusion coefficient
- lambda, the reaction coefficient
- f, the source term
- dirichletBCs, an N×2 matrix with rows in the format [id, value], where the value 'value' is enforced at the node 'id'.
- neumannBCs, an N×2 matrix in the same format, where the gradient 'value' is enforced at the node 'id'.

A test suite (Appendix 8) was written for the solver to ensure it applied boundary conditions correctly, it correctly solved a problem whose solution was known, and its function dependencies worked as required.

## Analytical Tests

### i. Laplace's Equation with Two Dirichlet Boundary Conditions

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

*Equation 2: Laplace's Equation*

Laplace's Equation was solved on a four-element mesh in the domain  $x = [0, 1]$ . Firstly, two Dirichlet boundary conditions were applied:

$$x = 0, \quad c = 2$$

$$x = 1, \quad c = 0$$

The value of  $c$  at each node was calculated by calling the solver function as follows:

```
finiteElementSolver(mesh,1,0,0,[1 2; 5 0],[]);
```

In this function call, the Dirichlet boundary conditions are enforced at node IDs 1 (the start of the mesh) and 5 (the end of the mesh, as there are five nodes in the mesh).

The results were compared against the analytical solution:

$$c = 2(1 - x)$$

Figure 3 shows the analytical solution as a blue line, with the results calculated by the simulation tool as green circles.

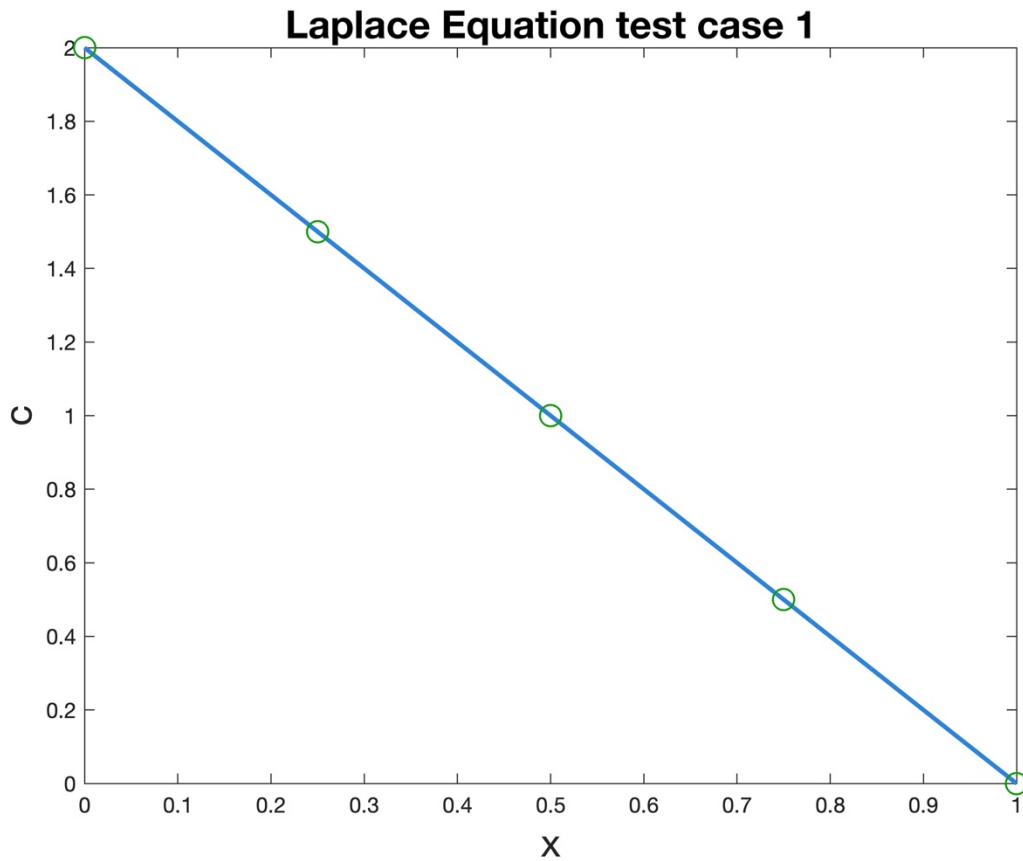


Figure 3: Results of Laplace Equation test case

The tool's results are exactly equal to the analytical solution. This is because Laplace's Equation gives a constant gradient of concentration throughout the solution space.

This solver can exactly solve any first-order system as the value of  $c$  at any point in a local element is approximated by the basis functions:

$$c(\xi) = c_0\psi_0 + c_1\psi_1$$

where:

$$\psi_0 = \frac{1 - \xi}{2}; \quad \psi_1 = \frac{1 + \xi}{2}$$

giving a straight-line approximation. This perfectly fits any straight-line equation but cannot give an exact solution for a curve.

Table 1: Results of solving Laplace's equation

x position	Solver results	Analytical results
0	2.0000	2
0.25	1.5000	1.5
0.5	1.0000	1
0.75	0.5000	0.5
1	0	0

## ii. Laplace's Equation with One Dirichlet and One Neumann Boundary Condition

Laplace's equation was solved on the same mesh with the following Dirichlet and Neumann boundary conditions:

$$x = 0, \quad \frac{\partial c}{\partial x} = 2$$

$$x = 1, \quad c = 0$$

This yielded the result shown in Figure 4.

Changing the boundary condition enforced a gradient of 2 across the entire solution space, because Laplace's equation produces a constant gradient.

To enforce the boundary condition, the top entry of the global source vector was set to -2. As the top row of the global matrix was  $[4 \quad -4 \quad 0 \quad 0]$ , this created the following equation:

$$4C_1 - 4C_2 = -2$$

This enforced the gradient across this element.

Physically, this boundary condition represents a constant flux of the variable  $c$  leaving the domain at  $x = 0$ .

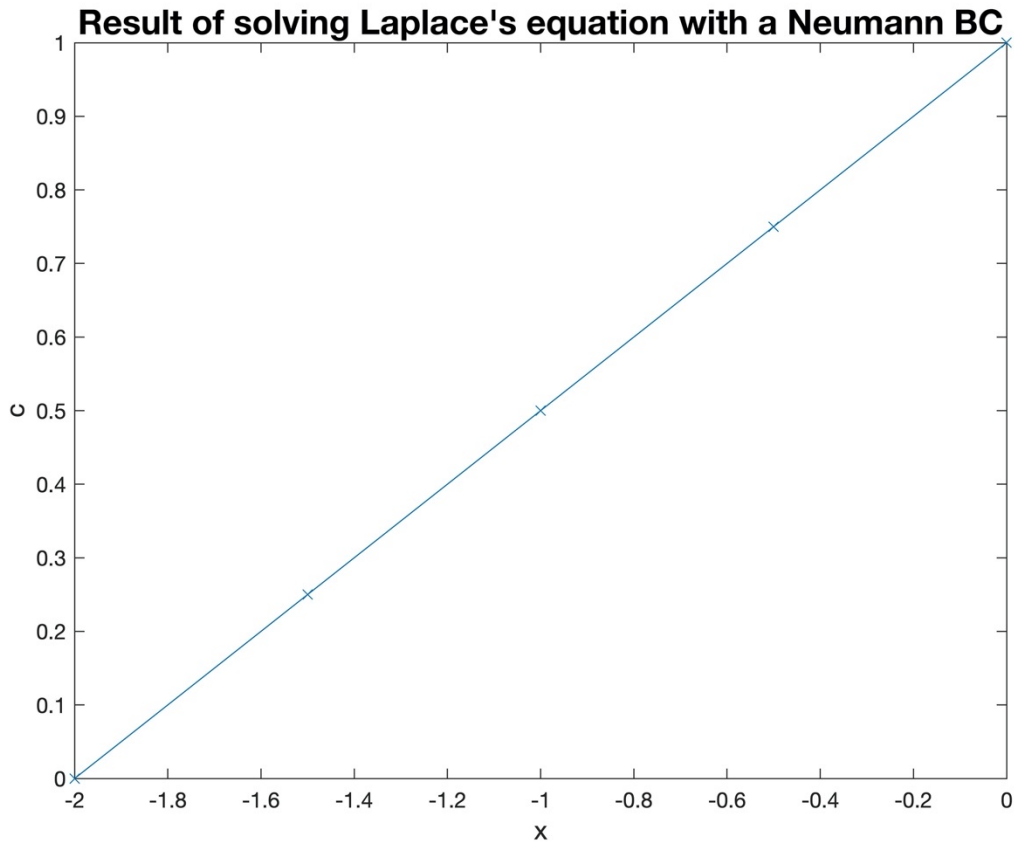


Figure 4: Result of the simulation specified in part ii

### iii. Diffusion-reaction Equation

To test that the solver correctly solved reaction terms, it was used to solve the diffusion-reaction equation:

