Name	Ahmed Saeed Mohamed Sherif
Course	Finite Elements in Fluids
Master	MSc Computational Mechanics
Report no.	5
Topics	Incompressible flow (Stokes and Cavity flows)

Contents

1 Stokes flow

1.1 Mesh convergence for the elements Q2Q1 and Q2Q0

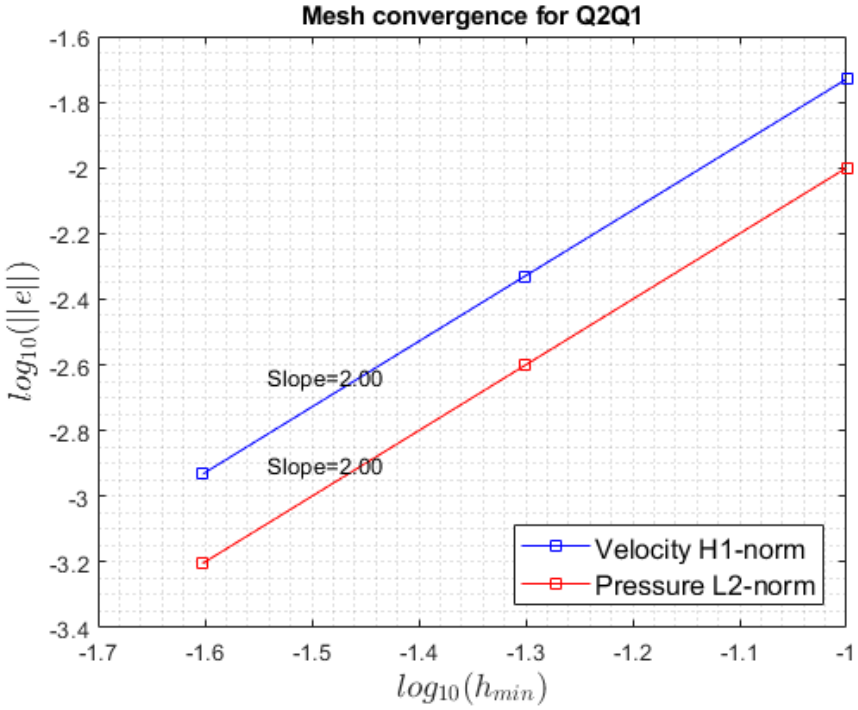


Figure 1: Mesh convergence for the elements Q2Q1

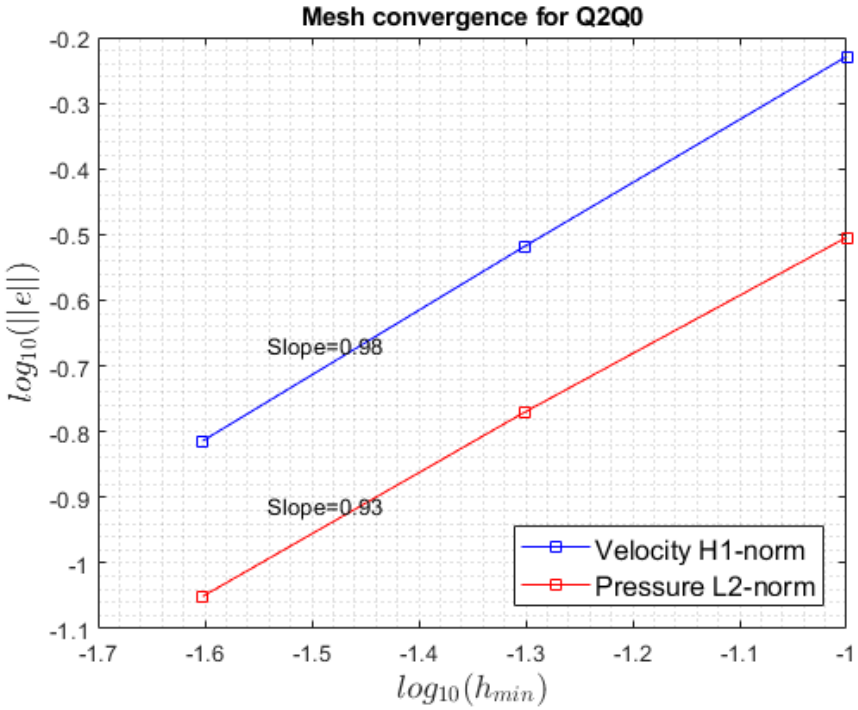


Figure 2: Mesh convergence for the elements Q2Q0

It is seen from Figures 1 and 2 that the errors behave as expected. Using the element Q2Q1 means that the degree of interpolation of the velocity is $k = 2$ and that of pressure is $l = 1$, meaning that the error bounds are:

$$\|p - p^h\|_{\mathcal{L}_2} \leq C_1 h^{l+1} = C_1 h^2$$

$$\|\boldsymbol{\nabla v} - \boldsymbol{\nabla v}^h\|_{\mathcal{H}^1} \leq C_2 h^k = C_2 h^2$$

