Chapter 4

Computer Implementation for 1D and 2D Problems

In this chapter MATLAB codes for 1D and 2D problems are provided. Also a manual for 2D mesh generator is given.

4.1 MATLAB Code for 1D FEM (steady1D.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                                steady1D.m                                     %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                   Middle East Technical University                            %
%                 Department of Mechanical Engineering                          %
%            ME 582 Finite Element Analysis in Thermofluids                     %
%               http://www.me.metu.edu.tr/courses/me582                         %
%                                                                               %
%  Author       : Dr. Cuneyt Sert                                               %
%                 csert@metu.edu.tr                                             %
%                 http://www.me.metu.edu.tr/people/cuneyt                       %
%  Last Update  : 19/03/2012                                                    %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                        FEATURES and LIMITATIONS                               %
%                                                                               %
% This code                                                                     %
% - is written for educational purposes.                                       %
% - solves the following model DE in a 1D domain.                              %
%                                                                               %
%          - d/dx [a(x) du/dx] + b(x) du/dx + c(x) * u = f(x)                   %
%                                                                               %
%   with u(x) being the scalar unknown.                                         %
% - uses Galerkin Finite Element Method (GFEM).                                 %
% - uses linear or quadratic, Lagrange type elements.                           %
% - automatically generates a mesh with clustering at the boundaries.           %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                                HOW TO USE?                                    %
%                                                                               %
% Modify setupProblem function and run the code.                                %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                                VARIABLES                                      %
%                                                                               %
% NE            : Number of elements                                            %
% NN            : Number of nodes in the mesh                                   %
% NEN           : Number of element nodes. Same for all elements                %
% NGP           : Number of GQ points                                           %
% xMin          : Minimum x value of the problem domain                         %
% xMax          : Maximum x value of the problem domain                         %
% funcs         : Structure to hold a(x), b(x), c(x), f(x) as strings           %
% exactSol      : String that holds the exact solution                          %
```

$$-\frac{d}{dx}\left(a\frac{du}{dx}\right)+b\frac{du}{dx}+cu=f$$

```
% isClustered  : 0: Equispaced elements, 1: Non-equispaced elements     %
% clusterValue : Negative values cluster elements at xMin, positive     %
%                values cluster elements at xMax                        %
% coord        : Array of length NN to hold the coordinates of mesh     %
%                nodes                                                  %
% bcType       : Array of size 2 for two BCs                            %
%                1: EBC, 2: NBC, 3: MBC                                 %
% bcValue      : Matrix of size 2x2. For EBC and NBC only one value     %
%                is necessary. For mixed BC two values (alpha and       %
%                beta) are necessary                                    %
% GQ           : Structure to store GQ points and weights               %
% S            : Shape functions evaluated at GQ points. Matrix of size %
%                NENxNGP                                                %
% dS           : Derivatives of shape functions wrt to ksi evaluated at %
%                GQ points. Matrix of size NENxNGP                      %
%                                                                       %
% elem         : Structure for elements                                 %
%   - LtoG     : Local to global node mapping array of size NEN         %
%   - Ke       : Elemental stiffness matrix of size NENxNEN             %
%   - Fe       : Elemental force vector of size NEN                     %
%                                                                       %
% soln         : Structure for the global system of equations           %
%   - K        : Global stiffness matrix of size NNxNN                  %
%   - F        : Global force vector of size NN                         %
%   - U        : Global unknown vector of size NN                       %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
%                               FUNCTIONS                               %
%                                                                       %
% steady1D      : Main function                                         %
% setupProblem  : User provides necessary input values here             %
% generateMesh  : Generate 1D mesh                                      %
% setupLtoG     : Generate LtoG for each element                        %
% setupGQ       : Generate variables for GQ points and weights          %
% calcShape     : Evaluate shape functions and their derivatives at     %
%                 GQ points                                             %
% calcGlobalSys : Calculate global K and F                              %
% calcElemSys   : Calculate elemental Ke and Fe                         %
% assemble      : Assemble elemental systems into a global system       %
% applyBC       : Apply BCs to the global system of equations           %
% solve         : Solve the global system of equations                  %
% postProcess   : Plot the approximate solution. Caculate the SV's at   %
%                 the EBC and MBC boundaries                            %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%==========================================================================
function [] = steady1D
%==========================================================================
clc;              % Clear MATLAB's command window
clear all;        % Clear variables and functions in the memory
close all;        % Close figure windows

disp('************************************************');
disp('******      ME 582 - 1D, Steady Solver      ******');
disp('************************************************');

setupProblem();
generateMesh();

timeStart = tic;                  % Start measuring time.

setupLtoG();
setupGQ();
calcShape();
calcGlobalSys();
applyBC();
solve();

timeTotal = toc(timeStart);       % Stop measuring time.

postProcess();

fprintf('\nTotal run time is = %f seconds.\n', timeTotal);
fprintf('Program is terminated successfully.\n');

% End of function steady1D()
```

This is the main function that calls other functions

```
%==========================================================================
function [] = setupProblem
%==========================================================================
global funcs exactSol NE NEN xMin xMax isClustered clusterValue NGP
global bcType bcValue;

% Remember that the DE we are solving is
% - d/dx [a(x) du/dx] + b(x) du/dx + c(x) * u = f(x)

funcs.a = '1.0';     % a(x)
funcs.b = '3.0';     % b(x)
funcs.c = '0.0';     % c(x)
funcs.f = '1.0';     % f(x)

% Provide '?' if exact solution is not known.
exactSol = 'x/3 - exp(3*x)/(3*exp(3) - 3) - (exp(3) - 4)/(9*exp(3) - 9) + 1/9';

NE           = 5;
NEN          = 2;    % Only 2 and 3 are supported.
xMin         = 0.0;
xMax         = 1.0;
isClustered  = 0;    % Provide 0 to generate equi-sized elements.
clusterValue = 0;    % A positive/negative value will cluster the elements
                     % towards the positive/negative x axis.
NGP          = 3;

% Note that mixed BCs are provided as SV = alpha * PV + beta
bcType(1)    = 1;      % BC type at xMin. 1: EBC, 2: NBC, 3: mixed BC
bcType(2)    = 1;      % BC type at xMax. 1: EBC, 2: NBC, 3: mixed BC

bcValue(1,1) = 0.0;    % Value of PV or SV at xMin. Alpha value for a mixed BC
bcValue(1,2) = 0.0;    % Beta value, if this is a mixed BC

bcValue(2,1) = 0.0;    % Value of PV or SV at xMax. Alpha value for a mixed BC
bcValue(2,2) = 0.0;    % Beta value, if this is a mixed BC

% End of function setupProblem()
```

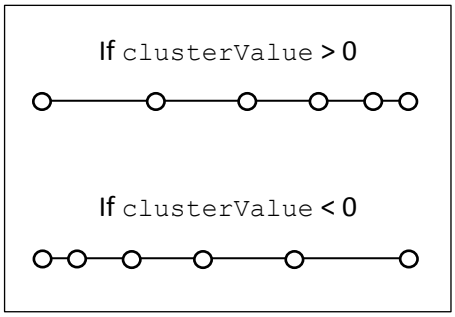
Problem of Section 2.10

$$-\frac{d^2u}{dx^2} + 3\frac{du}{dx} = 1\,, \qquad x \in [0,1]$$

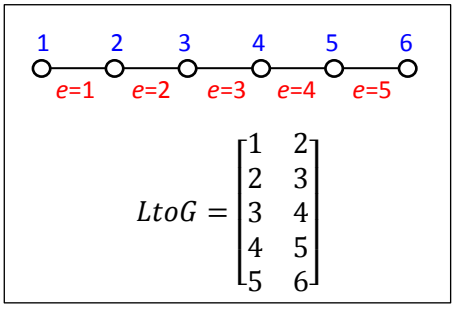
$$u(0) = 0\,, \qquad u(1) = 0$$

Known exact solution

```
%=========================================================================
function [] = generateMesh
%=========================================================================
global NE NN NEN xMax xMin isClustered clusterValue coord;

% Determine number of mesh nodes
NN = NE * (NEN - 1) + 1 ;

% Calculate nodal coordinates
range = xMax - xMin;

if isClustered == 0 || clusterValue == 0    % No clustering
  dx = range / (NN-1);
  for i = 1:NN
    coord(i) = xMin + (i-1)*dx;
  end
else
  MAX = sinh(abs(clusterValue));   % Nonlinear sinh function is used for
  for e = 1:NE+1                   % clustering
    xx = (1.0 * (e-1)) / NE;                % 0 < xx < 1
    xxx = sinh(abs(clusterValue) * xx);     % 0 < xxx < MAX
    if (clusterValue < 0)
      dxE(e) = range/MAX * xxx;
    else
      dxE(NE+2-e) = range - range/MAX * xxx;
    end
  end

  % Coordinates of end-nodes of elements
  for e = 1:NE
    first = (e-1)*(NEN-1) + 1;
    coord(first) = dxE(e);
    coord(first + NEN - 1) = dxE(e+1);
  end

  % Coordinates of mid-nodes of elements (for elements with NEN > 2)
  for e = 1:NE
    first = (e-1)*(NEN-1) + 1;
    dE = dxE(e+1) - dxE(e);  % Elements length
    for i = 1:NEN-1
      coord(first + i) = coord(first) + dE /(NEN-1)*i;
    end
  end
end

% End of function generateMesh()
```




```
%=========================================================================
function [] = setupLtoG
%=========================================================================
global NE NEN elem;

% Setup LtoG for each element. We assume that nodes are globally numbered
% consecutively starting from the left boundary node at xMin.
for e = 1:NE
  for i = 1:NEN
    elem(e).LtoG(i) = (e-1)*(NEN-1) + i;
  end
end

% End of function setupLtoG()
```



```
%==========================================================================
function [] = setupGQ
%==========================================================================
global NGP GQ;

if NGP == 1                   % 1-point quadrature
  GQ.point(1)  = 0;
  GQ.weight(1) = 2;
elseif NGP == 2               % 2-point quadrature
  GQ.point(1)  = -sqrt(1/3);
  GQ.point(2)  = sqrt(1/3);
  GQ.weight(1) = 1;
  GQ.weight(2) = 1;
elseif NGP == 3               % 3-point quadrature
  GQ.point(1)  = -sqrt(3/5);
  GQ.point(2)  = 0;
  GQ.point(3)  = sqrt(3/5);
  GQ.weight(1) = 5/9;
  GQ.weight(2) = 8/9;
  GQ.weight(3) = 5/9;
end

% End of function setupGQ()
```

Gauss Quadrature points and weights are coming from Table 2.1

```
%==========================================================================
function calcShape()
%==========================================================================
% Calculates the values of the shape functions and their derivatives with
% respect to ksi coordinate at GQ points.
global NEN NGP S dS GQ;

S  = zeros(NEN, NGP);
dS = zeros(NEN, NGP);

if NEN == 2  % Linear Lagrange shape functions
  for k = 1:NGP
    ksi = GQ.point(k);
    S(1,k)  = 0.5 * (1 - ksi);
    S(2,k)  = 0.5 * (1 + ksi);
    dS(1,k) = -0.5;
    dS(2,k) = 0.5;
  end
elseif NEN == 3  % Quadratic Lagrange shape functions
  for k = 1:NGP
    ksi = GQ.point(k);
    S(1,k)  = 0.5 * ksi * (ksi - 1);
    S(2,k)  = 1 - ksi * ksi;
    S(3,k)  = 0.5 * ksi * (1 + ksi);
    dS(1,k) = -0.5 + ksi;
    dS(2,k) = -2 * ksi;
    dS(3,k) = 0.5 + ksi;
  end
end

% End of function calcShape()
```

Equation 2.20 (NEN=2)

$$S_1 = \frac{1}{2}(1-\xi) \quad , \quad S_2 = \frac{1}{2}(1+\xi)$$

$$\frac{dS_1}{d\xi} = -0.5 \quad , \quad \frac{dS_2}{d\xi} = 0.5$$

Equation 2.28 (NEN=3)

$$S_1 = \frac{1}{2}\xi(\xi-1) \quad , \quad S_2 = (1-\xi^2) \quad , \quad S_3 = \frac{1}{2}\xi(1+\xi)$$

$$\frac{dS_1}{d\xi} = -0.5 + \xi \quad , \quad \frac{dS_2}{d\xi} = -2\xi \quad , \quad \frac{dS_3}{d\xi} = 0.5 + \xi$$

```
%=========================================================================
function calcGlobalSys()
%=========================================================================
% Calculates the global stiffness matrix [K] and force vector {F}.
global NE NN soln;

soln.K = zeros(NN,NN);
soln.F = zeros(NN,1);    % {F} is a column vector, not a row vector.

for e = 1:NE
  calcElemSys(e);
  assemble(e);
end

% End of function calcGlobalSys()
```

```
%=========================================================================
function [] = calcElemSys(e)
%=========================================================================
% Calculates the element level stiffness matrix and force vector.
global NEN NGP coord GQ elem funcs S dS;

elem(e).Fe = zeros(NEN,1);
elem(e).Ke = zeros(NEN,NEN);

for k = 1:NGP   % Gauss Quadrature loop
  ksi = GQ.point(k);   % k-th GQ point.