$$D \frac{\partial^2 c}{\partial x^2} + \lambda c = 0$$

The following parameters were used:

$$D = 1$$

$$\lambda = -9$$

The following Dirichlet boundary conditions were used:

$$x = 0, \quad c = 0$$

$$x = 1, \quad c = 1$$

The analytical solution of this equation for these parameters and boundary conditions is [2]:

$$c(x) = \frac{e^3}{e^6 - 1} (e^{3x} - e^{-3x})$$

First, the solver was used to evaluate the equation on a four-element mesh. Figure 5 shows the results obtained from the simulation tool in orange, with the analytical solution shown in blue.

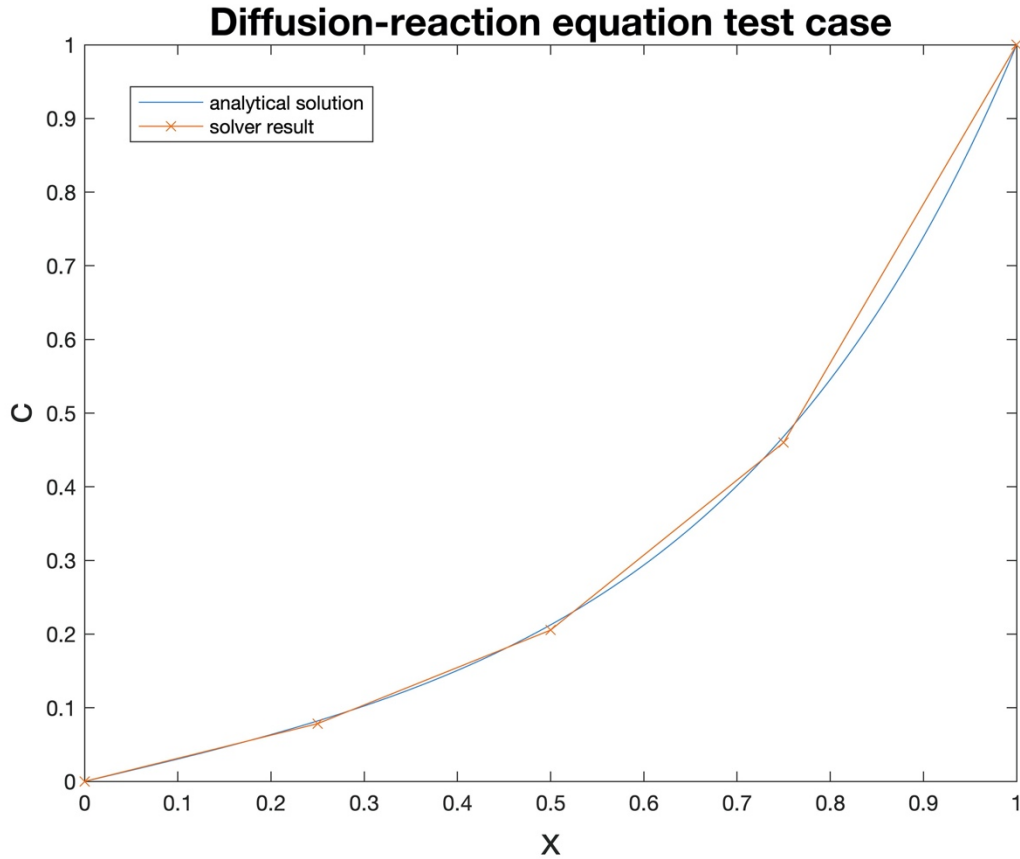


Figure 5: results of solving the diffusion-reaction equation using the solver

Visually, the solver appeared to solve the equation with a good degree of accuracy, but this accuracy needed to be quantified. For this purpose, the *inaccuracy* was defined as the expected difference between the estimated result and the analytical solution at any randomly-selected point.

To quantify the inaccuracy of the solver's solution, the following method was devised:

- Query the solver for the estimated solution at a large number of points (1000 in this case)
- Evaluate the error by finding the difference between the analytical solution and the estimated solution at each point
- Get the average square-root of the error squared at all points (RMS error)



The inaccuracy of the solver's solution was then tested at a range of mesh resolutions from a 1-element mesh to a 100-element mesh. The Matlab script used to perform this procedure is shown in Appendix 4. The results of this script are shown in Figure 6.

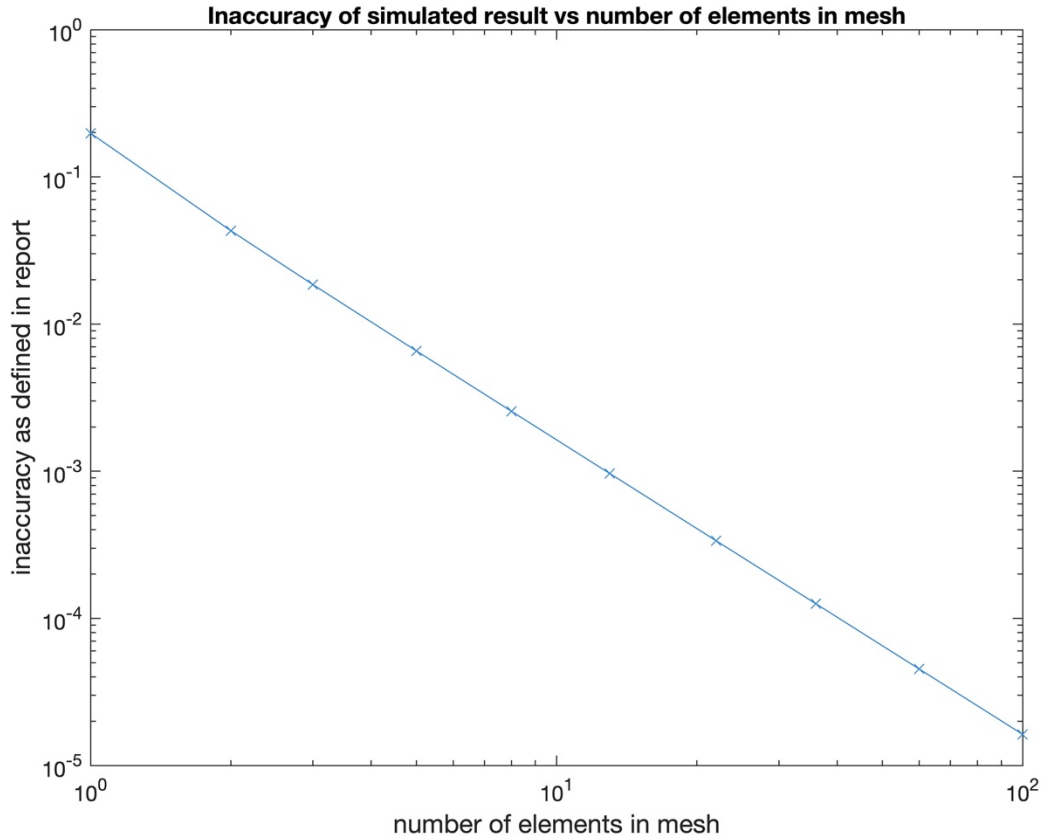


Figure 6: result of testing inaccuracy at a range of mesh resolutions

The solver's solution approaches the analytical solution as the number of mesh elements increases, as shown by the inaccuracy approaching zero.