where h is the element size. On the other hand, using the element Q2Q0 ($k = 2, l = 0$) doesn't follow the previous error bound for the velocity because the error in the velocity is dominated by the lower order approximation for the pressure, so the error bounds are as follows:

$$\|p - p^h\|_{\mathcal{L}_2} \leq C_1 h^{l+1} = C_1 h^1$$

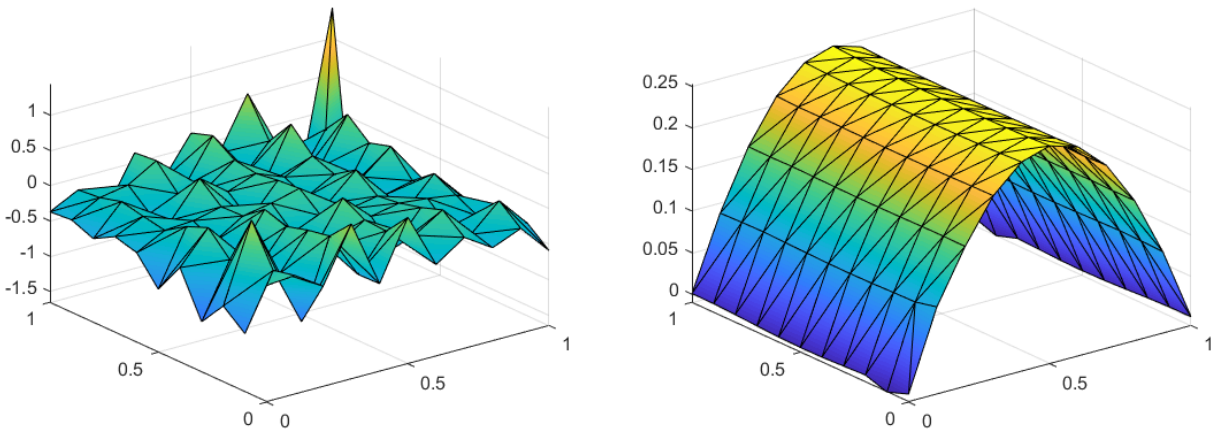
$$\|\boldsymbol{\nabla v} - \boldsymbol{\nabla v}^h\|_{\mathcal{H}^1} \leq C_2 h^1$$

1.2 Stabilized GLS formulation for the element P1P1

Using elements with equal order interpolations for both velocity and pressure yields unstable solution for the pressure when the standard Galerkin formulation is considered. For this, stabilized formulations such as Galerkin/Least-squares (GLS) is used. Following the steps explained in [1] for linear elements, the following stabilization term is added to the mass conservation equation:

$$-\sum_{e=1}^{\mathtt{n_{el}}} \tau_e (\boldsymbol{\nabla} q^h, \boldsymbol{\nabla} p^h - \boldsymbol{b}^h)_{\Omega_e}$$

Figure 3 shows the solution of the pressure obtained using the element P1P1 employing both the Galerkin and the GLS formulations. Figure 4 shows the convergence plot of the velocity and pressure. It can be seen that the optimal convergence rates are achieved.



(a) Galerkin (b) GLS

Figure 3: The pressure solution on a mesh of elements P1P1 using Galerkin (left) and GLS (right) formulations

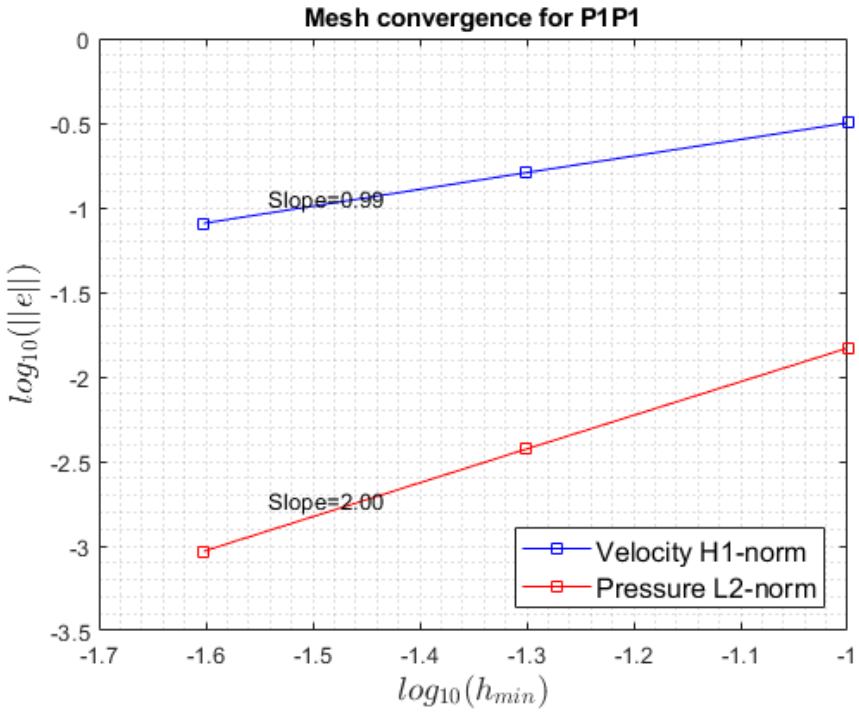
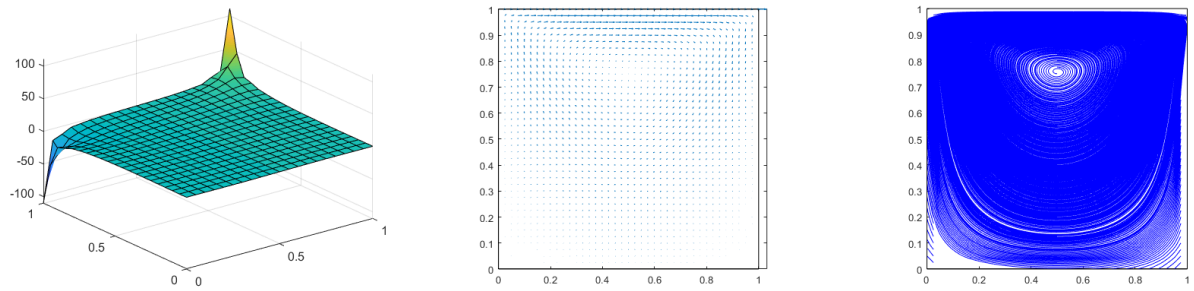


