# Chapter 8

# Computer Implementation of 2D, Incompressible N-S Solver

## 8.1 MATLAB Code for 2D, Incompressible N-S Solver (steadyNavierStokes2D.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                           steadyNavierStokes2D.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                                               %
%                 http://www.me.metu.edu.tr/people/cuneyt                       %
%  Last Update  : 30/04/2012                                                    %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                         FEATURES and LIMITATIONS                              %
%                                                                               %
% This code                                                                     %
% - is written for educational purposes.                                       %
% - solves 2D, steady, incompressible Navier-Stokes equations.                 %
% - stores the global stiffness matrix as a sparse matrix using                %
%   coordinate storage format.                                                  %
% - uses Galerkin Finite Element Method (GFEM).                                %
% - uses Lagrange type, triangular or quadrilateral elements, with             %
%   support for Taylor-Hood elements.                                           %
% - reads the problem data and mesh from an input file.                        %
% - generates an output file to be opened by the Tecplot software.             %
%                                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                               %
%                               HOW TO USE?                                     %
%                                                                               %
% This code needs an input file with an extension "inp" to run. You can        %
% generate it by hand or use generate2DinputFileNS.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                                            %
% NCN           : Number of corner nodes of the mesh                            %
% NN            : Number of both corner and mid-face nodes, if there are        %
%                 any                                                           %
% NEC           : Number of element corners                                     %
%                 3: Triangular, 4: Quadrilateral                               %
% NENv          : Number of velocity nodes on an element.                       %
% NENp          : Number of pressure nodes on an element.                       %
% NEU           : 2*NENv + NENp                                                 %
```

```
% Ndof          : All velocity and pressures of the solution.            %
% eType         : 1: Quadrilateral, 2: Triangular. Same for all elements %
% funcs         : Structure to hold fx and fy functions                  %
% coord         : Matrix of size NNx2 to hold the x and y coordinates of %
%                 the mesh nodes (both corner and mid-face nodes)        %
% NGP           : Number of GQ points.                                   %
% GQ            : Structure to store GQ points and weights.              %
%                                                                        %
% Sv, Sp        : Shape functions for velocity and pressure evaluated at %
%                 GQ points. Matrices of size NENvxNGP and NENpxNGP      %
% dSv, dSp      : Derivatives of shape functions for velocity and        %
%                 pressure wrt to ksi and eta evaluated at GQ points.    %
%                 Matrices of size 2xNENvxNGP and 2xNENpxNGP             %
%                                                                        %
% GtoLdof       : Holds the information of elements that are used by     %
%                 each dof (unknown). In a sense it is the reverse of    %
%                 LtoGdof of each element. It is necessary for sparse    %
%                 storage of [K]. It is a matriz of size Ndofx? ,ie. its %
%                 second dimension is not known beforehand.              %
%                                                                        %
% BC            : Structure for boundary conditions                      %
%  - nBC        : Number of different BCs specified.                     %
%  - type       : Array of size nBC.                                     %
%                 1: Given velocity, i.e. inlet or wall                  %
%                 2: Outflow                                             %
%  - str        : BC values/functions are stored as strings in this cell %
%                 array of size nBCx2. For boundaries of type 1 1st and  %
%                 2nd columns store u and v velocity components. For     %
%                 boundaries of type 2 1st column stores pressure and    %
%                 2nd column is not used.                                %
%  - nVelNodes  : Number of mesh nodes where velocity is specified.      %
%  - nOutFaces  : Number of element faces where outflow is specified.    %
%  - velNodes   : Array of nVelNodesx2. 1st column stores global node    %
%                 numbers for which velocity is provided. 2nd column     %
%                 stores the specified BC number.                        %
%  - outFaces   : Array of nOutFacesx3. 1st and 2nd columns store        %
%                 element and face numbers for which NBC is given. 3rd   %
%                 column is used to store which BC is specified.         %
%  - pNode      : Node number where pressure is fixed to zero. Provide a %
%                 negative value in the input file if you do not want to %
%                 fix pressure at a point.                               %
%                                                                        %
% elem          : Structure for elements                                 %
%  - he         : Size of the element                                    %
%  - LtoGnode   : Local to global node mapping array of size NENvx1      %
%  - LtoGdof    : Local to global DOF mapping array of size NEUx1        %
%  - Ke         : Elemental stiffness matrix of size NEUxNEU             %
%  - KeKsparseMap : Matrix of size NEUxNEU to store the location of      %
%                 each entry of elemental Ke in sparsely stored global   %
%                 stiffness matrix.                                      %
%  - Fe         : Elemental force vector of size NEUx1                   %
%  - gDSv       : Derivatives of shape functions for velocity wrt x and  %
%                 y at GQ points. Size is 2xNENvxNGP                     %
%  - gDSp       : Derivatives of shape functions for pressure wrt x and  %
%                 y at GQ points. Size is 2xNENpxNGP                     %
%  - detJacob   : Determinant of the Jacobian matrix evaluated at a      %
%                 certain (ksi, eta)                                     %
%                                                                        %
% soln          : Structure for the global system of equations           %
%  - Ksparse    : Structure for storing the global stiffness matrix of   %
%                 NdofxNdof in sparse format. It uses coordinate type    %
%                 sparse storage.                                        %
%    - NNZ      : Number of nonzero entries in global stiffness matrix.  %
%    - value    : Nonzero values of the stiffness matrix. Array of size  %
%                 NNZ.                                                   %
%    - row      : Rows of the nonzeros of the stiffness matrix. Array of %
%                 size NNZ.                                              %
%    - col      : Columns of the nonzeros of the stiffness matrix. Array %
%                 of size NNZ.                                           %
```

These are the three vectors used to store [K] as a sparse matrix using coordinate storage format.