## Part 2: Modelling and Simulation Results

The solver was used to simulate steady-state heat transfer through a material with small channels passing through it (Figure 7). These channels carry a heating fluid of temperature  $T_L$  at flow rate  $Q$ .

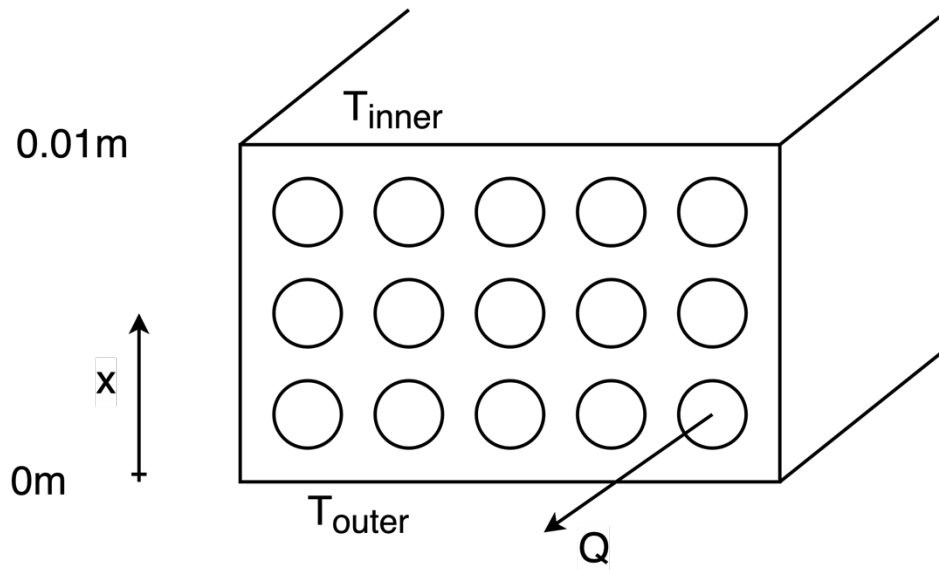


Figure 7: material cross-section, adapted from Cookson (2019) [2]

The system approximates to a one-dimensional problem with the following equation [2] for the static heat distribution:

$$k \frac{\partial^2 T}{\partial x^2} + Q(T_L - T) = 0$$

Rearranging the equation into the same form as Equation 1 gives the values for  $D$ ,  $\lambda$ , and  $f$  which will be used by the solver.

$$k \frac{\partial^2 T}{\partial x^2} - QT + QT_L = 0$$

$$D = k$$

$$\lambda = -Q$$

$$f = QT_L$$

This system has the following values:

Attribute	Value
$T_{inner}$	293.15K (fixed)
$T_{outer}$	323.15K (fixed)
$k$	1.01e-5 (fixed)
$Q$	0.5 to 1.5 (variable)
$T_L$	294.15K to 322.15K (variable)

Table 2: Values of heat transfer system

a. Use your code to solve this model and hence compute the temperature distribution  $T$  for different combinations of values of  $Q$  and  $T_L$

From the investigation shown in Figure 6, a mesh with 100 elements produced a very low error, but this mesh still computed quickly. Therefore, the system was simulated using a 100-element mesh. The following values were used initially:

- $Q = 0.5$
- $T_L = 294.15\text{K}$ .

Figure 8 shows the resulting temperature profile simulation.

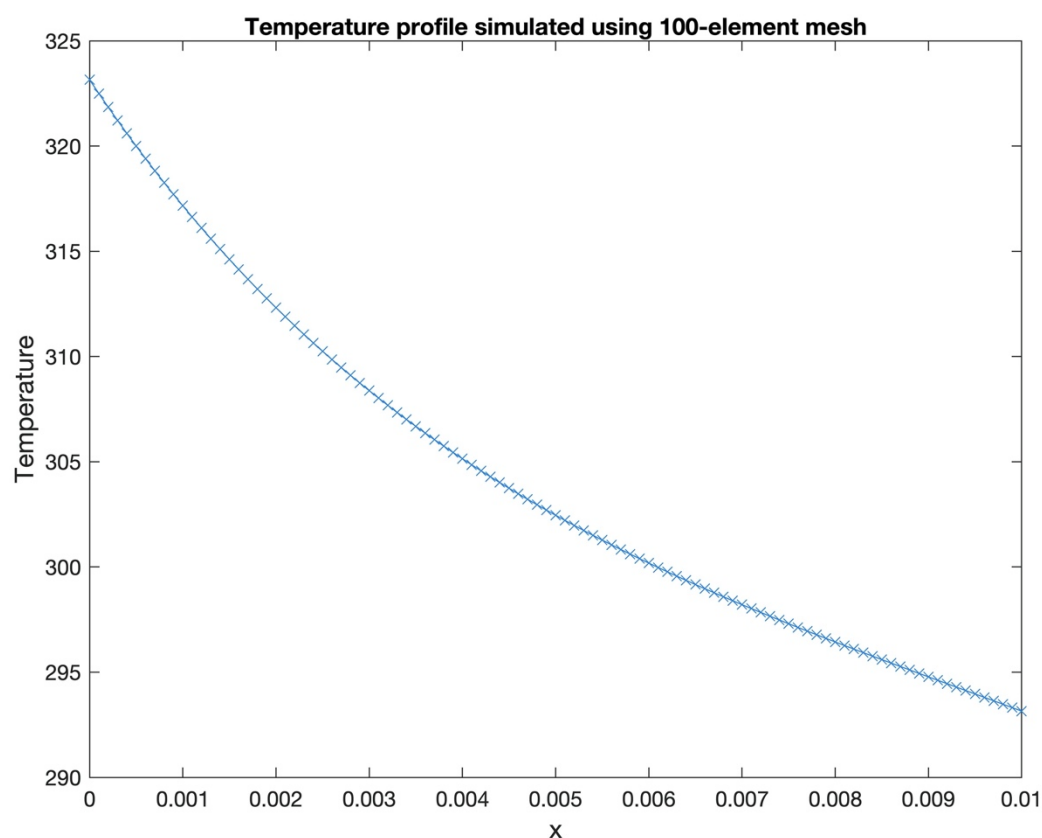


Figure 8: temperature profile as simulated

- Plot the results from your parameter space study and explain the effect each parameter has on both the temperature distribution and on the gradients of the temperature.

Increasing the flow rate increases the rate of temperature transfer between the fluid and the material. This has the effect of moving the temperature at all

points to be closer to the fluid temperature. Choosing a fluid temperature midway between  $T_{\text{inner}}$  and  $T_{\text{outer}}$  (307K for example) shows this effect (Figure 9), with a high value of  $Q$  showing the effect most strongly.