Figure 4: Mesh convergence for the elements P1P1 using GLS formulation

2 Cavity flow

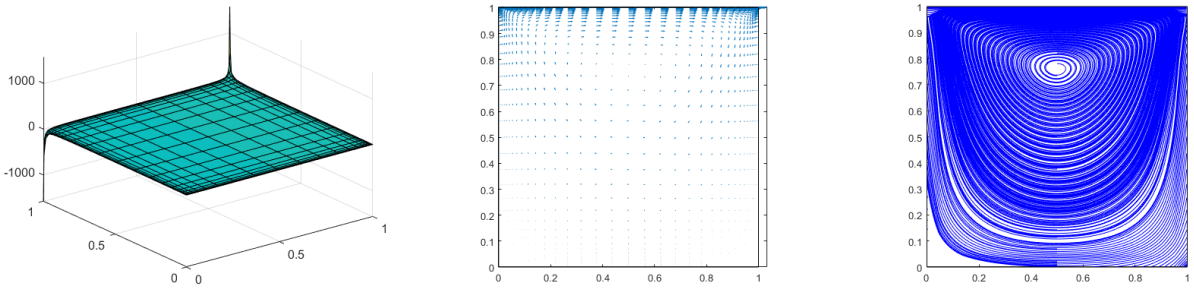
2.1 Comparison between uniform mesh and refined mesh near the walls

The cavity flow problem has a pressure field that is constant everywhere with a point discontinuity at the two upper corners as shown in Figures 5a and 6a. The velocity field is a circulating vortex as seen in Figures 5b and 6b or as observed from the streamlines in Figures 5c and 6c.



(a) Pressure (b) Velocity vectors (c) Streamlines

Figure 5: The solution of the cavity flow problem using uniform mesh of Q2Q1 elements



(a) Pressure (b) Velocity vectors (c) Streamlines

Figure 6: The solution of the cavity flow problem using adaptively-refined mesh of Q2Q1 elements

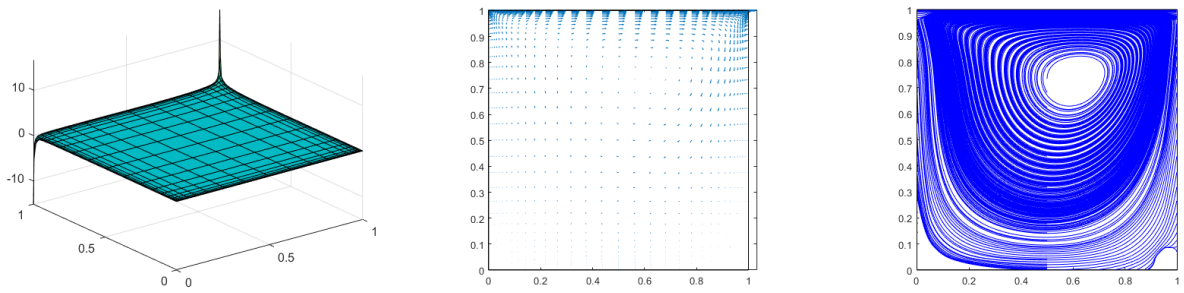
The difference in the solution obtained using the uniform mesh and the adaptively-refined mesh is clearly seen. The adaptively-refined mesh near the walls gives a more accurate solution when compared to the uniform mesh because the refinement near the walls allows to capture the point discontinuity in the pressure at the upper corners easily, which in turns results in more accurate results for both the pressure and the velocity.