  % Calculate the x value inside element e that corresponds to ksi.
  xFirst = coord(elem(e).LtoG(1));   % Coord. of the first node of elem(e)
  xLast = coord(elem(e).LtoG(NEN));  % Coord. of the last node of elem(e)
  x = 0.5 * (xLast - xFirst) * ksi + 0.5 * (xFirst + xLast);

  aValue = eval(funcs.a);    % Evaluate a, b, c and f functions at the x
  bValue = eval(funcs.b);    % value calculated above.
  cValue = eval(funcs.c);
  fValue = eval(funcs.f);

  % Calculate Jacobian of the element
  Jacob = 0.5 * (xLast - xFirst);

  for i = 1:NEN
    for j = 1:NEN
      elem(e).Ke(i,j) = elem(e).Ke(i,j) + ...
                        (aValue * dS(i,k)/Jacob * dS(j,k)/Jacob + ...
                         bValue * S(i,k) * dS(j,k)/Jacob + ...
                         cValue * S(i,k) * S(j,k) ) ...
                         * Jacob * GQ.weight(k);
    end
  end

  for i = 1:NEN
    elem(e).Fe(i) = elem(e).Fe(i) + fValue * S(i,k) * Jacob * GQ.weight(k);
  end

end  % End of GQ loop

% End of function calcElemSys()
```

Equation 2.21

$$x = \frac{h^e}{2}\xi + \frac{x_1^e + x_2^e}{2}$$

$$K_{ij}^e = \int_{-1}^{1} \left(a \frac{dS_i}{d\xi}\frac{1}{J^e}\frac{dS_j}{d\xi}\frac{1}{J^e} + bS_i\frac{dS_j}{d\xi}\frac{1}{J^e} + +cS_iS_j \right) J^e \, d\xi$$

$$F_i^e = \int_{-1}^{1} fS_i J^e \, d\xi$$

```
%=========================================================================
function [] = assemble(e)
%=========================================================================
% Inserts [Ke] and {Fe} into proper locations of [K] and {F} by the help
% of LtoG arrays of elements.

global NEN elem soln;

% Assemble Ke into K.
for i = 1:NEN
  I = elem(e).LtoG(i);      % I is the global node corresponding to the
                            % i-th local node of element e
  for j = 1:NEN
    J = elem(e).LtoG(j);    % J is similar to I.
    soln.K(I,J) = soln.K(I,J) + elem(e).Ke(i,j);
  end
end

% Assemble Fe into F.
for i = 1:NEN
  I = elem(e).LtoG(i);
  soln.F(I) = soln.F(I) + elem(e).Fe(i);
end

% End of function assemble()
```

$K_{ij}^e \rightarrow K_{IJ}$ where $I = Ltog(e,i)$, $J = Ltog(e,j)$

$F_i^e \rightarrow F_I$ where $I = Ltog(e,i)$

```
%=========================================================================
function [] = applyBC()
%=========================================================================
global NN soln bcType bcValue;
% For EBCs reduction is NOT applied. Instead global [K] and {F} are
% modified. SV values specified for NBCs are added to {F}, there is no
% separate {B} vector for boundary integrals. For mixed type BCs both
% [K] and {F} are modified using the provided alpha and beta values.

for i = 1:2   % Loop for two BCs

  % Determine the node for which this BC is specified.
  if i == 1
     node = 1;
  else
     node = NN;
  end

  if bcType(i) == 1             % If this BC is of EBC type
    soln.F(node) = bcValue(i,1);
    soln.K(node,:) = 0.0;     % Equate i-th row of K to zero.
    soln.K(node,node) = 1.0;  % Equate diagonal of the node-th row of K
                              % to 1.

  elseif bcType(i) == 2         % If this BC is of NBC type
    soln.F(node) = soln.F(node) + bcValue(i,1);   % Add given SV value
                                                  % to F.

  elseif bcType(i) == 3         % If this BC is of mixed type
    % Subtract alpha value from the diagonal of the node-th row of K.
    soln.K(node,node) = soln.K(node,node) - bcValue(i,1);
    soln.F(node) = soln.F(node) + bcValue(i,2);    % Add beta value
                                                   % to F.
  end
end

% End of function applyBC()
```

EBCs : Reduction is NOT applied.

NBCs : Given SV is added to the $\{F\}$ vector.

MBCs : Given β is added to the $\{F\}$ vector and given α is subtracted from the $[K]$ matrix.

```
%=========================================================================
function [] = solve()
%=========================================================================
% Solves the system [K]{U}={F}. Note that this is generally the most time
% consuming part of the solution. Backslash operator that we use may be
% inefficient, especially for large problems. There are alternative
% techniques.
global soln;

soln.U = soln.K \ soln.F;

% End of function solve()
```

$$\{U\} = [K]^{-1}\{F\}$$

Note that $\{F\}$ also includes $\{B\}$

```
%=========================================================================
function [] = postProcess()
%=========================================================================
% Writes the calculated nodal unknowns to the screen. Plots the FEM
% solution and the error if the exact solution is known. Calculates the
% SVs at the boundaries where EBC or MBC is specified.
global NN NE NEN coord soln elem exactSol bcType funcs

% Write the calculated unknowns to the screen.
fprintf(1,'\nCalculated unknowns are \n\n');
fprintf(1,' Node              x                    u \n');
fprintf(1,'=============================================\n');
for i = 1:NN
  fprintf(1, ' %-5d %18.8f %20.8f\n', i, coord(i), soln.U(i));
end



% Create plot windows for the FEM solution and the error in case we know
% the exact solution.
if exactSol == '?'
  subplot(1,1,1);  % If exact solution is not known generate a single plot.
else
  subplot(2,1,1);  % If exact solution is known generate two plots. This
                   % one is for the FEM solution, and the other will be
                   % for the error.
end


% Plot the approximate solution as circles.
firstPlot = gca;   % Axes handle for the 1st plot. Will be used later.
plot(coord, soln.U, 'ko');
hold(firstPlot,'on');

if exactSol ~= '?'
  subplot(2,1,2);     % This plot is for the exact error.
  secondPlot = gca;   % Axes handle for the 2nd plot. Will be used later.
  hold(secondPlot,'on');
end



% Plot the solution over each element as lines or curves. For high order
% elements, straight lines are not enough, curves are necessary. Here
% these curves are generated based on np equally spaced points over each
% element.
np = 20;  % Number of points used to plot the solution over an element.
for e = 1:NE
  for i = 1:NEN
    X(i) = coord(elem(e).LtoG(i));    % NEN coordinates of element e.
    Y(i) = soln.U(elem(e).LtoG(i));   % Nodal unknowns at X(i)
  end
```

```
  p = polyfit(X, Y, NEN-1);  % p is the polynomial curve describing the
                             % approximate solution over element e. For an
                             % element with NEN points, the polynomial is
                             % of order NEN-1.
  x = X(1):(X(NEN)-X(1))/np:X(NEN);    % np+1 equally spaced points over
                                       % element e.
  pVal = polyval(p, x);    % Evaluate p at np+1 many points over element e.
  plot(firstPlot, x, pVal, 'k', 'LineWidth', 2);

  % Plot the exact solution and exact error on this element if the exact
  % solution is known.
  if exactSol ~= '?'
    exact = eval(exactSol);
    plot (firstPlot, x, exact, 'r:', 'LineWidth', 2);
    plot(secondPlot, x, exact - pVal);
  end
end



% Add grid lines and titles to the plots
axes(firstPlot);  % Set current axes to the first plot.
grid on;
if exactSol == '?'
  title('GFEM solution of the model DE');
else
  title({'GFEM solution of the model DE'; 'Black: Approx., Red: Exact'});
end

if exactSol ~= '?'
  axes(secondPlot);
  grid on;
  title('Error = Exact solution - Approximate solution');
end



% Calculate SVs at the boundary nodes where EBC or MBC is specified
% Left boundary
if (bcType(1) ~= 2)
  if NEN == 2
    slope = (soln.U(2) - soln.U(1)) / (coord(2) - coord(1));
  else
    slope = (-3 * soln.U(1) + 4 * soln.U(2) - soln.U(3)) / ...
            (coord(3) - coord(1));
  end
  x = coord(1);
  SV = - eval(funcs.a) * slope;    % Minus sign is due to nx = -1
  fprintf(1,'\nSV at the left boundary : %f\n', SV);
end

% Right boundary
if (bcType(2) ~= 2)
  if NEN == 2
    slope = (soln.U(NN) - soln.U(NN-1)) / (coord(NN) - coord(NN-1));
  else
    slope = (-3 * soln.U(NN-2) + 4 * soln.U(NN-1) - soln.U(NN)) / ...
            (coord(3) - coord(1));
  end
  x = coord(1);
  SV = eval(funcs.a) * slope;
  fprintf(1,'\nSV at the right boundary : %f\n', SV);
end

% End of function postProcess()
```

First calculate $\left.\frac{du}{dx}\right|_{left}$

$$SV|_{left} = -\,a \left.\frac{du}{dx}\right|_{left}$$

First calculate $\left.\frac{du}{dx}\right|_{right}$

$$SV|_{right} = a \left.\frac{du}{dx}\right|_{right}$$

4.2 MATLAB Code for 2D FEM (steady2D.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                                steady2D.m                                     %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                    Middle East Technical University                           %
%                  Department of Mechanical Engineering                         %
%             ME 582 Finite Element Analysis in Thermofluids                    %
%                http://www.me.metu.edu.tr/courses/me582                        %
%                                                                               %
%  Author       : Dr. Cuneyt Sert                                               %
%                 csert@metu.edu.tr                                             %
%                 http://www.me.metu.edu.tr/people/cuneyt                       %
%  Last Update  : 19/03/2012                                                    %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                         FEATURES and LIMITATIONS                              %
%                                                                               %
% This code                                                                     %
% - is written for educational purposes.                                       %
% - solves the following model DE in a 2D domain.
%
%     V(x,y) . del(u) - del . [a(x,y) del(u)] + c(x,y) * u = f(x,y)
%
%   where u(x,y) is the scalar unknown,
%         V(x,y) is the known velocity field,                                   %
%         a(x,y), c(x,y) and f(x,y) are known functions.                        %
%                                                                               %
% - uses Galerkin FEM.                                                          %
% - uses Lagrange type, triangular or quadrilateral elements.                   %
% - reads problem data and mesh from an input file.                             %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                                HOW TO USE?                                    %
%                                                                               %
% This code needs an input file with an extension "inp" to run. You can        %
% generate it by hand or use generate2DinputFile.m code. After creating        %
% the input file run this code and provide name of the input file when          %
% asked for.                                                                    %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                                VARIABLES                                      %
%                                                                               %
% prName        : Name of the problem                                           %
% NE            : Number of elements                                            %
% NN            : Number of nodes in the mesh                                   %
% NEN           : Number of element nodes. Same for all elements.               %
%                 Only 4 node quadrilaterals and 3 node triangles are           %
%                 supported.                                                    %
% eType         : 1: Quadrilateral, 2: Triangular. Same for all elements        %
% funcs         : Structure to hold a, V1, V2, c, f functions as strings        %
% coord         : Matrix of size NNx2 to hold the x and y coordinates of        %
%                 the mesh nodes                                                %
% nBC           : Number of different BCs specified.                            %
%                                                                               %
% BC            : Structure for boundary conditions                             %
%  - type       : Array of size nBC. 1: EBC, 2: NBC, 3: mixed.                  %
%  - str        : BC values/functions are stored as strings in this cell        %
%                 array of size nBCx2. Second column is only used for           %
%                 storing beta value of mixed BCs.                              %
%  - nEBC       : Number of nodes where EBC is specified.                       %
%  - nNBC       : Number of element faces where NBC is specified.               %
```