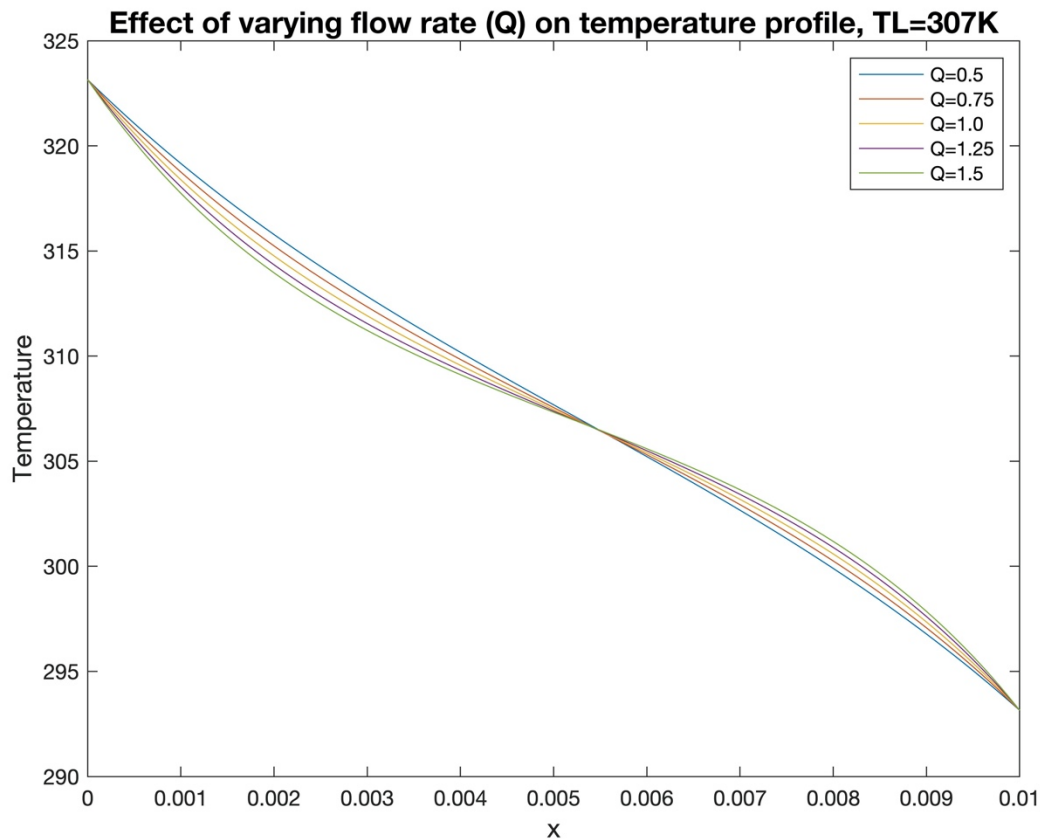


Figure 9: Simulation results at  $Q=0.5$  and  $Q=1.5$

A low value of  $Q$  leads to a more constant gradient between the two boundary conditions, as the diffusion term of the diffusion-reaction equation dominates. The diffusion term leads to a constant gradient if no other term is present, as shown in Part 1. As  $Q$  is increased, the gradient in the middle of the material is shallower, while the gradient at the ends is steeper. This is because the source term of the equation is greater. Physically, a higher flow rate accelerates the heat transfer between the material and the fluid, accentuating the effect of moving all points closer to  $T_L$ .

Because a high value of  $Q$  gives a stronger effect from the fluid,  $Q$  was set to 1.5 while the  $T_L$  was varied from 294.15K to 322.15K. The results of this test are shown in Figure 10.

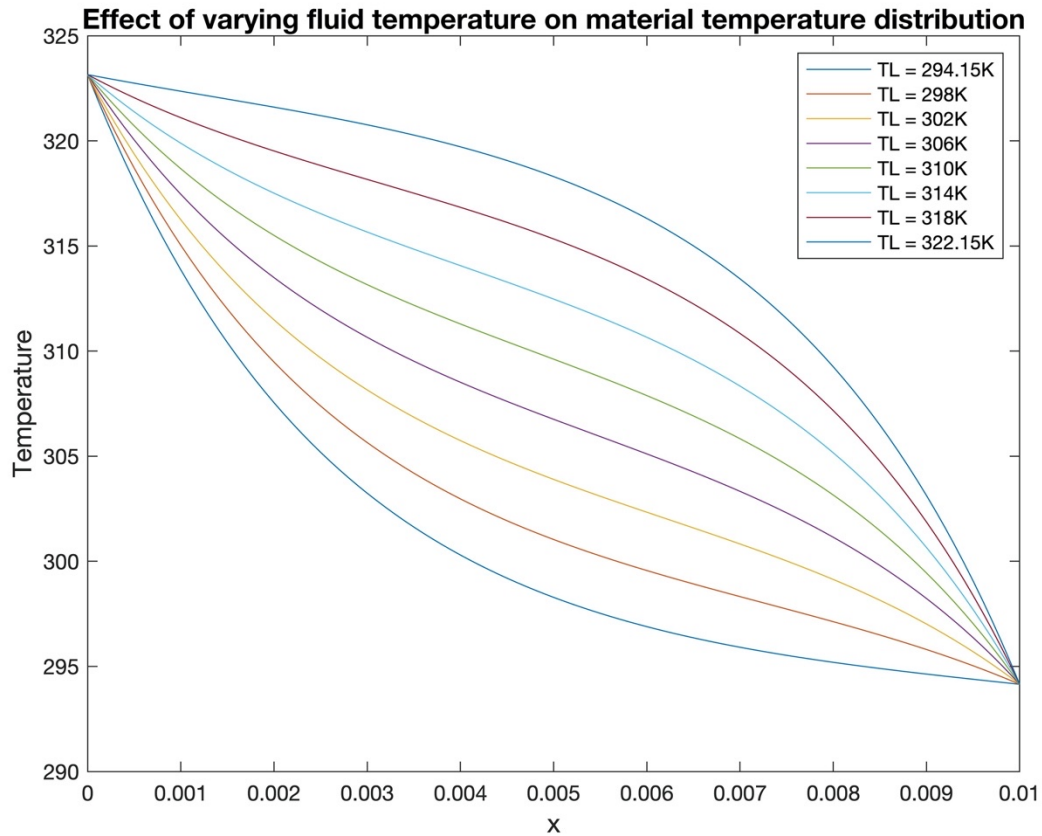


Figure 10: Graph showing temperature profiles for varying fluid temperatures, with  $Q = 1.5$

A high value of  $T_L$  leads to a steeper gradient at the cooler inside of the material, because the difference between  $T$  and  $T_L$  is greater, accelerating heat transfer. Similarly, a low value of  $T_L$  leads to a steeper gradient at the hotter outside of the material, for the same reason.

Higher values of  $T_L$  lead to higher overall temperatures throughout the material.

Figure 11 shows that the effect is the same, but less pronounced, with a low value of  $Q$  (0.5).

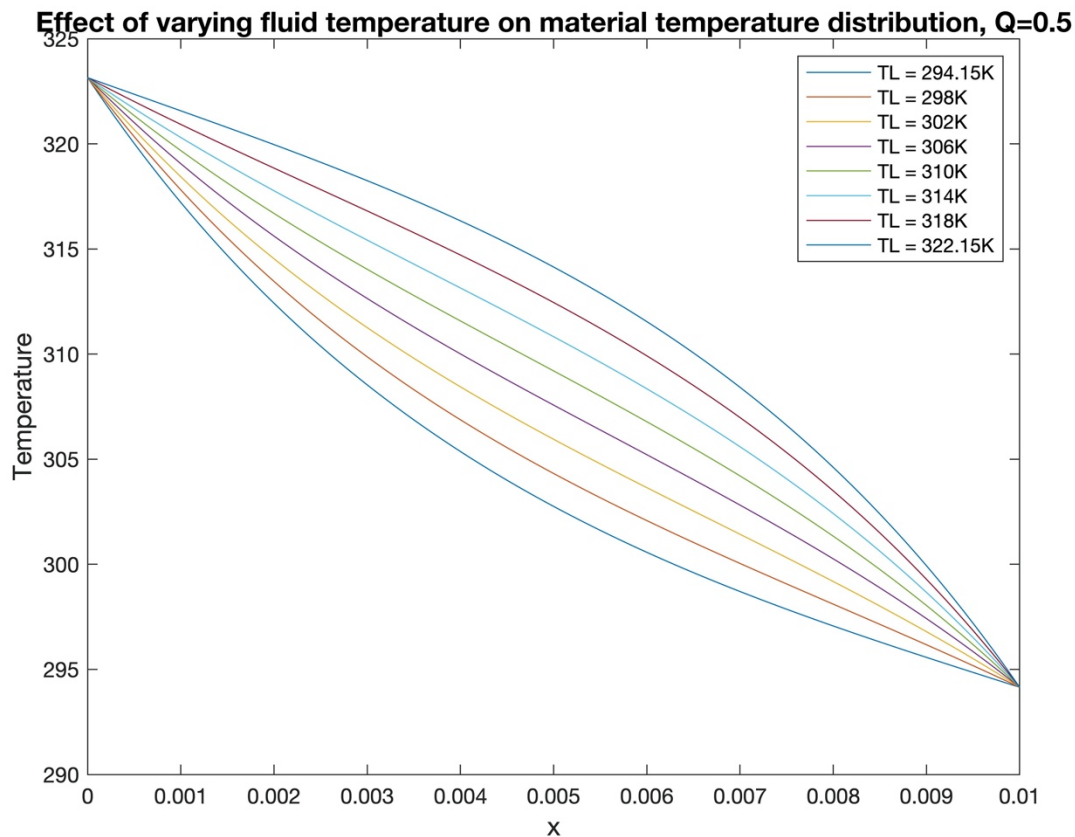


Figure 11: Graph showing temperature profiles for varying fluid temperatures, with  $Q = 0.5$

- ii. If some particles are initially placed at  $x=0$  and are allowed to diffuse for a short amount of time, describe how you think their distribution, and hence the variation in stiffness, might change with  $Q$  and  $T_L$ , based on the effect these have on the temperature gradients in the material?