2.2 Solving Navier-Stokes equations with Picard method

The results for the cavity flow obtained by solving the steady Navier-Stokes equation employing a standard Galerkin formulation on an adaptively-refined mesh near the wall with elements Q2Q1 are presented for Reynolds numbers $Re = 100, 500, 1000, 2000$ in Figures 7, 8, 9 and 10, respectively.

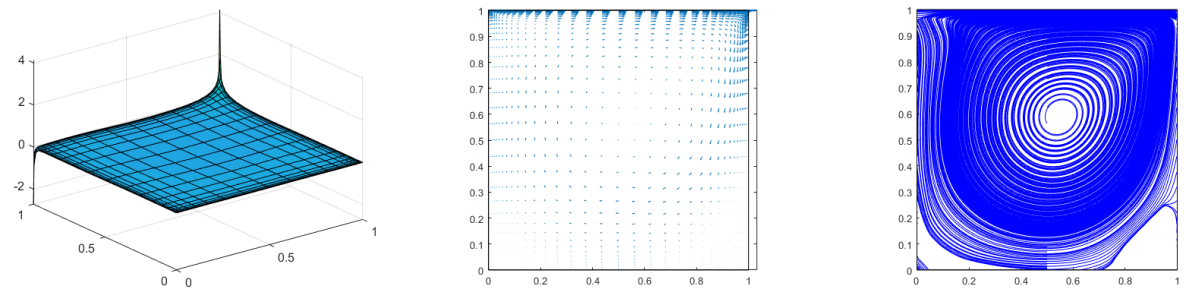
By increasing the Reynolds number, the boundary layers are more obvious and the velocity profile become sharper. Furthermore, the position of the main vortex moves towards the center of the cavity as the Reynolds number increases. A secondary vortex starts to appear at the right bottom corner of the cavity and a third vortex appears at the lower left corner.

The number of iterations of Picard method required to achieve convergence is shown in Table 1 for the different values values of Re. It is noticed that more iterations are needed for large values of the flow Reynolds number, implying the introduced difficulty at high Reynolds number. At the end, the results are matching with those presented in literature for $Re = 100$ and $Re = 1000$, see [2].



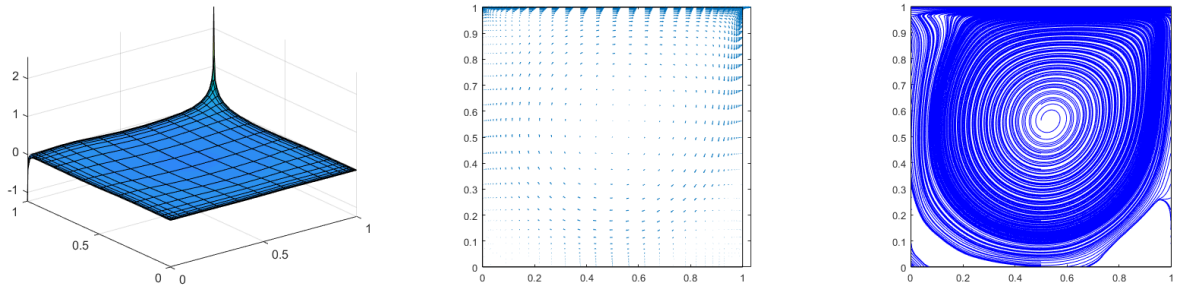
(a) Pressure (b) Velocity vectors (c) Streamlines

Figure 7: The solution of the cavity flow problem (with convection) using adaptively-refined mesh of Q2Q1 elements - $Re = 100$



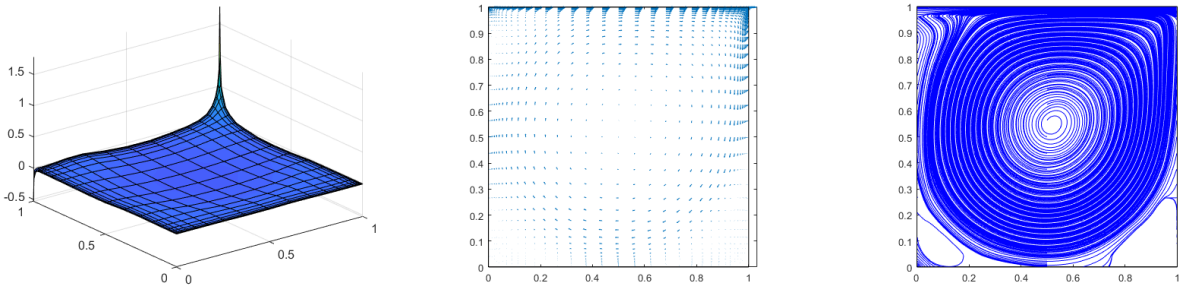
(a) Pressure (b) Velocity vectors (c) Streamlines

Figure 8: The solution of the cavity flow problem (with convection) using adaptively-refined mesh of Q2Q1 elements - $Re = 500$