$$\vec{V} \cdot \nabla u - \nabla \cdot (k \nabla u) + cu = f$$

```
%  - nMBC      : Number of element faces where mixed BC is specified.   %
%  - EBCnodes  : Array of size nEBCx2. First column stores global node  %
%                numbers for which EBC is provided. Second column       %
%                stores the specified BC number.                        %
%  - NBCfaces  : Array of nNBCx3. First and second columns store        %
%                element and face numbers for which an NBC is given.    %
%                Last column stores the specified BC number.            %
%  - MBCfaces  : Array of nMBCx3. Works similar to NBCfaces.            %
%                                                                       %
% NGP          : Number of GQ points.                                   %
% GQ           : Structure to store GQ points and weights.              %
% S            : Shape functions evaluated at GQ points. Matrix of size %
%                NENxNGP                                                %
% dS           : Derivatives of shape functions wrt to ksi and eta      %
%                evaluated at GQ points. Matrix of size 2xNENxNGP       %
%                                                                       %
% elem         : Structure for elements                                 %
%  - LtoG      : Local to global node mapping array of size NEN         %
%  - Ke        : Elemental stiffness matrix of size NENxNEN             %
%  - Fe        : Elemental force vector of size NEN                     %
%  - gDS       : Derivatives of shapes functions wrt x and y at GQ      %
%                points. Size is 2xNENxNGP                              %
%  - detJacob  : Determinant of the Jacobian matrix evaluated at a      %
%                certain (ksi, eta)                                     %
%                                                                       %
% soln         : Structure for the global system of equations           %
%  - K         : Global sitffness matrix of size NNxNN                  %
%  - F         : Global force vector of size NN                         %
%  - U         : Global unknown vector of size NN                       %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %
%                               FUNCTIONS                               %
%                                                                       %
% steady2D      : Main function                                         %
% readInputFile : Reads the input file                                  %
% setupGQ       : Generate variables for GQ points and weights          %
% calcShape     : Evaluate shape functions and their ksi and eta        %
%                 derivatives at GQ points                              %
% calcJacob     : Calculate the Jacobian, and its determinant and       %
%                 derivatives of shape functions wrt x and y at GQ      %
%                 points                                                %
% calcGlobalSys : Calculate global K and F                              %
% calcElemSys   : Calculate elemental Ke and Fe                         %
% assemble      : Assemble elemental systems into a global system       %
% applyBC       : Apply BCs to the global system of equations           %
% getFaceDetails: Find nodes of an element's face and calculate the     %
%                 length of the face                                    %
% solve         : Solve the global system of equations                  %
% postProcess   : Generate contour plot of the scalar unknown           %
% calculateSV   : Calculates the SV at the boundaries                   %
% createTecplotOutput : Create an output file for Tecplot software      %
%                                                                       %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%==========================================================================
function [] = steady2D
%==========================================================================
clc;
clear all;
close all;

disp('************************************************');
disp('******      ME 582 - 2D, Steady Solver      ******');
disp('************************************************');

readInputFile();

timeStart = tic;

setupGQ();
calcShape();
calcJacob();
calcGlobalSys();
applyBC();
solve();

timeTotal = toc(timeStart);

postProcess();
createTecplotOutput();

fprintf('\nTotal CPU time is = %f seconds.\n', timeTotal);
fprintf('Program is terminated successfully.\n');

% End of function steady2D()
```

```
%==========================================================================
function [] = readInputFile
%==========================================================================
global prName eType NE NN NEN NGP funcs coord elem nBC BC

% Ask the user for the input file name
prName = input('\nEnter the name of the input file (without the .inp extension): ','s');
inputFile = fopen(strcat(prName,'.inp'), 'r');

fgets(inputFile);  % Read the header line of the file
fgets(inputFile);  % Read the next dummy line.


% Read problem parameters
eType = fscanf(inputFile, 'eType    :%d');    fgets(inputFile);
NE    = fscanf(inputFile, 'NE       :%d');    fgets(inputFile);
NN    = fscanf(inputFile, 'NN       :%d');    fgets(inputFile);
NEN   = fscanf(inputFile, 'NEN      :%d');    fgets(inputFile);
NGP   = fscanf(inputFile, 'NGP      :%d');    fgets(inputFile);

funcs.a  = fscanf(inputFile, 'aFunc    :%s');    fgets(inputFile);
funcs.V1 = fscanf(inputFile, 'V1Func   :%s');    fgets(inputFile);
funcs.V2 = fscanf(inputFile, 'V2Func   :%s');    fgets(inputFile);
funcs.c  = fscanf(inputFile, 'cFunc    :%s');    fgets(inputFile);
funcs.f  = fscanf(inputFile, 'fFunc    :%s');    fgets(inputFile);
```

```
% Read node coordinates
fgets(inputFile);  % Read the next dummy line.
fgets(inputFile);  % Read the next dummy line.
for i = 1:NN
  dummy = str2num(fgets(inputFile));
  coord(i,1) = dummy(2);
  coord(i,2) = dummy(3);
end


% Read connectivity of the elements, i.e. LtoG
fgets(inputFile);  % Read the next dummy line.
fgets(inputFile);  % Read the next dummy line.
for e = 1:NE
   dummy = str2num(fgets(inputFile));
   for i = 1:NEN
      elem(e).LtoG(i) = dummy(1+i);
   end
end


% Read number of different BC types and data for each BC
fgets(inputFile);  % Read the next dummy line.
fgets(inputFile);  % Read the next dummy line.
nBC   = fscanf(inputFile, 'nBC       :%d');      fgets(inputFile);
BC.type = zeros(nBC,1);
BC.str  = cell(nBC,2);

for i = 1:nBC
  fscanf(inputFile, 'BC %*d      :');
  BC.type(i) = fscanf(inputFile, '%d', 1);

  if BC.type(i) == 1 || BC.type(i) == 2  % For EBC and MBC read and store
                                         % only 1 string
    BC.str(i,1) = cellstr(fgets(inputFile));
  else                                   % For mixed BC read 1 string and
                                         % divide it into 2 strings.
    string = fgets(inputFile);
    [str1, str2] = strtok(string, ':');
    [str2, str3] = strtok(str2, ':');

    BC.str(i,1) = cellstr(str1);
    BC.str(i,2) = cellstr(str2);
  end
end


% Read the number of EBC nodes and number of NBC and MBC faces.
fgets(inputFile);  % Read the next dummy line.
BC.nEBC = fscanf(inputFile, 'nEBCnodes :%d');      fgets(inputFile);
BC.nNBC = fscanf(inputFile, 'nNBCfaces :%d');      fgets(inputFile);
BC.nMBC = fscanf(inputFile, 'nMBCfaces :%d');      fgets(inputFile);


% Read EBC data
BC.EBCnodes = zeros(BC.nEBC,2);
fgets(inputFile);  % Read the next dummy line.
fgets(inputFile);  % Read the next dummy line.
for i = 1:BC.nEBC
  BC.EBCnodes(i,1) = fscanf(inputFile, '%d', 1);
  BC.EBCnodes(i,2) = fscanf(inputFile, '%d', 1);
  fgets(inputFile);
end
```

```
% Read NBC data
BC.NBCfaces = zeros(BC.nNBC,3);
fgets(inputFile);  % Read the next dummy line.
for i = 1:BC.nNBC
  BC.NBCfaces(i,1) = fscanf(inputFile, '%d', 1);     % Read element no.
  BC.NBCfaces(i,2) = fscanf(inputFile, '%d', 1);     % Read face no.
  BC.NBCfaces(i,3) = fscanf(inputFile, '%d', 1);     % Read BC no.
  fgets(inputFile);
end


% Read MBC data
BC.MBCfaces = zeros(BC.nMBC,3);
fgets(inputFile);  % Read the next dummy line.
for i = 1:BC.nMBC
  BC.MBCfaces(i,1) = fscanf(inputFile, '%d', 1);     % Read element no.
  BC.MBCfaces(i,2) = fscanf(inputFile, '%d', 1);     % Read face no.
  BC.MBCfaces(i,3) = fscanf(inputFile, '%d', 1);     % Read BC no.
  fgets(inputFile);
end

fclose(inputFile);

% End of function readInputFile()
```

```
%=========================================================================
function [] = setupGQ
%=========================================================================
global eType NGP GQ;

if eType == 1        % Quadrilateral element
  if NGP == 1        % One-point quadrature
    GQ.point(1,1) = 0.0;  GQ.point(1,2) = 0.0;
    GQ.weight(1) = 4.0;
  elseif NGP == 4    % Four-point quadrature
    GQ.point(1,1) = -sqrt(1/3);   GQ.point(1,2) = -sqrt(1/3);
    GQ.point(2,1) = sqrt(1/3);    GQ.point(2,2) = -sqrt(1/3);
    GQ.point(3,1) = -sqrt(1/3);   GQ.point(3,2) = sqrt(1/3);
    GQ.point(4,1) = sqrt(1/3);    GQ.point(4,2) = sqrt(1/3);
    GQ.weight(1) = 1.0;
    GQ.weight(2) = 1.0;
    GQ.weight(3) = 1.0;
    GQ.weight(4) = 1.0;
  elseif NGP == 9    % Nine-point quadrature
    GQ.point(1,1) = -sqrt(3/5);   GQ.point(1,2) = -sqrt(3/5);
    GQ.point(2,1) = 0.0;          GQ.point(2,2) = -sqrt(3/5);
    GQ.point(3,1) = sqrt(3/5);    GQ.point(3,2) = -sqrt(3/5);
    GQ.point(4,1) = -sqrt(3/5);   GQ.point(4,2) = 0.0;
    GQ.point(5,1) = 0.0;          GQ.point(5,2) = 0.0;
    GQ.point(6,1) = sqrt(3/5);    GQ.point(6,2) = 0.0;
    GQ.point(7,1) = -sqrt(3/5);   GQ.point(7,2) = sqrt(3/5);
    GQ.point(8,1) = 0.0;          GQ.point(8,2) = sqrt(3/5);
    GQ.point(9,1) = sqrt(3/5);    GQ.point(9,2) = sqrt(3/5);
    GQ.weight(1) = 5/9 * 5/9;
    GQ.weight(2) = 8/9 * 5/9;
    GQ.weight(3) = 5/9 * 5/9;
    GQ.weight(4) = 5/9 * 8/9;
    GQ.weight(5) = 8/9 * 8/9;
    GQ.weight(6) = 5/9 * 8/9;
    GQ.weight(7) = 5/9 * 5/9;
    GQ.weight(8) = 8/9 * 5/9;
    GQ.weight(9) = 5/9 * 5/9;
  end
```