A low value of  $T_L$  leads to a steeper gradient at the outside of the material, causing the diffusion rate of the particles to be higher. This leads to a more even distribution of the particles into the material, so the stiffness will vary less with distance. However, the particles would not be able to diffuse well into the inside of the material, because the gradient towards the inside is very shallow. This would lead to an even distribution of particles towards the outside, but a lack of particles towards the inside. The material stiffness would smoothly change from the outside in, and be almost constant towards the inside.

A high value of  $T_L$  creates a shallow gradient towards the outside of the material. This would have the effect of reducing the diffusion of particles, causing them to remain concentrated at the outside, leading to a material whose stiffness is very different in a layer at the outside, but largely the same through the rest.

In both cases, a higher value of  $Q$  will make the effect more pronounced, while a low value of  $Q$  will lead to less of an effect.

iii. Assess the effect of different mesh resolutions and explain how this affects the accuracy of the results and your conclusions.

The temperature gradient at  $x=0$  is the most important result of the simulation, as this is where the particles are added, so the analysis of the model's accuracy was based on this gradient. The following values were used to give the steepest gradient at  $x=0$ :

- $Q = 1.5$
- $T_L = 294.15\text{K}$

Varying the number of mesh elements from 1 to 1000 using the Matlab script in Appendix 5 gave the result in Figure 12. As shown, the result tends towards a stable value as the error reduces.

Above 100 elements, there is little change in the calculated value, so the accuracy of the results above is satisfactory, and the conclusions are likely to be correct.

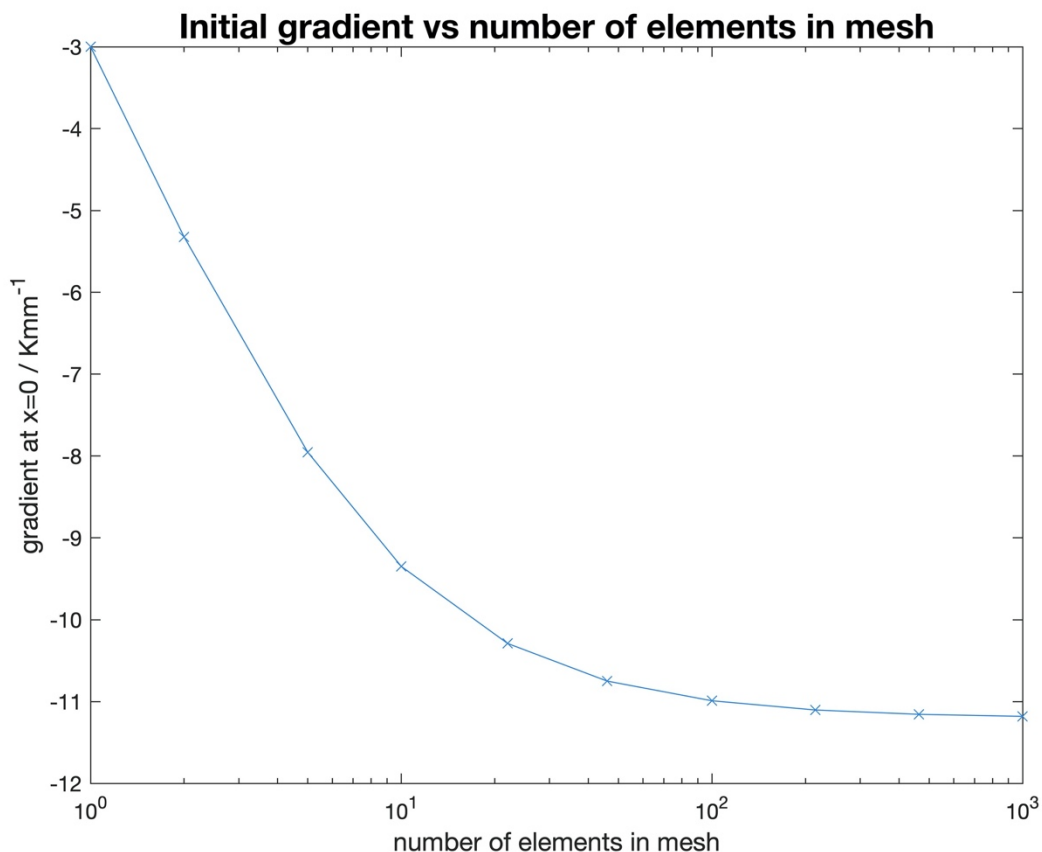


Figure 12: Initial gradient vs mesh resolution

Increasing the number of mesh elements increases the computation time. The computation time of a 1000-element mesh was measured using the following Matlab command:

```
tic; finiteElementSolver(mesh, k, -Q, Q * T_L, [1 T_out; mesh.ngn T_in],[]); toc
```

Elapsed time is 1.024093 seconds.

As shown in the output, even on the largest mesh size tested, the execution time was a little over a second on the author's consumer laptop, so accuracy does not need to be sacrificed by choosing a lower-resolution mesh. However, if computing power was limited or very short execution times were required, a mesh of 100 elements would still give an adequate result.

## b. Varying the Temperature as a Function of x

The temperature could be varied between the heating channels to give more control over the temperature distribution, with the fluid temperature given by the following expression:

$$T_L(1 + 4x)$$

The equation the needs solving is therefore [2]:

$$k \frac{\partial^2 T}{\partial x^2} + Q(T_L(1 + 4x) - T) = 0$$

### i. Derive the analytical expression for the local element vector

Expanding the equation into the form of Equation 1 gives the result:

$$k \frac{\partial^2 T}{\partial x^2} - QT + (QT_L + 4QT_Lx) = 0$$

Comparing with Equation 1 shows that the source term is:

$$f = QT_L + 4QT_Lx$$

The 'Method of Weighted Residuals' was applied: the source term expression was multiplied by the weighting function  $v$  and integrated over the mesh. To solve the equation using the solver, this integral needed to be calculated over each element, yielding

$$\int_{-1}^1 v f J d\xi = \int_{-1}^1 v J (QT_L + 4QT_Lx) d\xi$$

Within each element,  $x$  can be written:

$$x = x_n \psi_n, \quad n = 0, 1$$

The Galerkin Assumption was applied:

$$v = \psi_m, \quad m = 0, 1$$



For each local element, the integral becomes:

$$\int_{-1}^1 QT_L J \psi_m d\xi + \int_{-1}^1 4QT_L x_n \psi_n \psi_m d\xi$$

The first term simplifies to  $QT_L J$  while the second term can be separated into a matrix and a vector, with  $m$  as the row index, and  $n$  as the column index:

$$\begin{bmatrix} Int_{00} & Int_{01} \\ Int_{10} & Int_{11} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$$

Calculating the integrals yields the integral matrix:

$$QT_L J \begin{bmatrix} 8/3 & 4/3 \\ 4/3 & 8/3 \end{bmatrix}$$

- ii. Implement the modified source term in your code and investigate & explain how it changes the behaviour of this system

This system was simulated using a new solver function shown in Appendix 6. With  $Q=1.5$  (to give the most pronounced results),  $T_L$  was varied across the range of possible values. The result of this is shown in Figure 13.