(a) Pressure (b) Velocity vectors (c) Streamlines

Figure 9: The solution of the cavity flow problem (with convection) using adaptively-refined mesh of Q2Q1 elements - $Re = 1000$



(a) Pressure (b) Velocity vectors (c) Streamlines

Figure 10: The solution of the cavity flow problem (with convection) using adaptively-refined mesh of Q2Q1 elements - $Re = 2000$

Re	n_{iters}
100	14
500	30
1000	36
2000	70

Table 1: Number of iterations of Picard method until convergence for different Reynolds numbers

2.3 Solving Navier-Stokes equations with Newton-Raphson method

In order to formulate the Newton-Raphson iterative scheme we first write the problem in a residual form as:

$$\boldsymbol{\mathcal{R}} = \begin{Bmatrix} \boldsymbol{\mathcal{R}}^u \\ \boldsymbol{\mathcal{R}}^p \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{K}\boldsymbol{u} + \boldsymbol{C}(\boldsymbol{u})\boldsymbol{u} + \boldsymbol{G}\boldsymbol{p} - \boldsymbol{f} + \boldsymbol{C}(\boldsymbol{u})\boldsymbol{v}_D \\ \boldsymbol{G}^T\boldsymbol{u} - \boldsymbol{h} \end{Bmatrix} = \boldsymbol{0}$$

Now in each Newton-Raphson iteration we have to solve the following system of equations to obtain the solution increment $\{\delta\boldsymbol{u}^T \ \delta\boldsymbol{p}^T\}^T$:

$$\begin{bmatrix} \frac{\partial \boldsymbol{\mathcal{R}}^u}{\partial \boldsymbol{u}} & \frac{\partial \boldsymbol{\mathcal{R}}^u}{\partial \boldsymbol{p}} \\ \frac{\partial \boldsymbol{\mathcal{R}}^p}{\partial \boldsymbol{u}} & \frac{\partial \boldsymbol{\mathcal{R}}^p}{\partial \boldsymbol{p}} \end{bmatrix} \begin{Bmatrix} \delta\boldsymbol{u} \\ \delta\boldsymbol{p} \end{Bmatrix} = \begin{Bmatrix} -\boldsymbol{\mathcal{R}}^u \\ -\boldsymbol{\mathcal{R}}^p \end{Bmatrix}$$

which after doing the differentiations reads

$$\begin{bmatrix} \frac{\partial \boldsymbol{\mathcal{R}}^u}{\partial \boldsymbol{u}} & \boldsymbol{G} \\ \boldsymbol{G}^T & \boldsymbol{0} \end{bmatrix} \begin{Bmatrix} \delta\boldsymbol{u} \\ \delta\boldsymbol{p} \end{Bmatrix} = \begin{Bmatrix} -\boldsymbol{\mathcal{R}}^u \\ -\boldsymbol{\mathcal{R}}^p \end{Bmatrix}$$

where according to the lecture notes, the matrix term $\frac{\partial \boldsymbol{\mathcal{R}}^u}{\partial \boldsymbol{u}}$ is computed as:

$$\frac{\partial \boldsymbol{\mathcal{R}}^u}{\partial \boldsymbol{u}} = \boldsymbol{K} + \boldsymbol{C}_1 + \boldsymbol{C}_2$$

where the matrix $\boldsymbol{C}_1$ arises from the discretization of the first convection term given by:

$$\boldsymbol{c}(\boldsymbol{w}, \boldsymbol{v}, \boldsymbol{v}^*) = \int \boldsymbol{w} \cdot (\boldsymbol{v}^* \cdot \boldsymbol{\nabla}\boldsymbol{v}) \ d\Omega$$

and the matrix $\boldsymbol{C}_2$ arises from the discretization of the second convection term given by:

$$\boldsymbol{c}(\boldsymbol{w}, \boldsymbol{v}^*, \boldsymbol{v}) = \int \boldsymbol{w} \cdot (\boldsymbol{\nabla}\boldsymbol{v}^* \cdot \boldsymbol{v}) \ d\Omega$$

The number of iterations of Newton-Raphson method required to achieve convergence is shown in Table 2 for the different values values of Re. It is noticed that more iterations are needed as the flow Reynolds number increases, implying the introduced difficulty at high Reynolds number. In fact, the method didn't converge for Reynolds number $Re = 1000, 2000$. However, the number of iterations required for Newton-Raphson method is less compared to Picard method in cases of $Re = 100, 500$, the reason is that Newton-Raphson method converges quadratically while Picard method converges linearly.