```
%  - F         : Global force vector of size Ndofx1                    %
%  - U         : Global unknown vector of size Ndofx1                  %
%  - Uprev     : Solution of the previous iteration.                   %
%                                                                      %
% nonlinearIterMax : Maximum number of nonlinear iterations.           %
%                Set it to 1 to obtain Stokes solution.                %
% nonlinearTol : Nonlinear iteration tolerance                         %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                      %
%                              FUNCTIONS                               %
%                           (incomplete list)                          %
%                                                                      %
% steadyNavierStokes2DSparse : Main function                           %
% readInputFile : Read the input file                                  %
% setupLtoGdof  : Create LtoGdof for each element based on LtoGnode    %
%                 values read from the input file                      %
% setupGtoLdof  : Creates GtoLdof. Required for sparse storage         %
% 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          %
% picardIter    : Perform nonlinear iterations of a N-S solution       %
% solve         : Solve the global system of equations                 %
% postProcess   : Generate contour plots of velocity components and    %
%                 pressure                                             %
% createTecplot : Create an output file for Tecplot software           %
%                                                                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%==========================================================================
function [] = steadyNavierStokes2D
%==========================================================================
clc;
clear all;
close all;

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

readInputFile();

timeStart = tic;

calcElemSize();

fprintf('Setting up LtoGdof and GtoLdof ...   ');
setupLtoGdof();
setupGtoLdof();
fprintf('Done.\n');

fprintf('Setting up Ksparse and KeKsparseMap ...   ');
setupKsparse();
setupKeKsparseMap();
fprintf('Done.\n');

setupGQ();
calcShape();
calcJacob();

timePicardStart = tic;
fprintf('Starting nonlinear iterations ...\n');
picardIter();

timePicard = toc(timePicardStart);
timeTotal = toc(timeStart);

postProcess();
createTecplot();

fprintf('\nTotal run time is  %f seconds.', timeTotal);
fprintf('\nPicard iterations took  %f seconds.\n\n', timePicard);

fprintf('Program is terminated successfully.\n');

% End of function steadyNavierStokes2D()
```

Since a mesh node stores more than one unknown just an LtoG for nodes is not enough, instead an LtoG for unknowns (dofs) is necessary.

GtoL is the reverse of LtoG. It stores the unknowns (dofs) nodes that are associated with a certain element. It is necessary for sparse storage.

Preparation for sparse storage.

Elemental and global systems are constructed and solved inside the Picard iteration loop.