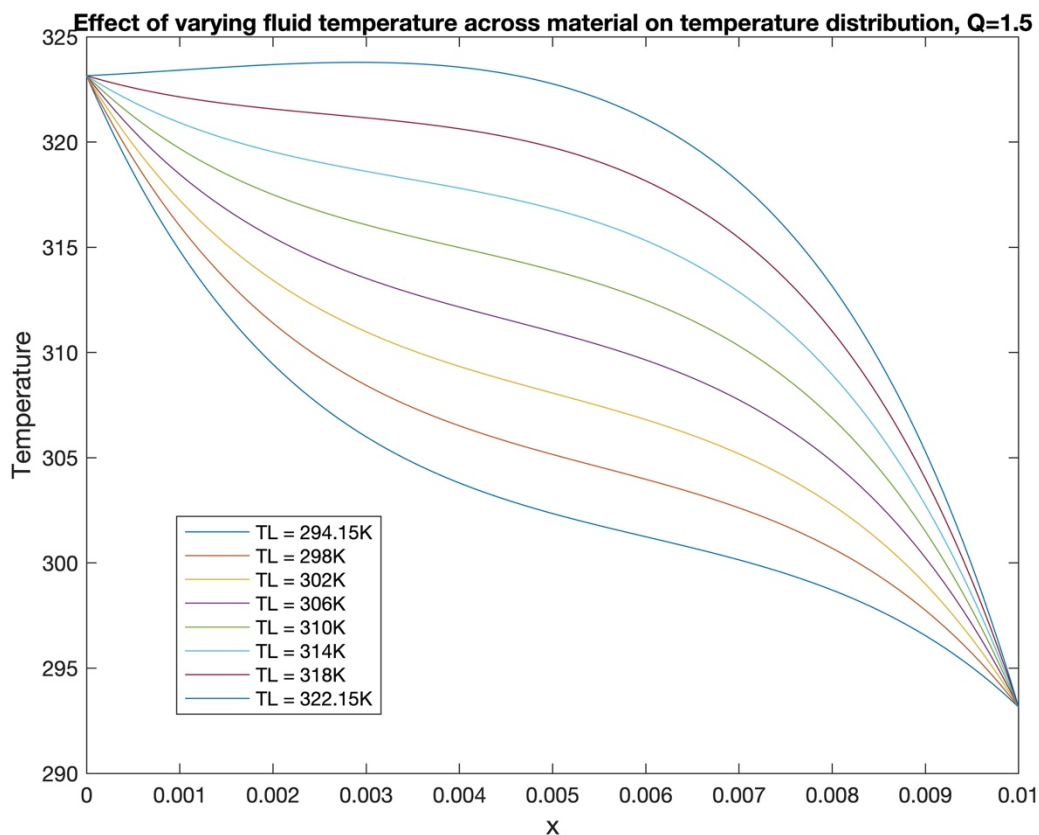


Figure 13: Fluid temperature varying across the material

Varying the fluid temperature as a function of  $x$  has the effect of allowing a steeper gradient at both the inside and the outside. This is because the inside is exposed to a higher temperature than the outside, while the Dirichlet boundary conditions fix the outside to a high temperature and the inside to a low temperature. This has the effect of increasing the difference between the material temperature and the fluid temperature at both sides, accelerating heat transfer.

The steeper gradients could be useful if particles were added to both the inside and the outside of the material, as they would diffuse further in, creating a smoother variation in material stiffness.

At the highest value of  $T_L$ , the gradient at the outside is positive, so particles would not diffuse into the material, instead accumulating on the surface.

The effect of a variable fluid temperature is contrasted with a constant temperature in Figure 14. For the same gradient at the outside, a steeper gradient has been achieved at the inside.

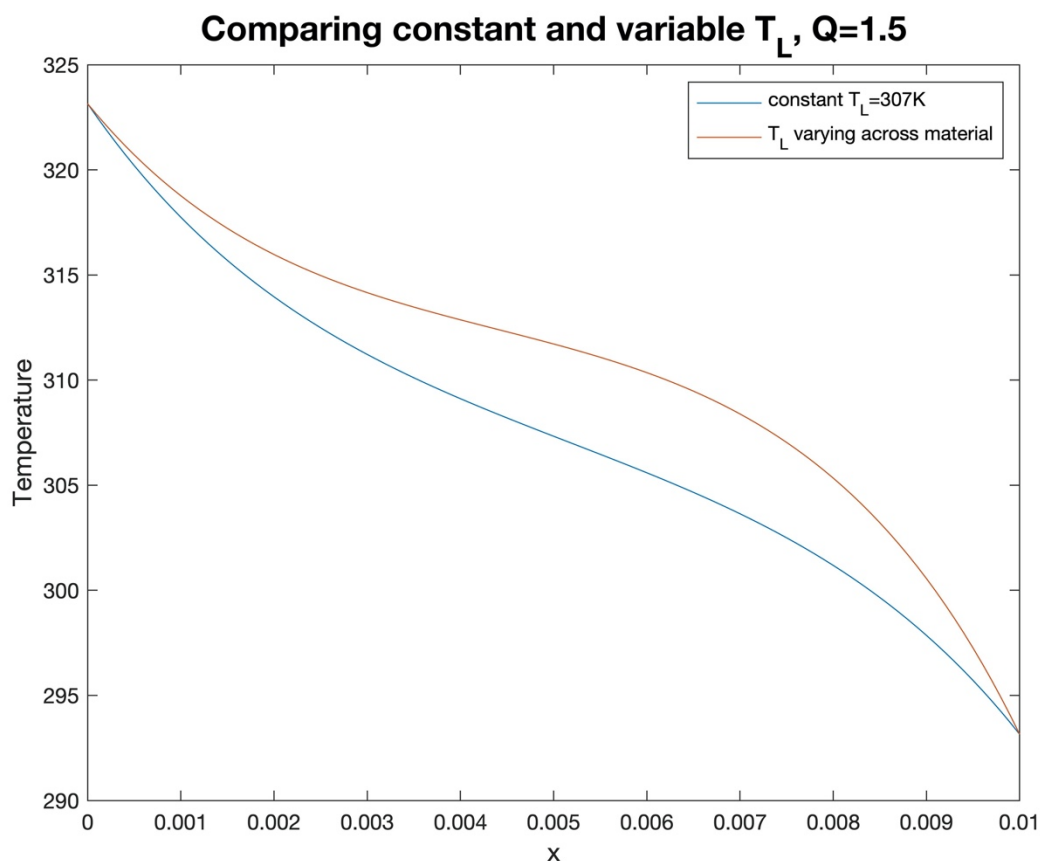


Figure 14: Constant vs variable fluid temperature

## Appendix 1 LaplaceElemMatrix.m file contents

```
function [diffusionMatrix] = LaplaceElemMatrix(D,eID,msh)
%LaplaceMatrix Generates local element matrix for the diffusion operator
% Returns a 2x2 matrix for the diffusion term of the diffusion-reaction
% equation

% verify that element eID is in the mesh
if eID > msh.ne
    error('mesh element ID is not in mesh')
end

% get Jacobian for this element
J = msh.elem(eID).J;
dXi_by_dx = 1 / J;

% create empty matrix
diffusionMatrix = zeros(2);

%% fill empty matrix

% m is the row index
for m = 0:1
    % n is the column index
    for n = 0:1
        % calculate element
        element = D * get_dPsi_by_dXi(n) * ...
            dXi_by_dx * get_dPsi_by_dXi(m) * ...
            dXi_by_dx * J * 2;

        %
        % x2 to perform integral from -1 to 1

        % get matrix indices
        i = m+1;
        j = n+1;

        % put element in the matrix
        diffusionMatrix(i,j) = element;
    end
end
end
```