Gauss Quadrature points and weights are coming from Tables 3.1 and 3.2.

```
elseif eType == 2   % Triangular element
  if NGP == 1       % One-point quadrature
    GQ.point(1,1) = 1/3;     GQ.point(1,2) = 1/3;
    GQ.weight(1) = 0.5;
  elseif NGP == 3   % Two-point quadrature
    GQ.point(1,1) = 0.5;     GQ.point(1,2) = 0.0;
    GQ.point(2,1) = 0.0;     GQ.point(2,2) = 0.5;
    GQ.point(3,1) = 0.5;     GQ.point(3,2) = 0.5;
    GQ.weight(1) = 1/6;
    GQ.weight(2) = 1/6;
    GQ.weight(3) = 1/6;
  elseif NGP == 4   % Four-point quadrature
    GQ.point(1,1) = 1/3;     GQ.point(1,2) = 1/3;
    GQ.point(2,1) = 0.6;     GQ.point(2,2) = 0.2;
    GQ.point(3,1) = 0.2;     GQ.point(3,2) = 0.6;
    GQ.point(4,1) = 0.2;     GQ.point(4,2) = 0.2;
    GQ.weight(1) = -27/96;
    GQ.weight(2) = 25/96;
    GQ.weight(3) = 25/96;
    GQ.weight(4) = 25/96;
  elseif NGP == 7   % Seven-point quadrature
    GQ.point(1,1) = 1/3;                 GQ.point(1,2) = 1/3;
    GQ.point(2,1) = 0.059715871789770;   GQ.point(2,2) = 0.470142064105115;
    GQ.point(3,1) = 0.470142064105115;   GQ.point(3,2) = 0.059715871789770;
    GQ.point(4,1) = 0.470142064105115;   GQ.point(4,2) = 0.470142064105115;
    GQ.point(5,1) = 0.101286507323456;   GQ.point(5,2) = 0.797426985353087;
    GQ.point(6,1) = 0.101286507323456;   GQ.point(6,2) = 0.101286507323456;
    GQ.point(7,1) = 0.797426985353087;   GQ.point(7,2) = 0.101286507323456;
    GQ.weight(1) = 0.225 / 2;
    GQ.weight(2) = 0.132394152788 / 2;
    GQ.weight(3) = 0.132394152788 / 2;
    GQ.weight(4) = 0.132394152788 / 2;
    GQ.weight(5) = 0.125939180544 / 2;
    GQ.weight(6) = 0.125939180544 / 2;
    GQ.weight(7) = 0.125939180544 / 2;
  end
end

% End of function setupGQ()
```

```
%=========================================================================
function [] = calcShape()
%=========================================================================
% Calculates the values of shape functions and their derivatives with
% respect to ksi and eta at GQ points.
global eType NEN NGP GQ S dS;

if eType == 1  % Quadrilateral element
  if NEN == 4  % Linear Lagrange shape functions
    for k = 1:NGP
      ksi = GQ.point(k,1);
      eta = GQ.point(k,2);

      S(1,k) = 0.25*(1-ksi)*(1-eta);
      S(2,k) = 0.25*(1+ksi)*(1-eta);
      S(3,k) = 0.25*(1+ksi)*(1+eta);
      S(4,k) = 0.25*(1-ksi)*(1+eta);

      % ksi derivatives of S
      dS(1,1,k) = -0.25*(1-eta);
      dS(1,2,k) =  0.25*(1-eta);
      dS(1,3,k) =  0.25*(1+eta);
      dS(1,4,k) = -0.25*(1+eta);

      % eta derivatives of S
      dS(2,1,k) = -0.25*(1-ksi);
      dS(2,2,k) = -0.25*(1+ksi);
      dS(2,3,k) =  0.25*(1+ksi);
      dS(2,4,k) =  0.25*(1-ksi);
    end
  else
    disp('ERROR: For quadratic elements only 4-node is supported.');
  end
elseif eType == 2  % Triangular element
  if NEN == 3   % Linear Lagrange shape functions
    for k = 1:NGP
      ksi = GQ.point(k,1);
      eta = GQ.point(k,2);

      S(1,k) = 1 - ksi - eta;
      S(2,k) = ksi;
      S(3,k) = eta;

      % ksi derivatives of S
      dS(1,1,k) = -1;
      dS(1,2,k) =  1;
      dS(1,3,k) =  0;

      % eta derivatives of S
      dS(2,1,k) = -1;
      dS(2,2,k) =  0;
      dS(2,3,k) =  1;
    end
  else
    disp('ERROR: For triangular elements only 3-node is supported.');
  end
end

% End of function calcShape()
```

Shape functions for bilinear quadrilaterals and triangles are coming from Figures 3.1 and 3.2.

```
%=========================================================================
function [] = calcJacob()
%=========================================================================
% Calculates Jacobian matrix and its determinant for each element. Also
% calculates and stores the derivatives of shape functions wrt x and y
% coordinates at GQ points for each element.
global NE NEN NGP coord dS elem;

for e = 1:NE
  % To calculate the Jacobian matrix first generate e_coord matrix of size
  % NENx2. Each row of it stores x and y coords of the nodes of elem e.
  for i = 1:NEN
    iG = elem(e).LtoG(i);
    e_coord(i,:) = coord(iG,:);  % Copy both x and y cooordinates at once.
  end

  % For each GQ point calculate the 2x2 Jacobian matrix, its inverse and
  % determinant. Only store the determinant for each element. Also
  % calculate and store the shape function derivatives wrt x and y.
  for k = 1:NGP
    Jacob(:,:) = dS(:,:,k) * e_coord(:,:);
    elem(e).gDS(:,:,k) = inv(Jacob(:,:)) * dS(:,:,k);

    elem(e).detJacob(k) = det(Jacob);
  end
end

% End of function calcJacob()
```

Equation 3.15
Equation 3.16

```
%=========================================================================
function calcGlobalSys()
%=========================================================================
% Calculates the global stiffness matrix [K] and force vector {F}.
global NN NE soln;

soln.K = zeros(NN,NN);
soln.F = zeros(NN,1);    % {F} is a column vector, not a row vector.

for e = 1:NE
  calcElemSys(e);
  assemble(e);
end

% End of function calcGlobalSys()
```

```
%=========================================================================
function [] = calcElemSys(e)
%=========================================================================
% Calculates the element level stiffness matrix and force vector.
global NEN NGP coord S GQ elem funcs;

elem(e).Fe = zeros(NEN,1);
elem(e).Ke = zeros(NEN,NEN);

for k = 1:NGP   % Gauss Quadrature loop
  % Using isoparametric formulation idea, calculate global x and y
  % coordinates corresponding to ksi and eta.
  x = 0;
  y = 0;
  for i = 1:NEN
    iG = elem(e).LtoG(i);
    x = x + S(i,k) * coord(iG,1);
    y = y + S(i,k) * coord(iG,2);
  end

  aValue = eval(funcs.a);
  V1Value = eval(funcs.V1);
  V2Value = eval(funcs.V2);
  cValue = eval(funcs.c);
  fValue = eval(funcs.f);

  for i = 1:NEN
    for j = 1:NEN
      elem(e).Ke(i,j) = elem(e).Ke(i,j) + ...
                 (aValue * (elem(e).gDS(1,i,k) * elem(e).gDS(1,j,k) + ...
                            elem(e).gDS(2,i,k) * elem(e).gDS(2,j,k)) + ...
                  S(i,k) * (V1Value *  elem(e).gDS(1,j,k) + ...
                            V2Value *  elem(e).gDS(2,j,k)) + ...
                  cValue * S(i,k) * S(j,k) ...
                 ) * elem(e).detJacob(k) * GQ.weight(k);
    end
  end

  for i = 1:NEN
    elem(e).Fe(i) = elem(e).Fe(i) + ...
                    S(i,k) * fValue * elem(e).detJacob(k) * GQ.weight(k);
  end

end  % End of GQ loop

% End of function calcElemSys()
```

Equation 3.12

$$x = \sum_{j=1}^{NEN} x_j^e \, S_j \quad , \quad y = \sum_{j=1}^{NEN} y_j^e \, S_j$$

$$K_{ij}^e = \int_{-1}^{1} \left[a \left(\frac{dS_i}{dx}\frac{dS_j}{dx} + \frac{dS_i}{dy}\frac{dS_j}{dy} \right) + S_i \left(V_1 \frac{dS_j}{dx} + V_2 \frac{dS_j}{dy} \right) + c S_i S_j \right] |J^e| \, d\xi$$

$$F_i^e = \int_{-1}^{1} f S_i \, J^e \, d\xi$$

```
%=========================================================================
function [] = assemble(e)
%=========================================================================
% Inserts [Ke] and {Fe} into proper locations of [K] and {F} by the help
% of LtoG arrays of elements.

global NEN elem soln;

% Assemble Ke into K.
for i = 1:NEN
  I = elem(e).LtoG(i);    % I is the global node corresponding to the i-th
                          % local node of element e
  for j = 1:NEN
    J = elem(e).LtoG(j);    % J is similar to I.
    soln.K(I,J) = soln.K(I,J) + elem(e).Ke(i,j);
  end
end

% Assemble Fe into F.
for i = 1:NEN
  I = elem(e).LtoG(i);
  soln.F(I) = soln.F(I) + elem(e).Fe(i);
end

% End of function assemble()
```

```
%=========================================================================
function [] = applyBC()
%=========================================================================
% For EBCs reduction is NOT applied. Instead, global [K] and {F} are
% modified. For NBCs only SVs specified as constants are supported.
% Provided SVs are added to {F}, there is no separate {B} vector. For
% MBCs only constant alpha and beta values are supported.

global soln coord BC;


%=========================================
% Modify {F} for NBCs.
%=========================================
for i = 1:BC.nNBC         % Loop over the faces with specified NBC
  e = BC.NBCfaces(i,1);   % This is the element where NBC is specified
  f = BC.NBCfaces(i,2);   % This is the face of element e
  whichBC = BC.NBCfaces(i,3);  % This is the no. of BC specified

  SVvalue = eval(char(BC.str(whichBC,1)));  % Specified SV value

  [nodes length] = getFaceDetails(e,f);  % Calculate the length of the
                                         % face where NBC is specified
                                         % and determine global node
                                         % numbers of the face.

  % Apply the formula derived for NBCs where constant SV is applied to a
  % 2-node face.
  soln.F(nodes(1)) = soln.F(nodes(1)) + 0.5 * SVvalue * length;
  soln.F(nodes(2)) = soln.F(nodes(2)) + 0.5 * SVvalue * length;
end
```

Equation (3.30)

```
%=========================================
% Modify [K] and {F} for mixed type BCs.
%=========================================
for i = 1:BC.nMBC        % Loop over the faces with specified mixed BC
  e = BC.MBCfaces(i,1);  % This is the element where mixed BC is specified
  f = BC.MBCfaces(i,2);  % This is the face of element e.
  whichBC = BC.MBCfaces(i,3);  % This is the no. of BC specified.

  alpha = eval(char(BC.str(whichBC,1)));  % alpha value of mixed BC.
  beta = eval(char(BC.str(whichBC,2)));   % beta value of mixed BC.

  [nodes length] = getFaceDetails(e,f);

  % For a mixed BC applied on a 2-node face with constant alpha and beta
  % values, we derived the following two relations for the elemental
  % boundary integral vector entries corresponding to the two nodes of
  % the face.
  %
  % B1 = (3*beta + 2*alpha*T1 + alpha*T2) * Lface / 6
  % B2 = (3*beta + alpha*T1 + 2*alpha*T2) * Lface / 6

  % Start with changes due to B1.

  % Apply (beta * Lface / 2)  part of B1
  soln.F(nodes(1)) = soln.F(nodes(1)) + 0.5 * beta * length;

  % Apply (alpha * Lface / 3 * T1)  part of B1. Move it to [K].
  soln.K(nodes(1),nodes(1)) = soln.K(nodes(1),nodes(1)) - ...
                              alpha * length / 3;

  % Apply (alpha * Lface / 6 * T2)  part of B1. Move it to [K].
  soln.K(nodes(2),nodes(2)) = soln.K(nodes(2),nodes(2)) - ...
                              alpha * length / 6;


  % Now apply changes due to B2.

  % Apply (beta * Lface / 2)  part of B2
  soln.F(nodes(2)) = soln.F(nodes(2)) + 0.5 * beta * length;

  % Apply (alpha * Lface / 3 * T2)  part of B2
  soln.K(nodes(2),nodes(2)) = soln.K(nodes(2),nodes(2)) - ...
                              alpha * length / 3;

  % Apply (alpha * Lface / 6 * T1)  part of B2
  soln.K(nodes(1),nodes(1)) = soln.K(nodes(1),nodes(1)) - ...
                              alpha * length / 6;
end  % End of mixed BC face loop


%=========================================
% Modify [K] and {F} for EBCs.
%=========================================
for i = 1:BC.nEBC
  node = BC.EBCnodes(i,1);     % Node at which this EBC is specified
  whichBC = BC.EBCnodes(i,2);  % Number of the specified BC

  x = coord(node,1);           % May be necessary for BCstring evaluation
  y = coord(node,2);

  soln.F(node) = eval(char(BC.str(whichBC,1)));  % Specified PV value
  soln.K(node,:) = 0.0;
  soln.K(node,node) = 1.0;
end

% End of function applyBC()
```