Re	n_{iters}
100	6
500	9
1000	Diverges
2000	Diverges

Table 2: Number of iterations of Newton-Raphson method until convergence for different Reynolds numbers

A Developed codes for Stokes flow

A.1 New function `ComputeErrorsFEM.m`

This function computes the required error norms for the velocity and pressure.

```
function [errL2_V, errH1_V, errL2_P] = ...
    ComputeErrorsFEM(X,T,XP,TP,velo,pres,RefElement)

errL2_V = 0; normL2_V = 0;
errH1_V = 0; normH1_V = 0;
errL2_P = 0; normL2_P = 0;

ngaus = RefElement.ngaus;
wgp = RefElement.GaussWeights;
N = RefElement.N;
Nxi = RefElement.Nxi;
Neta = RefElement.Neta;
NP = RefElement.NP;

for ielem = 1:size(T,1)
    Te = T(ielem,:);
    TPe = TP(ielem,:);
    Xe = X(Te,:);
    XPe = XP(TPe,:);
    ue = velo(Te,:);
    pe = pres(TPe);

    [errL2_e_V,errH1_e_V,normL2_e_V,normH1_e_V,errL2_e_P,normL2_e_P] = ...
       ComputeElementalErrors(Xe,XPe,ue,pe,N,Nxi,Neta,NP,ngaus,wgp);

    errL2_V = errL2_V + errL2_e_V;   normL2_V = normL2_V + normL2_e_V;
    errH1_V = errH1_V + errH1_e_V;   normH1_V = normH1_V + normH1_e_V;
    errL2_P = errL2_P + errL2_e_P;   normL2_P = normL2_P + normL2_e_P;
end

normL2_V = sqrt(normL2_V); errL2_V = sqrt(errL2_V)/normL2_V;
normH1_V = sqrt(normH1_V); errH1_V = sqrt(errH1_V)/normH1_V;
normL2_P = sqrt(normL2_P); errL2_P = sqrt(errL2_P)/normL2_P;

%-------------------------------------------------------------------------
function [errL2_e_V,errH1_e_V,normL2_e_V,normH1_e_V,errL2_e_P,normL2_e_P] ...
   = ComputeElementalErrors(Xe,XPe,u_e,p_e,N,Nxi,Neta,NP,ngaus,wgp)

xe = Xe(:,1); ye = Xe(:,2); nenV = length(xe);
xpe = XPe(:,1); ype = XPe(:,2); nenP = length(xpe);

errL2_e_V = 0; normL2_e_V = 0;
errH1_e_V = 0; normH1_e_V = 0;
errL2_e_P = 0; normL2_e_P = 0;

for ig = 1:ngaus
    N_ig    = N(ig,:);
    Nxi_ig  = Nxi(ig,:);
    Neta_ig = Neta(ig,:);
    NP_ig   = NP(ig,:);
```

```
    Jacob = [Nxi_ig(1:nenV); Neta_ig(1:nenV)]*Xe;
    J = det(Jacob);
    dvolu = wgp(ig)*J;

    res = Jacob\[Nxi_ig; Neta_ig];
    nx = res(1,:);
    ny = res(2,:);

    uh = N_ig*u_e(:,1); uh_x = nx*u_e(:,1); uh_y = ny*u_e(:,1);
    vh = N_ig*u_e(:,2); vh_x = nx*u_e(:,2); vh_y = ny*u_e(:,2);
    ph = NP_ig*p_e;

    pt_xy_V = N_ig(:,1:nenV)*[xe, ye];
    [u,v,u_x,u_y,v_x,v_y,¬] = ExactSol(pt_xy_V);

    normL2_e_V = normL2_e_V + (u^2 + v^2)*dvolu;
    normH1_e_V = normH1_e_V + (u_x^2 + v_y^2)*dvolu;
    errL2_e_V = errL2_e_V + ((u - uh)^2 + (v-vh)^2)*dvolu;
    errH1_e_V = errH1_e_V + ((u_x-uh_x)^2 + (v_y-vh_y)^2)*dvolu;

    pt_xy_P = NP_ig(:,1:nenP)*[xpe, ype];
    [¬,¬,¬,¬,¬,¬,p] = ExactSol(pt_xy_P);

    normL2_e_P = normL2_e_P + (p^2)*dvolu;
    errL2_e_P = errL2_e_P + ((p - ph)^2)*dvolu;
end
```