```
%======================================================================
function [] = readInputFile
%======================================================================

% ...
% ...
% ...

% End of function readInputFile()
```

Details of this function are not important.

```
%======================================================================
function [] = calcElemSize
%======================================================================
global eType NE coord elem;

% Calculates the diameter of the circumcircle around the triangle or the
% quadrilateral. It is used for GLS stabilization.

% ...
% ...
% ...

% End of function calcElemSize()
```

Details of this function are not important.

```
%======================================================================
function [] = setupLtoGdof
%======================================================================
global NE NN NENv NENp elem;

% Sets up LtoGdof for each element using LtoGnode. As an example LtoGdof
% uses the following local dof ordering for a quadrilateral element with
% NENv = 9 and NENp = 4
%
% u1, u2, u3, u4, u5, u6, u7, u8, u9, v1, v2, v3, v4, v5, v6, v7, v8, v9,
% p1, p2 , p3, p4

for e = 1:NE
  dofCounter = 0;

  % u velocity DOFs
  for i = 1:NENv
    dofCounter = dofCounter + 1;
    elem(e).LtoGdof(dofCounter) = elem(e).LtoGnode(i);
  end

  % v velocity DOFs
  for i = 1:NENv
    dofCounter = dofCounter + 1;
    elem(e).LtoGdof(dofCounter) = NN + elem(e).LtoGnode(i);
  end

  % pressure DOFs
  for i = 1:NENp
    dofCounter = dofCounter + 1;
    elem(e).LtoGdof(dofCounter) = 2 * NN + elem(e).LtoGnode(i);
  end
end

% End of function setupLtoGdof()
```

LtoG used in previous codes is now called LtoGnode. It is extended to LetoGdof using the fact that in global unknown numbering first all the u velocity components, than the v velocity components and finally the pressure values come.

```
%=========================================================================
function [] = setupGtoLdof
%=========================================================================
global NE NEU elem Ndof GtoLdof;

% GtoLdof stores information about which elements are using which DOFs.
% The information is stored in sparse form using two vectors
% GtoLdof.RowStart and GtoLdof.Col.

% We'll first store GtoLdofTemp information into a matrix of size Ndofx4.
% It is faster to do it first in this way and then tansfer it into a sparse
% format. Second dimension of it is set to be 4, but it will grow inside
% the following loop if necessary.
GtoLdofTemp = zeros(Ndof,4);

% Allocate a temporary array that is used to store the number of elements
% each dof is connected to.
dofElementCount = zeros(Ndof,1);

% Form GtoLdofTemp using LtoGdof of each element
for e = 1:NE
  for i = 1:NEU
    dof = elem(e).LtoGdof(i);
    dofElementCount(dof) = dofElementCount(dof)+1;
    GtoLdofTemp(dof,dofElementCount(dof)) = e;
  end
end

% Store the information in GtoLdofTemp in sparse format similar to
% compressed row storage
GtoLdof.RowStart = zeros(Ndof+1,1);
GtoLdof.Col = zeros(NE*NEU,1);

GtoLdof.RowStart(1) = 1;

for i = 2:Ndof+1
  GtoLdof.RowStart(i) = GtoLdof.RowStart(i-1) + dofElementCount(i-1);
end

colCount = 0;

for i = 1:Ndof
  for j = 1:dofElementCount(i)
    colCount = colCount + 1;
    GtoLdof.Col(colCount) = GtoLdofTemp(i,j);
  end
end

% End of function setupGtoLdof()
```

GtoLdof is the opposite of LtoGdof in meaning. Due to efficiency reasons it is storage in a sparse-storage like format. It will be deleted after necessary preparation for sparse storage of [K] is done.

```
%=========================================================================
function [] = setupKsparse
%=========================================================================
global NEU elem Ndof soln GtoLdof;

% Sets up row and column arrays of the global stiffness matrix in sparse
% storage format.