Equation (3.34)

Equation (3.35)

Reduction is NOT applied for EBCs.

```
%=========================================================================
function [nodes length] = getFaceDetails(e, f)
%=========================================================================
% Finds the nodes located at face f of element e. Works only for faces
% with 2 nodes. For higher order elements this part should change. Length
% of the face is also calculated.

global eType elem coord

if eType == 1
  nFace = 4;  % Quadrilateral elements have 4 faces.
else
  nFace = 3;  % Triangular elements have 3 faces.
end

% Determine the nodes on face f of element e.
if f ~= nFace  % Last face is treated separately than other faces.
  nodes(1) = elem(e).LtoG(f);
  nodes(2) = elem(e).LtoG(f+1);
else
  nodes(1) = elem(e).LtoG(nFace);
  nodes(2) = elem(e).LtoG(1);
end

% Calculate the length of face f of element e.
dx = coord(nodes(1),1) - coord(nodes(2),1);
dy = coord(nodes(1),2) - coord(nodes(2),2);
length = sqrt(dx*dx + dy*dy);

% End of function getFaceDetails()
```

```
%=========================================================================
function [] = solve()
%=========================================================================
% Solves the system [K]{U}={F}. Note that this is generally the most time
% consuming part of the solution. Backslash operator that we use is an
% inefficient operator and there are alternative techniques.
global soln;

soln.U = soln.K \ soln.F;

% soln.U = sparse(soln.K) \ soln.F;

% End of function solve()
```

Converting $[K]$ into a sparse matrix results in a faster solution.

```
%=========================================================================
function [] = postProcess()
%=========================================================================
% Generates contour plot of the scalar unknown.
global NN NE NEN coord elem soln

% Write the calculated unknowns to the screen.
fprintf(1,'\nCalculated unknowns are \n\n');
fprintf(1,' Node              x                   y                    u \n');
fprintf(1,'=================================================================\n');
for i = 1:NN
  fprintf(1, ' %-5d %18.8f %18.8f %20.8f\n', i, coord(i,1), coord(i,2), soln.U(i));
end



% Generate necessary data for MATLAB's patch function that'll be used to
% generate the contour plot.
for e = 1:NE
  for i = 1:NEN
    x(i,e) = coord(elem(e).LtoG(i),1);
    y(i,e) = coord(elem(e).LtoG(i),2);
    z(i,e) = soln.U(elem(e).LtoG(i));
  end
end


patch(x,y,z);
axis equal;
colorbar;    % Put a colorbar next to the graph
xlabel('x');
ylabel('y');
zlabel('T');
title('Contour plot of the FEM solution');

calculateSV();

% End of function postProcess()




%=========================================================================
function [] = calculateSV()
%=========================================================================
% Calculates SV's at EBC and MBC boundaries.

% This part is missing
% ...
% ...
% ...

% End of function calculateSV()
```

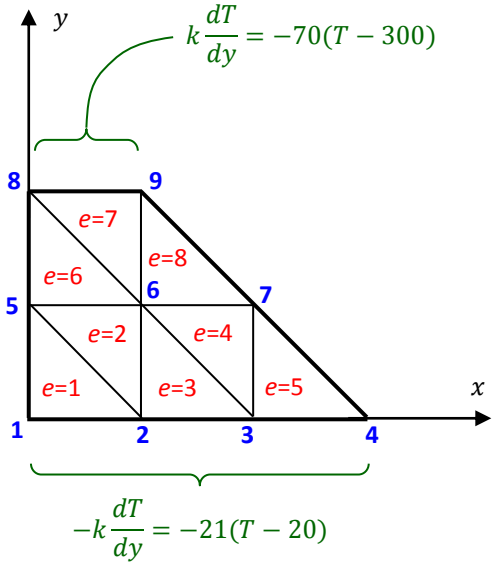
```
%=========================================================================
function [] = createTecplotOutput()
%=========================================================================
% Generates an output file that can be used to visualize the results using
% the commercial Tecplot software.

global NE NEN NN eType coord elem soln prName;

% Write the calculated unknowns to a Tecplot file
outputFile = fopen(strcat(prName, '_tecplot.dat'), 'w');

fprintf(outputFile, '%s %s \n', 'TITLE =', prName);
fprintf(outputFile, '%s \n', 'VARIABLES = x, y, T');
if (eType == 1)
   fprintf(outputFile, '%s %i %s %i %s \n', 'ZONE N=', NN, ...
                       ', E=', NE, ', F=FEPOINT , ET=QUADRILATERAL');
else
   fprintf(outputFile, '%s %i %s %i %s \n', 'ZONE N=', NN, ...
                       ', E=', NE, ', F=FEPOINT , ET=TRIANGLE');
end

% Print the coordinates and the calculated values
for i = 1:NN
   x = coord(i,1);
   y = coord(i,2);
   fprintf(outputFile, '%10.6f %10.6f %10.6f \n', x, y, soln.U(i));
end

% Print the connectivity list
for e = 1:NE
   for i = 1:NEN
      fprintf(outputFile, '%5i', elem(e).LtoG(i));
   end
   fprintf(outputFile, '\n');
end

fclose(outputFile);

% End of function createTecplotOutput()
```

4.3 Sample 2D Input File (ChimneyNE8.inp)

For 2D problems problem data is read from an input file, which has a very strict format. You can NOT make arbitrary modifications to it without changing the MATLAB code that reads it. The following input file is for the chimney problem that we solved in Section 3.7, but this one uses 8 triangular elements.

```
Sample 2D input file for steady2D.m (Spring 2012)
================================================
eType     : 2
NE        : 8
NN        : 9
NEN       : 3
NGP       : 4
aFunc     : 1.4
V1Func    : 0.0
V2Func    : 0.0
cFunc     : 0.0
fFunc     : 0.0
================================================
Node#          x          y
    1        0.0        0.0
    2        0.05       0.0
    3        0.1        0.0
    4        0.15       0.0
    5        0.0        0.05
    6        0.05       0.05
    7        0.1        0.05
    8        0.0        0.1
    9        0.05       0.1
================================================
Elem#     node1    node2    node3    node4
    1         1        2        5
    2         2        6        5
    3         2        3        6
    4         3        7        6
    5         3        4        7
    6         5        6        8
    7         6        9        8
    8         6        7        9
================================================
BCs (Number of specified BCs, their types and strings)
nBC         : 2
BC 1        : 3   -21 : 420
BC 2        : 3   -70 : 21000
================================================
nEBCnodes : 0
nNBCfaces : 0
nMBCfaces : 4
================================================
EBC Data (Node#   BC#)
NBC Data (Elem# Face# BC#)
MBC Data (Elem# Face# BC#)
  1     1     1
  3     1     1
  5     1     1
  7     2     2
```

Conductivity of the concrete



x and y coordinates of 9 nodes.

LtoG

α and β values are given for the BCs.

First BC is for the outside convection.
Second BC is for the outside convection.

There are 4 faces where mixed BC is specified.

1st elements's 1st face, 3rd element's 1st face and 5th element's 1st face has the 1st BC.

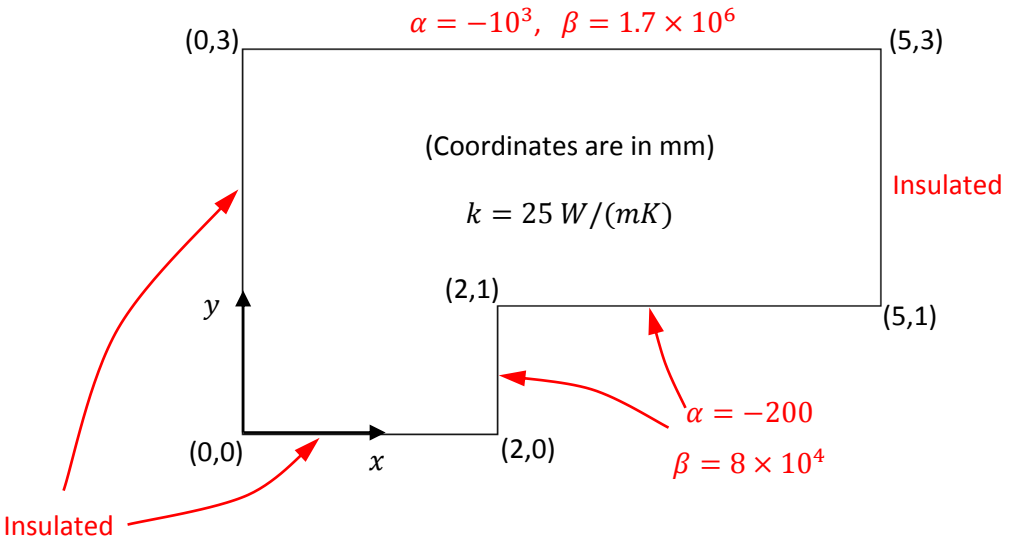
7th element's 2nd face has the 2nd BC.

4.4 How to Generate 2D Meshes with generate2DinputFile.m

mesh2d is a mesh generator written in MATLAB. It can freely be downloaded at http://www.mathworks.com/matlabcentral/fileexchange/25555-mesh2d-automatic-mesh-generation . It can be used to generate meshes of triangular elements over 2D domains. To see sample meshes created by it you can run `meshdemo` and `mesh_collection(#)` files that come with it.

The main difficulty in using a third party mesh generator like mesh2d is converting its output into the format of the input files that is understood by our FEM solvers. For this purpose we'll use a MATLAB code called **generate2DinputFile.m**. It is available at the course web site.

This document explains how to use generate2DinputFile.m code by the help of examples. Let's start with the following heat conduction problem, which is Exercise E-3.4 of Chapter 3. Coordinates shown below are in mm.



generate2DinputFile.m code is divided into 5 parts. Beginning of each part is labeled with comments to be detected easily. Problem dependent parameters that need to be modified for each problem are also mentioned at the beginning of each part.

In Part 1 shown below, we first select number of Gauss Quadrature Points, `NGP`. Also functions of the differential equation are specified as strings. For solving a conduction problem with a conductivity of $25\ W/(mK)$, `aFunc` parameter is equated to this value. All other functions are zero.

We then provide the number of BCs. For this problem there are 3 different boundary conditions and their types are provided in the `BCtype` array. Boundary condition types supported are

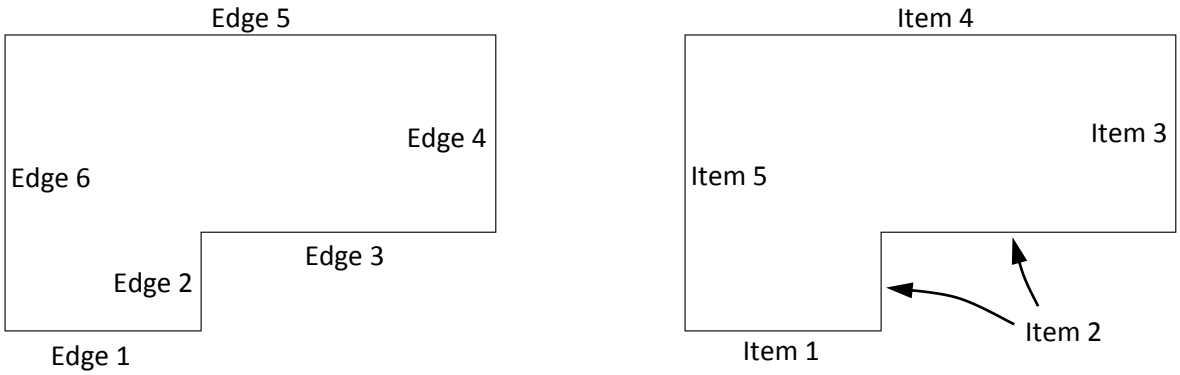
1: Essential BC (EBC) 2: Natural BC (NBC) 3: Mixed BC (MBC)

First BC is selected to be the insulated boundary, which is of type 2. The other two BCs are of mixed type, i.e. of type 3. We do not have any EBCs for this problem.