A.2 New function `stabilizedGLS.m`

This function computes the matrices arising from the GLS stabilization.

```
function [S1,S2] = stabilizedGLS(XP,TP,tau,referenceElement)
% [S1,S2] = stabilizedGLS(XP,TP,tau,referenceElement)
% Matrix S1 and r.h.s vector S2 obtained after discretizing a GLS ...
    stabilization
% term for Stokes problem
%
% XP,TP: nodal coordinates and connectivities for pressure
% tau: the stabilization parameter (viscous contribution)
% referenceElement: reference element properties (quadrature, shape ...
    functions...)

ngaus = referenceElement.ngaus;
wgp = referenceElement.GaussWeights;
NP = referenceElement.NP;
NPxi = referenceElement.NPxi;
NPeta = referenceElement.NPeta;

% Number of elements and number of nodes in each element
[nElem,nenP] = size(TP);

% Number of nodes
nPt_P = size(XP,1);
```

```
% Number of degrees of freedom
nedofP = nenP;
ndofP = nPt_P;

S1 = zeros(ndofP,ndofP);
S2 = zeros(ndofP,1);

% Loop on elements
for ielem = 1:nElem
    % Global number of the nodes in element ielem
    TPe = TP(ielem,:);
    % Coordinates of the nodes in element ielem
    XPe = XP(TPe,:);
    % Degrees of freedom in element ielem

    % Element matrices
    [S1e,S2e] = EleMatStokesGLS(XPe,nedofP,ngaus,wgp,NP,NPxi,NPeta,tau);

    % Assemble the element matrices
    S1(TPe, TPe) = S1(TPe, TPe) + S1e;
    S2(TPe) = S2(TPe) + S2e;
end




function [S1e,S2e] = EleMatStokesGLS(XPe,nedofP,ngaus,wgp,NP,NPxi,NPeta,tau)
% [S1e,S1e] = EleMatStokesGLS(XPe,nedofP,ngaus,wgp,NP,NPxi,NPeta,tau)

S1e = zeros(nedofP,nedofP);
S2e = zeros(nedofP,1);

% Loop on Gauss points
for ig = 1:ngaus
    NP_ig = NP(ig,:);
    NPxi_ig = NPxi(ig,:);
    NPeta_ig = NPeta(ig,:);
    Jacob = [
        NPxi_ig*(XPe(:,1))   NPxi_ig*(XPe(:,2))
        NPeta_ig*(XPe(:,1))  NPeta_ig*(XPe(:,2))
        ];
    dvolu = wgp(ig)*det(Jacob);
    res = Jacob\[NPxi_ig;NPeta_ig];


    S1e = S1e - tau*(res'*res)*dvolu;
    x_ig = NP_ig*XPe;
    f_igaus = SourceTerm(x_ig);
    S2e = S2e - tau*res'*f_igaus*dvolu;
end
```

A.3 Modified script `main.m`

The added parts to account for the GLS formulation are shown in lines (57-73), the function to compute the errors is added in line 105, this is then followed by some commands to produce the convergence plots.

```
% This program solves Stokes problem in a square domain

clear; close all; clc

addpath('Func_ReferenceElement')

dom = [0,1,0,1];
mu = 1;

formulation = 'GLS'; % 'Galerkin' or 'GLS'

% Element type and interpolation degree
% (0: quadrilaterals, 1: triangles, 11: triangles with bubble function)
elemV = 1; degreeV = 1; degreeP = 1;
% elemV = 0; degreeV = 2; degreeP = 0;
% elemV = 1; degreeV = 2; degreeP = 1;
% elemV = 11; degreeV = 1;  degreeP = 1;
if elemV == 11
    elemP = 1;
else
    elemP = elemV;
end
referenceElement = SetReferenceElementStokes(elemV,degreeV,elemP,degreeP);

% nx = cinput('Number of elements in each direction',10);

i = 1;
for nx = [10,20,40]
ny = nx;
hmin(i) = 1/nx;

[X,T,XP,TP] = CreateMeshes(dom,nx,ny,referenceElement);

figure; PlotMesh(T,X,elemV,'b-');
figure; PlotMesh(TP,XP,elemP,'r-');

% Matrices arising from the discretization
[K,G,f] = StokesSystem(X,T,XP,TP,referenceElement);
% [K,G,f] = StokesSystem2(X,T,XP,TP,referenceElement);
K = mu*K;
[ndofP,ndofV] = size(G);

% Matrix and r.h.s vector to impose Dirichlet boundary conditions using
% Lagrange multipliers
[A_DirBC, b_DirBC, nDir, confined] = BC(X,dom,ndofV);

% Total system of equations
if confined
   nunkP = ndofP-1;
   disp(' ')
```

```
   disp('Confined flow. Pressure on lower left corner is set to zero');
   G(1,:) = [];
else
   nunkP = ndofP;
end

if strcmp(formulation,'Galerkin')
    S1 = zeros(nunkP,nunkP);
    S2 = zeros(nunkP,1);
elseif strcmp(formulation,'GLS')
    tau = (1/3) * hmin(i)^2 / (4*mu);
    [S1,S2] = stabilizedGLS(XP,TP,tau,referenceElement);

    if confined
        S1 = S1(2:end,2:end);
        S2 = S2(2:end);
    end
end

Atot = [K          A_DirBC'             G'
        A_DirBC    zeros(nDir,nDir)     zeros(nDir,nunkP)
        G          zeros(nunkP,nDir)    S1];
btot = [f ; b_DirBC ; S2];

sol = Atot\btot;

% Postprocess
velo = reshape(sol(1:ndofV), 2, [])';
if confined
    pres = [0; sol(ndofV+nDir+1:ndofV+nDir+nunkP)];
else
    pres = sol(ndofV+nDir+1:ndofV+nDir+nunkP);
end

nPt = size(X,1);
figure;
quiver(X(1:nPt,1),X(1:nPt,2),velo(1:nPt,1),velo(1:nPt,2));
hold on
plot(dom([1,2,2,1,1]),dom([3,3,4,4,3]),'k')
axis equal; axis tight

PlotStreamlines(X,velo,dom);

PlotResults(X,T,velo(:,1),referenceElement.elemV,referenceElement.degreeV)

PlotResults(X,T,velo(:,2),referenceElement.elemV,referenceElement.degreeV)

if degreeP == 0
    PlotResults(X,T,pres,referenceElement.elemP,referenceElement.degreeP)
else
    PlotResults(XP,TP,pres,referenceElement.elemP,referenceElement.degreeP)
end

% Compute errors
[errL2_V, errH1_V, errL2_P] = ...
   ComputeErrorsFEM(X,T,XP,TP,velo,pres,referenceElement);
```

```
errL2_V_vector(i) = errL2_V;
errL2_P_vector(i) = errL2_P;
errH1_V_vector(i) = errH1_V;

i = i+1;
end


% mesh convergence plot
figure(100000000)
plot(log10(hmin),log10(errH1_V_vector),'b-s','DisplayName','Velocity ...
    H1-norm')
hold on
plot(log10(hmin),log10(errL2_P_vector),'r-s','DisplayName','Pressure ...
    L2-norm')
hold on
grid minor
xlabel('$log_{10}(h_{min})$','Interpreter','latex','FontSize',14,'FontName','cmr12')
ylabel('$log_{10}(||e||)$','Interpreter','latex','FontSize',14,'FontName','cmr12')
titleName = (['Q',num2str(degreeV),'Q',num2str(degreeP)]);
title(['Mesh convergence for ',titleName])
legend('show','Location','southeast')
% Compute slopes
n = length(hmin);
for i=1:n-1
    slopesH1_velo(i) = log10(errH1_V_vector(i+1)/errH1_V_vector(i)) / ...
        log10(hmin(i+1)/hmin(i));
    slopesL2_pres(i) = log10(errL2_P_vector(i+1)/errL2_P_vector(i)) / ...
        log10(hmin(i+1)/hmin(i));
end
% Write the slopes over the plot
x = log10(hmin(end))+0.2*abs(log10(hmin(end))-log10(hmin(end-1)));
y_velo = log10(errH1_V_vector(end))+0.5*abs(log10(errH1_V_vector(end))-...
    log10(errH1_V_vector(end-1)));
y_pres = log10(errL2_P_vector(end))+0.5*abs(log10(errL2_P_vector(end))-...
    log10(errL2_P_vector(end-1)));
text(x,y_velo,['Slope=',num2str(slopesH1_velo(end),'%2.2f')],'FontSize',10)
text(x,y_pres,['Slope=',num2str(slopesL2_pres(end),'%2.2f')],'FontSize',10)
```