% Calculate number of nonzero entries in the stiffness matrix. This is
% necessary to properly allocate sparse storage vectors. Also rowStart
% values can be determined at this step.

soln.Ksparse.NNZ = 0;  % Counts nonzero entries in [K]

soln.Ksparse.rowStart = zeros(Ndof+1,1);
soln.Ksparse.rowStart(1) = 1;


for r = 1:Ndof     % Row loop.
  nonZeroColumns = zeros(4*NEU,1);   % Initialization
  colCount = 0;
  colCount2 = 0;

  for i = GtoLdof.RowStart(r):GtoLdof.RowStart(r+1)-1
    e = GtoLdof.Col(i);   % This element contributes to row r.
    for j = 1:NEU
      colCount = colCount + 1;
      nonZeroColumns(colCount) = elem(e).LtoGdof(j);
    end
  end

  dummy = zeros(Ndof,1);

  % Remove duplicate entries of nonZeroColumns
  colCount2 = 0;
  for i = 1:colCount
    col = nonZeroColumns(i);
    if dummy(col) == 0
      dummy(col) = 1;
      colCount2 = colCount2 + 1;
    end
  end

  soln.Ksparse.NNZ = soln.Ksparse.NNZ + colCount2;
  soln.Ksparse.rowStart(r+1) = soln.Ksparse.rowStart(r) + colCount2;

end



% Allocate memory for three vectors of soln.Ksparse
soln.Ksparse.value = zeros(soln.Ksparse.NNZ,1);
soln.Ksparse.col = zeros(soln.Ksparse.NNZ,1);
soln.Ksparse.row = zeros(soln.Ksparse.NNZ,1);



% We'll find which columns of row r of [K] have non-zero entries
% This part has some repetitions with the previous loop.

NNZcounter = 0;

for r = 1:Ndof
  nonZeroColumns = zeros(4*NEU,1);     % Initialization
  nonZeroColumns2 = zeros(4*NEU,1);    % Initialization
  colCount = 0;
  colCount2 = 0;
```

Coordinate storage format uses three vectors, row, col and value. Each of these have the size of the number of non zero entries of [K]. row and col vectors storage the row and column position of each nonzero entry and value vector stores the actual nonzero values.

In this function NNZ of [K] is calculated. Also row and col vectors aregenerated.

```
  for i = GtoLdof.RowStart(r):GtoLdof.RowStart(r+1)-1
    e = GtoLdof.Col(i);    % This element contributes to row r.
    for j = 1:NEU
      colCount = colCount + 1;
      nonZeroColumns(colCount) = elem(e).LtoGdof(j);
    end
  end

  dummy = zeros(Ndof,1);

  % Remove duplicate entries of nonZeroColumns
  colCount2 = 0;
  for i = 1:colCount
    col = nonZeroColumns(i);
    if dummy(col) == 0
      dummy(col) = 1;
      colCount2 = colCount2 + 1;
      nonZeroColumns2(colCount2) = col;
    end
  end

  nonZeroColumns2 = nonZeroColumns2(1:colCount2);
  nonZeroColumns2 = sort(nonZeroColumns2);

  for k = 1:colCount2
    kk = NNZcounter + k;
    soln.Ksparse.row(kk,1) = r;
    soln.Ksparse.col(kk,1) = nonZeroColumns2(k);
  end
  NNZcounter = NNZcounter + colCount2;

end


clear GtoLdof.RowStart;
clear GtoLdof.Col;
clear GtoLdof;

% End of function setupKsparse()
```

GtoLdof is not necessary anymore, so we delete it.

```
%======================================================================
function [] = setupKeKsparseMap
%======================================================================
global NE NEU elem soln;

% Determine the location of the entries of Ke for all elements in Ksparse
% This will be used in the assembly process.

for e = 1:NE
  elem(e).KeKsparseMap = zeros(NEU,NEU);
  for i = 1:NEU
    r = elem(e).LtoGdof(i);
    col = soln.Ksparse.col(soln.Ksparse.rowStart(r) : ...
                           soln.Ksparse.rowStart(r+1)-1);
    for j = 1:NEU
      c = find(col == elem(e).LtoGdof(j), 1, 'first');
      elem(e).KeKsparseMap(i,j) = c + soln.Ksparse.rowStart(r) - 1;
    end
  end
end

% End of function setupKeKsparseMap()
```

KeKsparseMap stores the information of where each entry of each [Ke] should go in the value vector sparse [K] during the assembly process.