In Part 1 we also provide the specified values/functions for these three BCs in the `BCstr` cell array. Cell arrays are defined using braces instead of brackets. `BCstr` has two strings for each BC. For EBCs and NBCs only the first string is used to specify either the primary variable or the secondary variable. Here the first BC is of type NBC and we need to specify the known secondary variable, which is zero for an insulated boundary. Therefore the first string of the first BC is `'0.0'`. Second and third BCs are of mixed type and both strings are used to specify α and β values. The numbering of BCs is a user choice. It is possible to change their order in the way we want.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                             %
% PART 1. CONSTANTS and BOUNDARY CONDITION DATA                               %
%                                                                             %
% Problem dependent parameters are NGP, functions of the DE, nBC, BCtype      %
% and BCstr.                                                                  %
%                                                                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global LtoG coord itemCoord

eType = 2;          % mesh2d generates only triangular elements.
NEN = 3;            % Number of an element's node. Bi-linear triangles
                    % have 3 nodes.
NGP = 4;            % Number of GQ points that'll appear in the input file.

aFunc = '25.0';     % Functions of the Differential Equation.
V1Func = '0.0';
V2Func = '0.0';
cFunc = '0.0';
fFunc = '0.0';

nBC = 3;            % Number of specified BCs.

% Specify BC type in an array of length nBC. 1: EBC, 2: NBC, 3: MBC
BCtype = [2   3   3];

% Specify PVs, SVs or alpha and beta values for BCs as strings in a cell
% array of size nBCx2.
BCstr = {'0.0'      '0.0';
         '-200'     '8e4';
         '-1000'    '1.7e6'};
```

In the second part of generate2DinputFile.m, details about **edges** (line segments) that define the problem domain are given. The problem we are working on, shown below, is drawn using 6 line segments, i.e. it has 6 edges. We also use the concept of **item** which is different than an edge. An item can either be a single edge or multiple, connected edges. Later we'll assign BCs to these items, such that on each item only a single BC type can be specified. For the problem of interest we'll use the 5 items shown below on the right.



It is possible to treat each edge as a separate item, but for problem domains with 10s of edges grouping connected edges with the same BC into a single item simplifies the code and the BC specification part.

In Part 2 of generate2DinputFile.m shown below, we first specify the number of items (`nItem`) and the number of edges that form each item (`nItemEdges`). For the problem we are working on there are 5 items and only the second item has two edges, all the other items have only one edge.

Next the user provides the coordinates of the end points of each edge of each item using `itemCoord` variable. If an item has a single edge we need to specify the coordinates of two end points of the edge. For an item with N edges we need to specify the coordinates of N+1 points that define these edges.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
% PART 2. ITEM DETAILS                                                          %
%                                                                               %
% Problem dependent parameters are nItem, nItemEdges and itemCoord.             %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nItem = 5;
nItemEdges = [1 2 1 1 1];

nEdge = sum(nItemEdges);
nItemEdgesMax = max(nItemEdges);
itemCoord = zeros(nItem, nItemEdgesMax+1, 2);

% Provide coordinates of the nodes of each item
item = 1;
itemCoord(item,1,1) = 0.0;     itemCoord(item,1,2) = 0.0;
itemCoord(item,2,1) = 0.002;   itemCoord(item,2,2) = 0.0;

item = 2;
itemCoord(item,1,1) = 0.002;   itemCoord(item,1,2) = 0.0;
itemCoord(item,2,1) = 0.002;   itemCoord(item,2,2) = 0.001;
itemCoord(item,3,1) = 0.005;   itemCoord(item,3,2) = 0.001;

item = 3;
itemCoord(item,1,1) = 0.005;   itemCoord(item,1,2) = 0.001;
itemCoord(item,2,1) = 0.005;   itemCoord(item,2,2) = 0.003;

item = 4;
itemCoord(item,1,1) = 0.005;   itemCoord(item,1,2) = 0.003;
itemCoord(item,2,1) = 0.0;     itemCoord(item,2,2) = 0.003;

item = 5;
itemCoord(item,1,1) = 0.0;     itemCoord(item,1,2) = 0.003;
itemCoord(item,2,1) = 0.0;     itemCoord(item,2,2) = 0.0;
```

In Part 3 of generate2DinputFile.m shown below, we specify variables of `hdata` structure to control size of the elements. `hdata.hmax` is used to provide a single maximum element size for the whole mesh, so that all the elements will be smaller than the specified size. `hdata.edgeh` variable is used to specify the size of elements that are in contact with an edge. Finally `hdata.fun` variable is used to control the distribution of element size using functions of space coordinates x and y. For the code segment shown below the parts related to `hdata` variable are commented out, meaning that we are not controlling the size of elements in any way and accept the default behavior.

Towards the end of Part 3, mesh2d function is called to generate a mesh of triangular elements. Output of mesh2d is two matrices named `coord` and `LtoG`. The first one stores the coordinates of generated mesh nodes and the second one stores the connectivity information of the elements. mesh2d function plots the created mesh so that the user can see it before generating the input file. The user does not need to change anything in calling the `mesh2d` function.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                              %
% PART 3. MESH GENERATION                                                      %
%                                                                              %
% The only problem dependent variable is hdata, which is used to control       %
% size of the elements.                                                        %
%                                                                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% hdata structure is used by mesh2d to control element sizes. It has three
% main variables as hmax, edgeh and fun.
hdata = [];

% hdata.max and hdata.edgeh are are used to control element size for the
% whole mesh and for certain edges.

% hdata.hmax = 0.00025;
% hdata.edgeh(1,1) = 5;   hdata.edgeh(1,2) = 0.0002;


% hdata.fun can be used to specify functions for controling mesh size
% at different regions of the domain. Following is one line function, but
% you can also write separate, more complicated functions.

% hdata.fun = @(x,y) 0.000025 + 0.1 * sqrt((x-0.002).^2 + (y-0.001).^2);

% ......
% ......  Omitted lines...
% ......

% mesh2d function creates the mesh. First output is a matrix of NNx2 that
% holds x and y coordinates of mesh nodes. Second output is a NEx3 matrix
% that stores corner node numbers of NE elements.

[coord, LtoG] = mesh2d(nodeCoord, edgeNodes, hdata, []);
```

In Part 4 of generate2DinputFile.m shown below, we only provide the `itemBC` variable, which is used to store the BC of each item. For the sample problem of interest 1st BC is used for the 1st, 3rd and 5th items. 2nd BC is used for the 2nd item and 3rd BC is used for the 4th item.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
% PART 4. BC DETERMINATION                                                      %
%                                                                               %
% The only problem dependent parameter is itemBC.                               %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fprintf ('\nFinding BCs. This may take some time for large meshes.\n');

% Specify the BC number for each geometrical item. Length of itemBC array
% is nItem.
itemBC(1) = 1;
itemBC(2) = 2;
itemBC(3) = 1;
itemBC(4) = 3;
itemBC(5) = 1;
```

The last part, Part 5, of the code is used to generate the input file that can be used with our 2D FEM solvers. There is nothing in this part that we need to change. In this part we are asked to enter a file name at the command line and a file with the specified name with an INP extension is created. If the user does not want to generate an input file for the created mesh he/she can press Ctrl-C to abort the running code.

Now we'll present 5 different meshes created for the sample problem that is being discussed by specifying different size control options through the use of the variables of `hdata` structure.

First mesh given below uses no mesh size control at all.

For the second mesh `hdata.hmax` is specified as 0.00025. Therefore smaller elements are created everywhere and the mesh is uniform, i.e. all elements have similar size.

Third mesh uses `hdata.edgeh` variable only to specify the size of elements on the 5th edge to be 0.0002 using the following code.

```
hdata.edgeh(1,1) = 5;
hdata.edgeh(1,2) = 0.0002;
```

`edgeh` is a matrix with 2 columns. Its first column is used to specify the edge numbers on which we want to control the element size. Above we entered it to be 5. Second column of `edgeh` variable is used to store the desired element size. We entered it to be a small value of 0.0002.

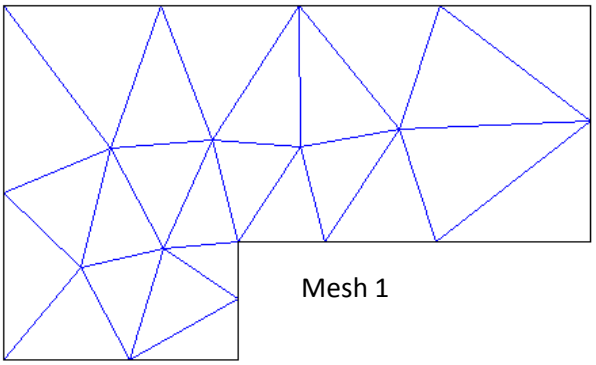
For the fourth mesh again `hdata.edgeh` is used but this time to specify element size at two edges, the 4th and the 6th edge. Code segment to do this is shown below

```
hdata.edgeh(1,1) = 4;
hdata.edgeh(1,2) = 0.0002;
hdata.edgeh(2,1) = 6;
hdata.edgeh(2,2) = 0.0002;
```

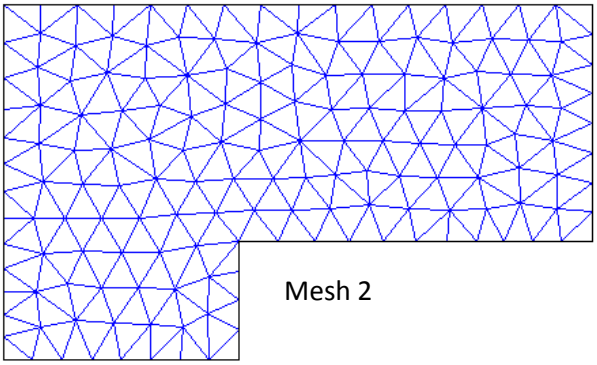
Finally for the fifth mesh `hdata.fun` variable is used to specify element size as follows

```
hdata.fun = @(x,y) 0.000025 + 0.1 * sqrt((x-0.002).^2 + (y-0.001).^2);
```

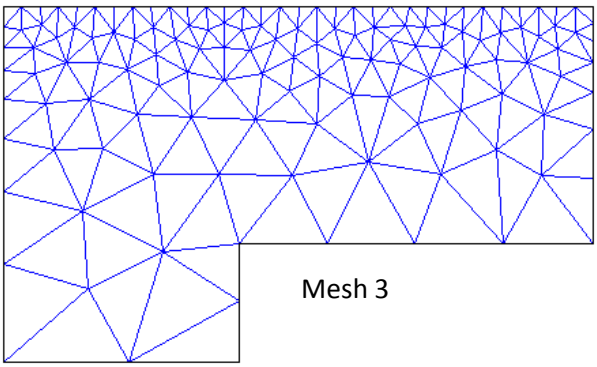
Here we provide a size distribution function. "`@(x,y)`" part says that the function we provide is a function of x and y. The rest of the line is the function itself. Its value is 0.000025 at point (0.002, 0.001), which is the smallest value that the function can take. As we radially go out from this point the function gets larger. Therefore this function is used to generate fine mesh near point (0.002, 0.001) and the mesh gradually gets coarser as we move away from this point. Much more complicated functions can be written to control the size of the elements in various different ways.