### get\_dPsi\_by\_dXi.m file contents

```
function [dPsi_by_dXi] = get_dPsi_by_dXi(n)
%GET_DPSI_BY_DXI Get the differential of basis function Psi_n

switch n
case 0
    dPsi_by_dXi = -0.5;
case 1
    dPsi_by_dXi = 0.5;
otherwise
    error('invalid basis function')
end
end
```

## Appendix 2 reactionElemMatrix.m file contents

```
function reactionMatrix = reactionElemMatrix(lambda, eID, msh)
%REACTIONELEMATRIX Generates reaction element matrix
% Generates the reaction element matrix for element 'eID' of mesh 'msh'
% given a reaction coefficient 'lambda'

% verify that element eID is in the mesh
if eID > msh.ne
    error('mesh element ID is not in mesh')
end

% get Jacobian for this element
J = msh.elem(eID).J;

% create empty matrix
reactionMatrix = zeros(2);

% fill the matrix
for m = 0:1
    for n = 0:1

        % get basis functions
        psi_m = getPsi(m);
        psi_n = getPsi(n);

        % construct function and calculate integral
        func = @(xi) lambda .* psi_m(xi) .* psi_n(xi) .* J;
        elem = integral(func, -1, 1);

        % put integral in matrix
        i = m + 1; j = n + 1;
        reactionMatrix(i, j) = elem;
    end
end
end
```

### getPsi.m file contents

```
function [psi] = getPsi(n)
%GETPSI Returns the basis function psi_n where n = {0, 1}
switch n
case 0
    psi = @(xi) (1 - xi) / 2;
case 1
    psi = @(xi) (1 + xi) / 2;
otherwise
    error('psi%d is not a valid function', n)
end
```

## Appendix 3 finiteElementSolver.m file contents

```
function cValues = finiteElementSolver(mesh, D, lambda, f, dirichletBCs,
neumannBCs)
%finiteElementSolver Solves a 1D finite-element problem

%% initialise global matrix and source vector
globalMatrix = zeros(mesh.ngn);
globalSrcVector = zeros(mesh.ngn, 1); % source vector

%% loop over mesh elements
for i = 1:mesh.ne

    % calculate local Laplace matrix
    laplaceMatrix = LaplaceElemMatrix(D, i, mesh);
    % add it to global matrix
    globalMatrix(i:i+1, i:i+1) = globalMatrix(i:i+1, i:i+1)...
        + laplaceMatrix;

    % calculate local reaction matrix
    reactionMatrix = reactionElemMatrix(lambda, i, mesh);
    % subtract it from global matrix
    globalMatrix(i:i+1, i:i+1) = globalMatrix(i:i+1, i:i+1)...
        - reactionMatrix;

    % calculate local source vector and add to global vector
    sourceVector = sourceElemVector(f, i, mesh);
    globalSrcVector(i:i+1) = globalSrcVector(i:i+1) + sourceVector;
end

%% apply neumann boundary conditions
% each condition is in the format [nID, grad], where grad is the known
% gradient at the node nID
for condition = neumannBCs.' % loop over rows

    % get required gradient and node ID
    nID = condition(1);
    grad = condition(2);

    switch nID
        % BC is at the start of the mesh
        case 1
            globalSrcVector(nID) = globalSrcVector(nID) - grad;

        % BC is at the end of the mesh
        case mesh.ngn
            globalSrcVector(nID) = globalSrcVector(nID) + grad;

        otherwise
            error("can only enforce Neumann BC at mesh ends");
    end
end

%% apply dirichlet boundary conditions
% each condition is in the format [nID, c], where c is the known
% concentration at the node nID
for condition = dirichletBCs.'
```

```

    % get required concentration and node ID
    nID = condition(1);
    c    = condition(2);

    % get the identity matrix for the number of nodes
    I = eye(mesh.ne + 1);

    % set corresponding matrix row to a slice of the identity matrix
    globalMatrix(nID, :) = I(nID, :);

    % over-write the corresponding element in the globalVector
    globalSrcVector(nID) = c;
end

%% calculate c vector using  $c = M^{-1} f$ 
cValues = globalMatrix \ globalSrcVector;
end

```

## sourceElemVector.m File Contents

```

function sourceVector = sourceElemVector(f, eID, msh)
%sourceVector Returns a vector representing the source term for element
%'eID' in mesh 'msh' given source coefficient 'f'

    % verify that element eID is in the mesh
    if eID > msh.ne
        error('mesh element ID is not in mesh')
    end

    % get Jacobian for this element
    J = msh.elem(eID).J;

    % create 2x1 source vector with both elements = fJ
    sourceVector = ones(2, 1) * f * J;

```

## Appendix 4 diffReactAccuracy.m File Contents

```
% define analytical solution
anal = @(x) ( (exp(3)/(exp(6)-1)) * (exp(3*x)-exp(-3*x)) );

% get accuracy of estimated results

% set parameters
D      = 1;
lambda = -9;
f      = 0;

% test between 1 and 100 mesh elements (generates 10 numbers)
elementNumbers = round(logspace(0, 2, 10));
inaccuracies   = zeros(1,10); % empty array for inaccuracy scores

for i = 1:length(elementNumbers)

    % create mesh object
    mesh = OneDimLinearMeshGen(0,1,elementNumbers(i));

    % get estimated results
    results = finiteElementSolver(mesh,D,lambda,f,[1 0;mesh.ngn 1],[]);

    % generate 1000 x-points to test at
    xPoints = linspace(0,1,1000);

    % query analytical solution and results vector
    correct = anal(xPoints);
    attempt = interp1(mesh.nvec,results,xPoints);

    % get root of error squared
    error = sqrt((attempt - correct).^2);

    % save inaccuracy score
    inaccuracies(i) = sum(error) / 1000;
end

%% plot results
plot(elementNumbers, inaccuracies, '-x', 'Color', '#2e83dd');
set(gca, 'XScale', 'log');
set(gca, 'YScale', 'log');
xlabel('number of elements in mesh', 'FontSize', 12)
ylabel('inaccuracy as defined in report', 'FontSize', 12)
title('Inaccuracy of simulated result vs number of elements in mesh');
```

## Appendix 5 tempAccuracy.m File Contents

```
%% get estimated gradients

% set parameters
k      = 1.0100e-05;
Q      = 1.5;
T_L    = 294.15;
T_in   = 293.15;
T_out  = 323.15;

% test between 1 and 1000 mesh elements (generates 10 numbers)
elementNumbers = round(logspace(0, 3, 10));
results        = zeros(1,10); % empty array for gradient values

for i = 1:length(elementNumbers)

    % create mesh object
    mesh = OneDimLinearMeshGen(0,0.01,elementNumbers(i));

    % get estimated gradient at x = 0 (node 1)
    tempDist = finiteElementSolver(mesh, k, -Q, Q * T_L, ...
                                   [1 T_out; mesh.ngn T_in],[]);
    gradient = (tempDist(2) - tempDist(1)) / (mesh.nvec(2) - mesh.nvec(1));
    results(i) = gradient;
end

%% plot results
% /1000 to get K/mm
plot(elementNumbers, results/1000, '-x', 'Color', '#2e83dd');
set(gca, 'XScale', 'log');
%set(gca, 'YScale', 'log');
xlabel('number of elements in mesh', 'FontSize', 12)
ylabel('gradient at x=0 / Kmm^{-1}', 'FontSize', 12)
title('Initial gradient vs number of elements in mesh', 'FontSize', 16);
```