```
%=========================================================================
function [] = setupGQ
%=========================================================================

% ...
% ...
% ...

% End of function setupGQ()
```

Details of this function are not important.

```
%=========================================================================
function [] = calcShape()
%=========================================================================
global eType NGP NENv NENp GQ Sv Sp dSv dSp;

% Calculates the values of the shape functions and their derivatives with
% respect to ksi and eta at GQ points.

% Sv, Sp   : Shape functions for velocity and pressure approximation.
% dSv, dSp : ksi and eta derivatives of Sv and Sp.

% ...
% ...
% ...

% End of function calcShape()
```

Details of this function are not important.

For the case of different NENv and NENp values, pressure and velocity shape functions and their derivatives are stored separately.

```
%=========================================================================
function [] = calcJacob()
%=========================================================================
global NE NEC NGP coord dSv dSp elem;

% Calculates Jacobian matrix and its determinant for each element. Shape
% functions for pressure nodes (Sp) are used for Jacobian calculation.
% Also calculates and stores the derivatives of velocity shape functions
% wrt x and y at GQ points for each element.

% ...
% ...
% ...

% End of function calcGlobalSys()
```

Details of this function are not important.

```
%=========================================================================
function [] = calcElemSys(e)
%=========================================================================
global eType NEC NENv NENp NEU NGP coord GQ elem funcs soln density viscosity;
global Sp Sv;

% Calculates the element level stiffness matrix and force vector.

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

% Define 3 sub-Fe vectors and 9 sub-Ke matrices.
Fe1 = zeros(NENv, 1);
Fe2 = zeros(NENv, 1);
Fe3 = zeros(NENp, 1);

Ke11 = zeros(NENv, NENv);
Ke12 = zeros(NENv, NENv);
Ke13 = zeros(NENv, NENp);
% Ke21 is the transpose of Ke12
Ke22 = zeros(NENv, NENv);
Ke23 = zeros(NENv, NENp);
Ke31 = zeros(NENp, NENv);
Ke32 = zeros(NENp, NENv);
Ke33 = zeros(NENp, NENp);
```

These are the mvectors and matrices defined in Eqn (7.28).

```
% Extract elemental u and v velocity values from the global solution
% solution array of the previous iteration.
for i = 1:NENv
  iG = elem(e).LtoGdof(i);
  u0_nodal(i) = soln.U(iG);

  iG = elem(e).LtoGdof(i + NENv);
  v0_nodal(i) = soln.U(iG);
end
```

Due to nonlinear terms, [K] depends on the velocity unknowns of the previous iteration level. u0_nodal and v0_nodal are the previous solution at the nodes of the current element e. They are extracted from the global unknown vector soln.U.

```
for k = 1:NGP   % Gauss Quadrature loop
  % Above calculated u0 and v0 values are at the nodes. However in GQ
  % integration we need them at GQ points. Let's calculate them using
  % interpolation based on shape functions.
  u0 = 0;
  v0 = 0;
  for i = 1:NENv
    u0 = u0 + Sv(i,k) * u0_nodal(i);
    v0 = v0 + Sv(i,k) * v0_nodal(i);
  end
```

Previous velocity components at the k-th GQ point is calculated using u0_nodal and v0_nodal values.

```
  %   Uncomment the following part if fx, fy functions are nonzero and
  %   they depend on x, y coordinates.

  %   % Using isoparametric formulation idea, calculate global x and y
  %   % coordinates that corresponds to the current GQ point.
  %   x = 0;
  %   y = 0;
  %   for i = 1:NENp
  %     iG = elem(e).LtoGnode(i);
  %     x = x + shape.Sp(i,k)*coord(iG,1);
  %     y = y + shape.Sp(i,k)*coord(iG,2);
  %   end
  %
  %   fxValue = eval(funcs.fx);
  %   fyValue = eval(funcs.fy);
```