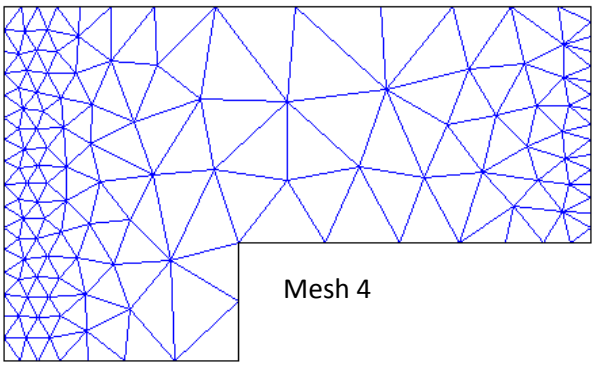
Mesh 1



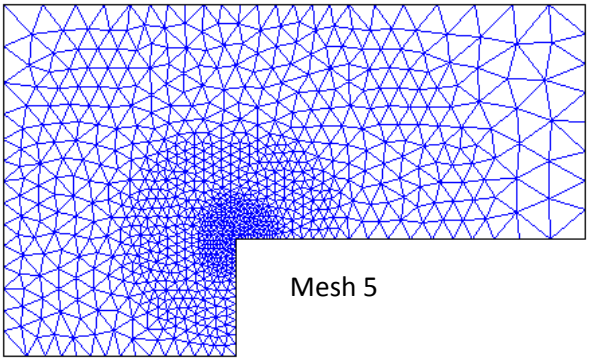
Mesh 2



Mesh 3

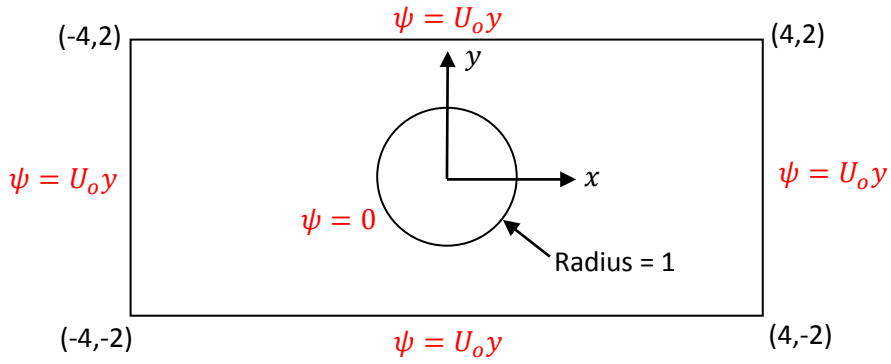


Mesh 4



Mesh 5

As a second problem let's consider the potential flow over a cylinder shown below.



There are two different BCs, one for the rectangular box and one for the circle. The same BC is specified on all 4 edges of the rectangular box, therefore we can consider the box as a single item. Similarly the circle is another item. In total there are 2 items.

The circular hole needs to be represented by a number of straight edges (line segments). Creation of these edges can be simplified by the use of a for loop as seen in Part 2 of the code given below. In the provided code the circle is drawn as a combination of 16 edges.

Here it is important to note that both the first and the second item are closed line segments, i.e. their first and second points should exactly match. In other words the coordinates of the first and last points of `itemCoord` variable should be the same.

In the mesh generation part let's not use any `hdata` variable for mesh size control.

In Part 4 we only specify `itemBC` variable.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 1. CONSTANTS and BOUNDARY CONDITION DATA                                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

global LtoG coord itemCoord

eType = 2;
NEN = 3;
NGP = 4;

aFunc  = '1.0';
V1Func = '0.0';
V2Func = '0.0';
cFunc  = '0.0';
fFunc  = '0.0';

nBC = 2;
BCtype = [1  1];
BCstr = {'y'      '0.0';    % Freestream velocity Uo is taken to be 1
         '0.0'    '0.0'};
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 2. ITEM DETAILS                                                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nItem = 2;
nCircleEdge = 16;              % We'll divide the circle into 16 edges
nItemEdges = [4 nCircleEdge];

nEdge = sum(nItemEdges);
nItemEdgesMax = max(nItemEdges);
itemCoord = zeros(nItem, nItemEdgesMax+1, 2);

% Provide coordinates of the nodes of each item
item = 1;  % 1st item is the rectangular box
itemCoord(item,1,1) = -4.0;    itemCoord(item,1,2) = -2.0;
itemCoord(item,2,1) = 4.0;     itemCoord(item,2,2) = -2.0;
itemCoord(item,3,1) = 4.0;     itemCoord(item,3,2) = 2.0;
itemCoord(item,4,1) = -4.0;    itemCoord(item,4,2) = 2.0;
itemCoord(item,5,1) = -4.0;    itemCoord(item,5,2) = -2.0;

item = 2;   % 2nd item is the circular hole
for i = 1:nCircleEdge
  angle = pi/(nCircleEdge/2) * (i-1);
  itemCoord(item,i,1) = cos(angle);
  itemCoord(item,i,2) = sin(angle);
end

% Make sure that the last coordinate that defines the circle is the same
% as the first coordinate. If we do this calculation inside the above for
% loop the two points may not match due to round-off errors.
itemCoord(item,nCircleEdge+1,1) = itemCoord(item,1,1);
itemCoord(item,nCircleEdge+1,2) = itemCoord(item,1,2);
```

Coordinates of the rectangular box

Coordinates of the circular hole

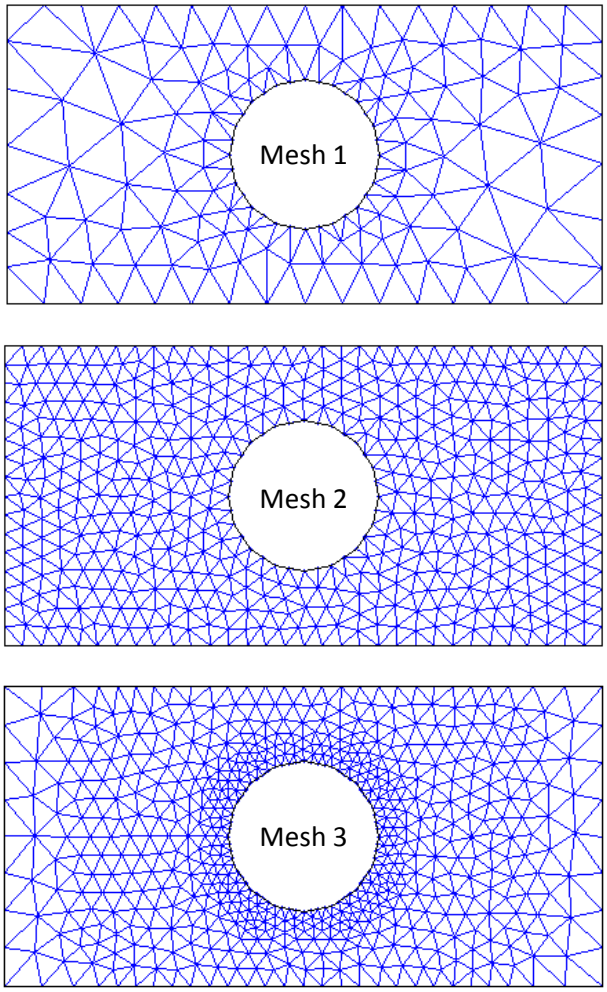
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 4. BC DETERMINATION                                                     %
% The only problem dependent parameter is itemBC.                              %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

itemBC(1) = 1;
itemBC(2) = 2;
```

The generated mesh without any element size control is shown below as the first mesh.

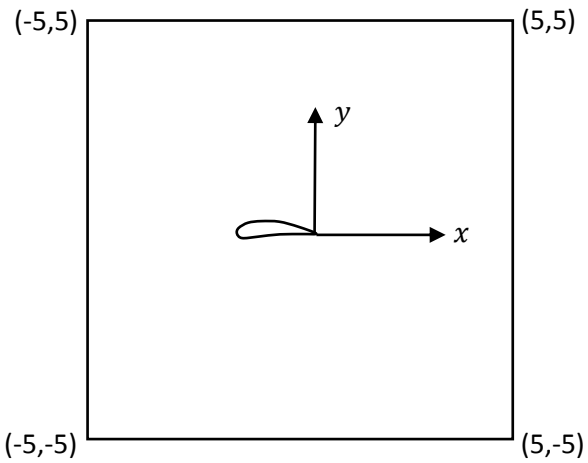
Second mesh is created by using a size control of `hdata.hmax = 0.2`.

Third one is obtained with the following size function (Note the use of `.*` operator instead of `*`)

```
hdata.fun = @(x,y) 0.05 + 0.1 * sqrt(x.*x + y.*y);
```



Finally let's discuss how to generate a mesh for the solution of potential flow around an airfoil to demonstrate how coordinates of an item can be read from an input file. Problem domain is shown below. Outside box and the airfoil are selected to be the first and second items, respectively.



Second part of generate2DinputFile.m is shown below. Coordinates of the outside box are entered directly into the code. However, coordinates of the airfoil are read from an input file named NACA4412.txt. Here we defined

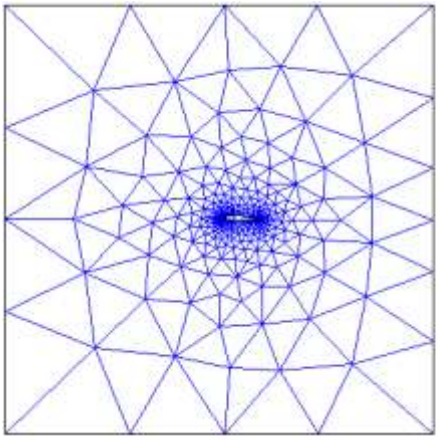
the airfoil in terms of 34 edges connecting 35 nodes and this input file contains x and y coordinates of these 35 nodes, first and last point being the same to form a closed airfoil shape. Each line in the file has the coordinates of one point.

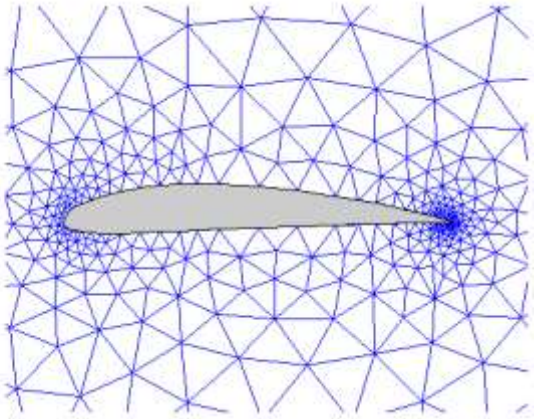
The generated mesh is shown below. As seen from the zoomed in view, very small elements are created at the trailing edge of the airfoil, although no constraint is provided for the element size. The reason of this unnecessary mesh refinement is the sharp and pointed geometry of the trailing edge and the way these type of geometries are handled by the mesh generation technique used in mesh2d. To avoid such unnecessarily fine elements the geometry of the airfoil may be altered slightly and the sharpness of the trailing edge may be reduced.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% PART 2. ITEM DETAILS                                                          %
% Problem dependent parameters are nItem, nItemEdges and itemCoord.             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

nItem = 2;
nAirfoilEdge = 34;   % Airfoil is defined using 34 edges.
nItemEdges = [4 nAirfoilEdge];

nEdge = sum(nItemEdges);
nItemEdgesMax = max(nItemEdges);
itemCoord = zeros(nItem, nItemEdgesMax+1, 2);