## Appendix 6 finiteElementSolver2.m File Contents

```
function cValues = finiteElementSolver2(mesh, D, lambda, Q, T_L,
dirichletBCs, neumannBCs)
%finiteElementSolver2 Solves a 1D finite element problem with a source
%that's a function of x

%% initialise global matrix and source vector
globalMatrix = zeros(mesh.ngn);
globalSrcVector = zeros(mesh.ngn, 1); % source vector

%% loop over mesh elements
for i = 1:mesh.ne

    % calculate local Laplace matrix
    laplaceMatrix = LaplaceElemMatrix(D, i, mesh);
    % add it to global matrix
    globalMatrix(i:i+1, i:i+1) = globalMatrix(i:i+1, i:i+1)...
        + laplaceMatrix;

    % calculate local reaction matrix
    reactionMatrix = reactionElemMatrix(lambda, i, mesh);
    % subtract it from global matrix
    globalMatrix(i:i+1, i:i+1) = globalMatrix(i:i+1, i:i+1)...
        - reactionMatrix;

    % calculate constant part of local source vector & add to global
    % vector
    sourceVector = sourceElemVector(Q * T_L, i, mesh);
    globalSrcVector(i:i+1) = globalSrcVector(i:i+1) + sourceVector;

    % calculate variable part of local source vector & add to global
    % vector
    J = mesh.elem(i).J; % get Jacobian of this element
    %hardcoded values for this problem
    varSrcMatrix = Q*T_L*J*[8/3, 4/3; 4/3, 8/3];
    varSrcVector = varSrcMatrix * [mesh.elem(i).x(1);
mesh.elem(i).x(2)];
    globalSrcVector(i:i+1) = globalSrcVector(i:i+1) + varSrcVector;
end

%% apply neumann boundary conditions
% each condition is in the format [nID, grad], where grad is the known
% gradient at the node nID
for condition = neumannBCs.' % loop over rows

    % get required gradient and node ID
    nID = condition(1);
    grad = condition(2);

    switch nID
        % BC is at the start of the mesh
        case 1
            globalSrcVector(nID) = globalSrcVector(nID) - grad;

        % BC is at the end of the mesh
        case mesh.ngn
            globalSrcVector(nID) = globalSrcVector(nID) + grad;
```

```

        otherwise
            error("can only enforce Neumann BC at mesh ends");
        end
    end

    %% apply dirichlet boundary conditions
    % each condition is in the format [nID, c], where c is the known
    % concentration at the node nID
    for condition = dirichletBCs.'

        % get required concentration and node ID
        nID = condition(1);
        c    = condition(2);

        % get the identity matrix for the number of nodes
        I = eye(mesh.ne + 1);

        % set corresponding matrix row to a slice of the identity matrix
        globalMatrix(nID, :) = I(nID, :);

        % over-write the corresponding element in the globalVector
        globalSrcVector(nID) = c;
    end

    %% calculate c vector using  $c = M^{-1} f$ 
    cValues = globalMatrix \ globalSrcVector;
end

```

## Appendix 7 reactionOperatorTest.m File Contents

```
% Test 1: test symmetry of the matrix
% Test that this matrix is symmetric
tolerance = 1e-14;
eID       = 1; % element ID
lambda    = 10; % reaction coefficient
mesh      = OneDimLinearMeshGen(0,1,10);

reactionMatrix = reactionElemMatrix(lambda, eID, mesh);

assert(abs(reactionMatrix(1,2) - reactionMatrix(2,1)) <= tolerance)

%% Test 2: test that one matrix is evaluted correctly
% Test that a known matrix is calculated correctly
tolerance = 1e-14;
eID       = 1; % element ID
lambda    = 10; % reaction coefficient
mesh      = OneDimLinearMeshGen(0,1,3);

attempt = reactionElemMatrix(lambda, eID, mesh);

correct = [lambda / 9 , lambda / 18; ...
           lambda / 18, lambda / 9];

diff = attempt - correct;
diffnorm = sum(sum(diff.*diff)); % total squared error
assert(abs(diffnorm) <= tolerance)

%% Test 3: test that the same matrix is produced for same-sized elements
tolerance = 1e-14;
eID       = 2;
lambda    = 9;
mesh      = OneDimLinearMeshGen(0,1,4);

matrix2 = reactionElemMatrix(lambda, eID, mesh);

eID = 3;
matrix3 = reactionElemMatrix(lambda, eID, mesh);

% check each element of the matrix is the same
for i=1:2
    for j=1:2
        assert(abs(matrix2(i,j)-matrix3(i,j)) <= tolerance)
    end
end
```

## Appendix 8 fullSolverTest.m File Contents

```
%% Test 1: test solution to laplace's equation with dirichlet BCs
tolerance = 1e-14; % tolerance for comparing floating-point values
D         = 1;      % diffusion coefficient
mesh      = OneDimLinearMeshGen(0,1,4); % create mesh object

% get results
c = finiteElementSolver(mesh,1,0,0,[1 2; mesh.ngn 0],[]);

% check against analytical solution
anal = @(x) 2*(1-x); % analytical solution
for i = 1:mesh.ngn % loop over mesh nodes

    % check solutions are the same
    assert(abs(c(i)-anal(mesh.nvec(i)))<=tolerance);
end

%% Test 2: test neumann boundary condition at x=0
tolerance      = 1e-14; % tolerance for comparing floating-point values
D              = 1;      % diffusion coefficient
mesh           = OneDimLinearMeshGen(0,1,4); % create mesh object
correctGradient = 2;      % required neumann BC

% get results
c = finiteElementSolver(mesh,1,0,0,[mesh.ngn 0],[1 correctGradient]);

% get gradient at x=0
gradient = (c(2) - c(1)) / (mesh.nvec(2) - mesh.nvec(1));
assert(abs(gradient - correctGradient) <= tolerance);

%% Test 3: test neumann boundary condition at x=1
tolerance      = 1e-14; % tolerance for comparing floating-point values
D              = 1;      % diffusion coefficient
mesh           = OneDimLinearMeshGen(0,1,4); % create mesh object
correctGradient = 2;      % required neumann BC

% get results
c = finiteElementSolver(mesh,1,0,0,[1 0],[mesh.ngn correctGradient]);

% get gradient at x=0
gradient = (c(2) - c(1)) / (mesh.nvec(2) - mesh.nvec(1));
assert(abs(gradient - correctGradient) <= tolerance);

%% Test 4: test source vector contains two values
tolerance = 1e-14; % tolerance for comparing floating-point values
f         = 9;      % source coefficient
mesh      = OneDimLinearMeshGen(0,1,4); % create mesh object

sourceVector = sourceElemVector(f,1,mesh);

assert(length(sourceVector) == 2);

%% Test 5: test source vector elements are equal
tolerance = 1e-14; % tolerance for comparing floating-point values
f         = 9;      % source coefficient
mesh      = OneDimLinearMeshGen(0,1,4); % create mesh object
```

```

sourceVector = sourceElemVector(f,1,mesh);

assert(abs(sourceVector(1)-sourceVector(2)) <= tolerance);

%% Test 6: test that source vector values are correct for known solution
tolerance = 1e-14; % tolerance for comparing floating-point values
f          = 9;      % source coefficient
mesh       = OneDimLinearMeshGen(0,1,4); % create mesh object

sourceVector = sourceElemVector(f,1,mesh);
knownVector  = [1.125; 1.125];

for i = 1:length(sourceVector)
    assert(abs(sourceVector(i) - knownVector(i)) <= tolerance);
end

%% Test 7: test that d_psi_by_d_xi gives the correct values
tolerance = 1e-14; % tolerance for comparing floating-point values

% get values
d_psi_by_d_xi_0 = get_dPsi_by_dXi(0);
d_psi_by_d_xi_1 = get_dPsi_by_dXi(1);

% check they're correct
assert(abs(d_psi_by_d_xi_0 + 0.5) <= tolerance);
assert(abs(d_psi_by_d_xi_1 - 0.5) <= tolerance);

```

## Works Cited

- [1] A. Cookson, *CourseworkOneUnitTest.m*, Bath, 2019.
- [2] A. Cookson, "Assignment 1: Static MATLAB-Based FEM Modelling," 2019. [Online]. Available: [https://moodle.bath.ac.uk/pluginfile.php/986615/mod\\_folder/content/0/Assignment1-2019-20-ME40064-Deadline-PrintVersion.pdf?forcedownload=1](https://moodle.bath.ac.uk/pluginfile.php/986615/mod_folder/content/0/Assignment1-2019-20-ME40064-Deadline-PrintVersion.pdf?forcedownload=1). [Accessed 7 11 2019].