```
% Define shortcuts
gDSv(:,:) = elem(e).gDSv(:,:,k);
gDSp(:,:) = elem(e).gDSp(:,:,k);

factor = elem(e).detJacob(k) * GQ.weight(k);

for i = 1:NENv
  for j = 1:NENv
    Ke11(i,j) = Ke11(i,j) + (viscosity * ( ...
                2 * gDSv(1,i) * gDSv(1,j) + ...
                gDSv(2,i) * gDSv(2,j)) + ...
                density * Sv(i,k) * ...
                (u0 * gDSv(1,j) + v0 * gDSv(2,j))) * factor;

    Ke12(i,j) = Ke12(i,j) + viscosity * gDSv(2,i) * gDSv(1,j) * factor;

    Ke22(i,j) = Ke22(i,j) + (viscosity * ( ...
                gDSv(1,i) * gDSv(1,j) + ...
                2 * gDSv(2,i) * gDSv(2,j)) + ...
                density * Sv(i,k) * ...
                (u0 * gDSv(1,j) + v0 * gDSv(2,j))) * factor;
  end

  for  j = 1:NENp
    Ke13(i,j) = Ke13(i,j) - gDSv(1,i) * Sp(j,k) * factor;

    Ke23(i,j) = Ke23(i,j) - gDSv(2,i) * Sp(j,k) * factor;
  end
end

%  Uncomment the folowing lines if fx, fy are nonzero.
%  for i = 1:NENv
%    Fe1(i) = Fe1(i) + density * Sv(i) * fxValue * ...
%             elem(e).detJacob * GQ.weight(k);
%
%    Fe2(i) = Fe2(i) + density * Sv(i) * fyValue * ...
%             elem(e).detJacob * GQ.weight(k);
%  end
```

Eqns (7.13) and (7.29).

```
Ke31 = Ke13';
Ke32 = Ke23';
```

Certain submatrices of [Ke] are transpose of each other.

```
% Apply GLS stabilization for linear elements with NENv = NENp
if (eType == 1 && NENv == 4 && NENp == 4) || ...
   (eType == 2 && NENv == 3 && NENp == 3)

  Tau = 1/12 * (elem(e).he) ^ 2 / viscosity;    % GLS parameter

  for i = 1:NENv
    for j = 1:NENv
      Ke11(i,j) = Ke11(i,j) + Tau * density * density * ...
                  (u0 * gDSv(1,i) + v0 * gDSv(2,i)) * ...
                  (u0 * gDSv(1,j) + v0 * gDSv(2,j)) * factor;

      Ke22(i,j) = Ke22(i,j) + Tau * density * density * ...
                  (u0 * gDSv(1,i) + v0 * gDSv(2,i)) * ...
                  (u0 * gDSv(1,j) + v0 * gDSv(2,j)) * factor;
    end
  end

  for i = 1:NENv
    for  j = 1:NENp
      Ke13(i,j) = Ke13(i,j) + Tau * density * ...
                  (u0 * gDSv(1,i) + v0 * gDSv(2,i)) * gDSp(1,j) * factor;
```

Eqn (7.30) for GLS stabilization.

GLS is used only if NENv and NENp are equal.

```
        Ke23(i,j) = Ke23(i,j) + Tau * density * ...
                    (u0 * gDSv(1,i) + v0 * gDSv(2,i)) * gDSp(2,j) * factor;
      end
    end


    for i = 1:NENp
      for j = 1:NENv
        Ke31(i,j) = Ke31(i,j) - Tau * density * ...
                    gDSp(1,j) * (u0 * gDSp(1,i) + v0 * gDSp(2,i)) * factor;

        Ke32(i,j) = Ke32(i,j) - Tau * density * ...
                    gDSp(2,j) * (u0 * gDSp(1,i) + v0 * gDSp(2,i)) * factor;
      end
    end


    for i = 1:NENp
      for j = 1:NENp
        Ke33(i,j) = Ke33(i,j) - Tau * ...
                    (gDSp(1,i) * gDSp(1,j) + gDSp(2,i) * gDSp(2,j)) * factor;
      end
    end

    % There should be lines here to modify elemental force vector if fx
    % and/or fy terms are nonzero.

  end % End of GLS stabilization

end  % End of GQ loop


% Form the elemental Ke and Fe by putting sub-Ke and sub-Fe's together.
elem(e).Ke = [Ke11 Ke12 Ke13; Ke12' Ke22 Ke23; Ke31 Ke32 Ke33];
elem(e).Fe = [Fe1; Fe2; Fe3];

% End of function calcElemSys()
```

Eqn (7.30) for GLS stabilization (cont'd).

Form [Ke] and {Fe}.