% Provide coordinates of the nodes of each item
item = 1;  % 1st item is the outside box
itemCoord(item,1,1) = -5.0;    itemCoord(item,1,2) = -5.0;
itemCoord(item,2,1) = 5.0;     itemCoord(item,2,2) = -5.0;
itemCoord(item,3,1) = 5.0;     itemCoord(item,3,2) = 5.0;
itemCoord(item,4,1) = -5.0;    itemCoord(item,4,2) = 5.0;
itemCoord(item,5,1) = -5.0;    itemCoord(item,5,2) = -5.0;


item = 2;  % 1nd item is the airfoil
% Instead of entering the coordinates of all points of the airfoil one
% by one, we can read them from a text file. We assume that each line of
% the file has x and y coordinates of nAirfoilEdge+1 points.
file = fopen('NACA4412.txt', 'r');

for i = 1:nAirfoilEdge + 1
  dummy = str2num(fgets(file));
  itemCoord(item,i,1) = dummy(1);
  itemCoord(item,i,2) = dummy(2);
end

fclose(file);
```

Coordinates of the outside box

Coordinates of the airfoil





4.5 Exercises

E-4.1. Consider the following 2nd order ODE and boundary conditions

$$-\frac{d^2u}{dx^2} + 2\frac{du}{dx} + u = 40\sin(x) + 40\cos(x) + 2e^x, \qquad 0 \le x \le 5$$

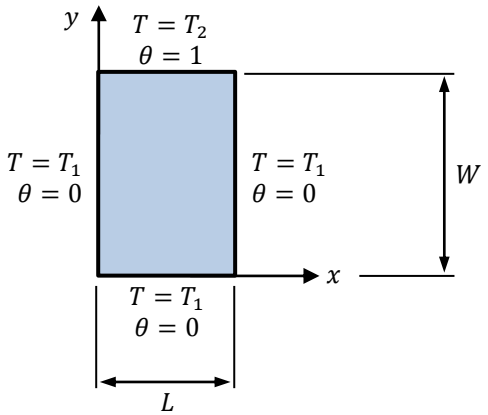
$$u(0) = 1\,, \qquad u(5) = 20\sin(5) + e^5 = 129.2346736093138$$

Its exact solution is $u_{exact}(x) = 20\sin(x) + e^x$.

a) Using steady1D.m code solve this problem with 4, 8, 16, 32, 64, 128, 256, 512, 1024 linear elements. Find the **maximum nodal error in absolute sense** for each run and generate a plot of "Maximum Absolute Nodal Error versus NN (number of nodes)". Use **logarithmic scales** for both the horizontal and vertical axes. You can use the error values already calculated in the `postProcess` function. Obtain a relation on how fast the error drops as the element size is reduced, i.e. a statement like "Order of the error is $\mathcal{O}(h^e)$, i.e. the error drops by a factor of 2 as the element sizes are reduced by a factor of 2". In order to keep errors due to numerical integration low, use 5 point GQ integration (**NGP = 5**).

b) Repeat the error analysis using 2, 4, 8, 16, 32, 64, 128, 256, 512 quadratic elements.

E-4.2. Consider the 2D heat conduction over a thin rectangular plate. Three sides of it are maintained at a constant temperature T_1, while the fourth side is maintained at a constant temperature $T_2 \neq T_1$.



To simplify the solution we introduce the following transformation

$$\theta = \frac{T - T_1}{T_2 - T_1}$$

The governing differential equation in terms of θ is

$$\nabla^2 \theta = 0$$

Exact solution of this problem is given by the following infinite series

$$\theta(x,y) = \frac{2}{\pi} \sum_{n=1}^{\infty} \frac{(-1)^{n+1} + 1}{n} \sin\left(\frac{n\pi x}{L}\right) \frac{\sinh(n\pi y/L)}{\sinh(n\pi W/L)}$$

Using steady2D.m solve this problem for $L = 1$ and $W = 2$ with a series of meshes from coarse to fine. For each solution obtain the temperature profile at $y = W/2$ and compare them with each other and with the exact solution.

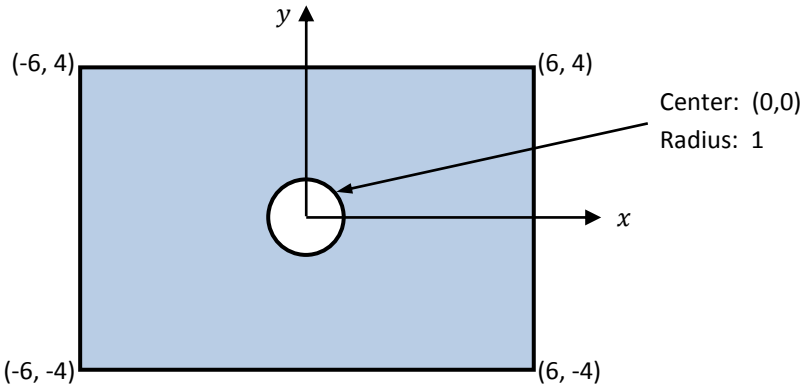
To get the temperature profile at $y = W/2$ of the FEM solution you can write a function called `extractData`. In this function first obtain the array of points at which you want to collect data. For example for this problem it can be an array of 20 points on $y = W/2$ line. Then determine the elements in which these points are located and perform an interpolation with the following equation to get the value of T on the exact points of interest.

$$T^e(x,y) = \sum_{j}^{NEN} T_j^e S_j(\xi,\eta)$$

Note that this equation requires a $(x,y) \rightarrow (\xi,\eta)$ switch to calculate the shape functions at the desired points.

This problem is taken from reference [1].

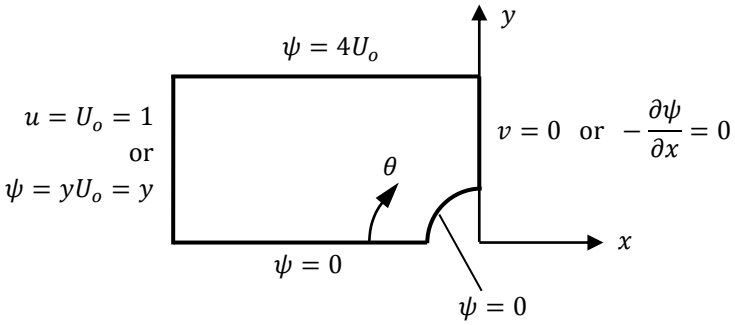
E-4.3. Laplace's equation can be used to study potential flow problems. We want to solve the potential flow over a circular cylinder as shown below.



The relation between streamfunction and the velocity components are given as

$$u = \frac{\partial \psi}{\partial y} \quad \text{and} \quad v = -\frac{\partial \psi}{\partial x}$$

Using the symmetry of the flow field we'll only use a quarter of the domain as shown below. Necessary BCs are also shown, where U_o is the constant inlet velocity that we'll consider to be 1. Note that $\psi = 0$ selection at the bottom boundary and on the cylinder surface is an arbitrary selection. Streamfunction could be equated to some other value here, but using zero is simple enough.

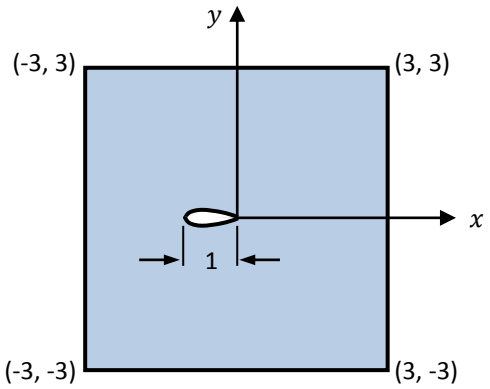


Create an input file for the problem and use steady2D.m code to obtain the streamfunction distribution. Using the relations between the velocity components and the streamfunction, calculate the velocity components and the speed ($V = \sqrt{u^2 + v^2}$) at the points on the cylinder (When you use generate2DinputFile.m code, it may be a good idea to assign a unique BC number to the cylinder to identify the points located on them easily).

To calculate the velocity components and speed at the nodes you can add a new function to the code and call it from the post processing function. Every node on the mesh is surrounded by a number of elements. For each node you need to find the list of the surrounding elements, calculate the velocity components on that node using the streamfunction distribution of each surrounding element and take the average of the values obtained by all surrounding elements.

Plot these speed values of the nodes on the cylinder as a function of θ, which is shown above. Repeat the solution for a coarse and a fine mesh and compare the results. Also compare the results with a reference solution that you can find in any introductory level fluid mechanics textbook.

E-4.4. NACA 0012 is a symmetric airfoil. We want to use the potential flow theory to obtain the pressure distribution over it. First obtain the coordinates of the NACA 0012 airfoil and if necessary modify them so that the chord length of it is equal to 1 and its trailing edge is located at the origin of the coordinate system, as shown below. Put the airfoil in a box of size 6x6 and generate a mesh for the flow field. The same BC of $\psi = y$ can be applied all over the outer boundaries, considering that the free stream velocity is 1. On the airfoil surface the BC is $\psi = 0$.



After obtaining the stremfunction values at the mesh nodes, use the technique explained in the previous problem to determine the speed at each node. Speed values corresponding to the nodes of the airfoil can be used in a Bernoulli equation to get the pressure distribution over the airfoil. Bernoulli equation needs to be written between a point on the surrounding box, called point ∞, and a point on the airfoil, called point A, as seen below

$$\frac{p_\infty}{\rho}+\frac{V_\infty^2}{2}=\frac{p_A}{\rho}+\frac{V_A^2}{2}$$

where $V_A=\sqrt{u_A^2+v_A^2}$ is the speed of point A and $V_\infty=1$ is the free stream velocity. The above Bernoulli equation can be arranged to get the following nondimensional pressure coefficient

$$c_p=\frac{p_A-p_\infty}{\frac{1}{2}\rho V_\infty^2}=1-V_A^2$$

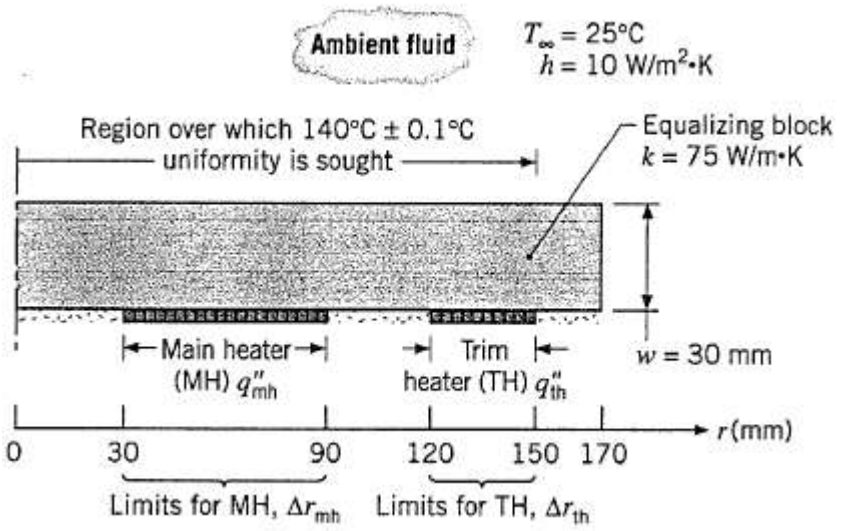
Using the speed values on the airfoil nodes calculate c_p values on them plot c_p vs. x for both the upper and lower surfaces of the airfoil. Compare the results with the reference values. For this symmetric airfoil at zero angle of attack we expect to see the same pressure distribution on the upper and lower surfaces, resulting in no lift.

E-4.5. A semiconductor industry roadmap for microlitography processing requires that a 300 mm diameter silicon wafer be maintained at a steady-state temperature of 140 °C to within a uniformity of 0.1 °C. The design of a hot-plate tool to hopefully meet this requirement is shown schematically. An equalizing block (EB), on which the wafer would be placed, is fabricated from an aluminum alloy of thermal conductivity $k=75\ \mathrm{W/(mK)}$ and is heated by two ring-shaped electrical heaters. The two-zone heating arrangement allows for independent control of a main heater (MH) and a trim heater (TH), which is used to improve the uniformity of the surface temperature for the EB. Your assignment is to size the heaters, MH and TH, by specifying their applied heat fluxes, q''_{mn} and q''_{mn} $(\mathrm{W/m^2})$, and their radial extents, Δr_{mh} and Δr_{th}. The constraints on radial positioning of the heaters are imposed by manufacturing considerations and are shown in the schematic.

EB has a diameter of 340 mm (temperature uniformity is required only on 300 mm diameter portion of it) and its top and lateral surfaces are exposed to convective heat transfer. Lower surface of the EB is adiabatic, except the two heaters.

To obtain initial estimates of heater fluxes, first perform a solution with the upper surface of the EB fixed at 140 °C with heaters fully extending to their specified radial limits. Then use these estimates with convective heat transfer BC at the upper surface of the EB and see if the uniformity is satisfied or not. If not change the heater fluxes and their sizes and positions to get the desired uniformity. Plot the upper surface temperature of different trials.

This problem is taken from reference [1].



References

[1] F. P. Incropera, D. P. Dewitt, T. L. Bergman, A. S. Lavine, Fundamentals of Heat and Mass Transfer, 6th ed., John Wiley and Sons, 2007.