```
%=====================================================================
function [] = assemble(e)
%=====================================================================
global NEU elem soln;

% Inserts [Ke] and {Fe} into proper locations of [K] and {F} by the help
% of LtoGdof and KeKsparseMap variables of elements.

for i = 1:NEU
  iG = elem(e).LtoGdof(i);  % iG is the global dof corresponding to the
                            % i-th local dof of element e
  soln.F(iG) = soln.F(iG) + elem(e).Fe(i);
  for j = 1:NEU
    soln.Ksparse.value(elem(e).KeKsparseMap(i,j)) = ...
       soln.Ksparse.value(elem(e).KeKsparseMap(i,j)) + elem(e).Ke(i,j);
  end
end

% End of function assemble()
```

Assemble process makes use of LtoGdof and KeKapsrseMap of each element.

```
%==========================================================================
function [] = picardIter()
%==========================================================================
global NN Ndof nonlinearIterMax nonlinearTol soln;

% Do the necesary memory allocations
soln.U     = zeros(Ndof,1);
soln.Uprev = zeros(Ndof,1);  % Stores the solution of the previous
                             % iteration.

% Previous solution is necessary for error calculation.
soln.Uprev = soln.U;

% Start the Picard iterations loop
for iter = 1:nonlinearIterMax
  calcGlobalSys();
  applyBC();
  solve();


  % Calculate the maximum difference of velocity components between the
  % current solution and the previous solution.
  errorMax = max(abs(soln.U(1:2*NN) - soln.Uprev(1:2*NN)));

  fprintf('Iter = %4d ,   errorMax =  %10.5e \n', iter, errorMax);
  if errorMax < nonlinearTol
    break;
  end

  % The last solution is copied to the previous one for the next iteration.
  soln.Uprev = soln.U;
end

% This part limits the number of iterations.
if iter >= nonlinearIterMax
  fprintf('Solution did not converge in %d iterations.\n', nonlinearIterMax);
else
  fprintf('\nConvergence is achieved at %d iterations. \n', iter);
end

% End of picardIter() function
```

A global system is generated and solved for each Picard iteration.

Check convergence of Picard iterations by comparing current and previous velocity values.

```
%=========================================================================
function [] = applyBC()
%=========================================================================
global NN soln BC coord;

% For velocity BCs reduction is NOT applied. Instead, global [K] and {F}
% are modified.

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

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

  % Change [K] and {F} for the given u velocity.
  uVel = eval(char(BC.str(whichBC,1)));
  soln.F(node) = uVel;

  rowStart = soln.Ksparse.rowStart(node);
  rowEnd = soln.Ksparse.rowStart(node + 1)-1;
  for j = rowStart:rowEnd
    if node == soln.Ksparse.col(j)
      soln.Ksparse.value(j) = 1.0;  % Equate diagonal entry to 1.
    else
      soln.Ksparse.value(j) = 0.0;  % Equate off-diagonal entries to zero.
    end
  end

  % Change [K] and {F} for the given v velocity.
  dof = node + NN;     % v velocities are globally numbered after u
                       % velocities.
  vVel = eval(char(BC.str(whichBC,2)));
  soln.F(dof) = vVel;

  rowStart = soln.Ksparse.rowStart(dof);
  rowEnd = soln.Ksparse.rowStart(dof + 1)-1;
  for j = rowStart:rowEnd
    if dof == soln.Ksparse.col(j)
      soln.Ksparse.value(j) = 1.0;  % Equate diagonal entry to 1.
    else
      soln.Ksparse.value(j) = 0.0;  % Equate off-diagonal entries to zero.
    end
  end

end

% Modify [K] and {F} for the node where pressure is set to zero.
node = BC.pNode;      % Node at which pressure is set to zero.

if (node > 0) % If node is negative it mena swe do not set
              % pressure to zero at  node.
  dof = node + 2 * NN;     % Pressures are globally numbered after u
                           % and v velocities.
  soln.F(dof) = 0.0;

  rowStart = soln.Ksparse.rowStart(dof);
  rowEnd = soln.Ksparse.rowStart(dof + 1)-1;
  for j = rowStart:rowEnd
    if dof == soln.Ksparse.col(j)
      soln.Ksparse.value(j) = 1.0; % Equate diagonal entry to 1.
    else
      soln.Ksparse.value(j) = 0.0;  % Equate off-diagonal entries to zero.
    end
  end
end
% End of function applyBC()
```

Change [K] and {F} for the given u velocity components at inlets and walls.

Change [K] and {F} for the given v velocity components at inlets and walls.

Change [K] and {F} for the point where pressure is fixed to zero.

The code support s no NBC for pressure.

```
%=========================================================================
function [] = solve()
%=========================================================================
global soln Ndof;

% 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.


% First form a sparse matrix using soln.Ksparse
A = sparse(soln.Ksparse.row, soln.Ksparse.col, soln.Ksparse.value, ...
           Ndof, Ndof);

soln.U = A \ soln.F;

% End of function solve()
```

First a sparse matrix is formed using row, col and value vectors of [K]. Than the global system is solved using the backslash operator.

```
%=========================================================================
function [] = postProcess()
%=========================================================================
% Generates contour plot of u, v, and pressure and streamline plot. Only
% the data at element corners are used even if there are velocity nodes at
% mid faces.


% ...
% ...
% ...

% End of function postProcess()
```

Contour plots for velocity components and pressure are generated.

```
%=========================================================================
function [] = createTecplot()
%=========================================================================


% ...
% ...
% ...


% End of function createTecplot()
```

An output file with DAT extension is generated to be used with the commercial Tecplot software.

```
%=========================================================================
function [] = createTecplotSimple()
%=========================================================================


% ...
% ...
% ...


% End of function createTecplotSimple()
```

## 8.1 Sample 2D Input File for Steady N-S Solver (CavityNE864Tri_TH.inp)

The following input file is for the lid-driven cavity problem. It is generated using generate2DinputFileNS.m code. It has 864 Taylor Hood triangle elements each having 6 velocity and 3 pressure nodes.

```
This input file is created by generate2DinputFileNS.m code
================================================
eType     : 2
NE        : 864
NCN       : 485
NN        : 1833
NENv      : 6
NENp      : 3
NGP       : 4
iterMax   : 20
tolerance: 0.000001
density   : 1.0
viscosity: 0.01
fx        : 0.0
fy        : 0.0
================================================
Node No           x               y
      1       0.0000000       0.0000000
      2       0.0000000       0.0312500
      3       0.0000000       0.0625000
       . . . .
       . . . .
       . . . .
       . . . .
================================================
Elem No   node1   node2   node3   ......
      1     244     233     249     486     487     488
      2     227     206     204     489     490     491
      3     249     233     221     487     492     493
       . . . .
       . . . .
       . . . .
       . . . .
================================================
BCs (Number of specified BCs, their types and strings)
nBC       : 2
BC 1      : 1  0.0 : 0.0
BC 2      : 1  1.0 : 0.0
================================================
nVelNodes : 208
nOutFaces : 0
================================================
Velocity BC (Node#  BC#)
1  1
2  1
. . . .
. . . .
================================================
Outflow BC (Elem# Face# BC#)
================================================
Node number where pressure is taken to be zero
1
```

NCN stands for Number of Corner Nodes, i.e. the number of nodes where pressure is stored.

$x$ and $y$ coordinates of 1833 nodes.

LtoG for 864 elements.

Each element has 6 velocity nodes.

Lid-driven cavity problem has 2 EBCs. First one is for the fixed walls and the second one is for the sliding lid.

Totally there are 208 nodes where velocity is specified. And there is no face with outflow BC.

Pressure is fixed to zero at node 1, which is the lower left corner of the cavity.