B Developed codes for Cavity flow

B.1 Modified function `PlotStreamlines.m`

The modifications are done to draw more streamlines to show the features of the flow field at the corners.

```
function PlotStreamlines(X,velo,dom)

% Define a grid
xGrid = linspace(dom(1),dom(2),25);
yGrid = linspace(dom(3),dom(4),25);
% Interpolate the solution
tri = delaunay(X(:,1),X(:,2));
uGrid = tri2grid(X',tri',velo(:,1),xGrid,yGrid);    % x,y,f are column ...
    vectors.
vGrid = tri2grid(X',tri',velo(:,2),xGrid,yGrid);    % x,y,f are column ...
    vectors.
[xGrid,yGrid] = meshgrid(xGrid,yGrid);

% Points where the streamlines start
y1 = min(X(:,2));
aux = find(abs(X(:,2)-y1) < 1e-6);
xx_1 = X(aux(round(length(aux)/2)),1);
ind_1 = find(abs(X(:,1)-xx_1) < 1e-6);
xx_2 = X(aux(2),1);
ind_2 = find(abs(X(:,1)-xx_2) < 1e-6);
xx_3 = X(aux(end-1),1);
ind_3 = find(abs(X(:,1)-xx_3) < 1e-6);
sx_1 = [X(ind_1,1)];
sy_1 = [X(ind_1,2)];
sx_2 = [X(ind_2,1)];
sy_2 = [X(ind_2,2)];
sx_3 = [X(ind_3,1)];
sy_3 = [X(ind_3,2)];


figure;
streamline(xGrid,yGrid,uGrid,vGrid,sx_1,sy_1)
hold on
streamline(xGrid,yGrid,uGrid,vGrid,sx_2,sy_2)
hold on
streamline(xGrid,yGrid,uGrid,vGrid,sx_3,sy_3)
hold on
%plot(sx,sy,'r*')
plot(dom([1,2,2,1,1]),dom([3,3,4,4,3]),'k')
axis equal; axis tight
```

B.2 New function `ConvectionMatrix.m`

This function gives the convection matrix arising from the discretization of the convection term in the Navier-Stokes equations.

```
function C = ConvectionMatrix(X,T,referenceElement,velo)
% C = ConvectionMatrix(X,T,referenceElement,velo)
% Matrix C obtained after discretizing the convection term in NSE
%
% X,T: nodal coordinates and connectivities for velocity
% referenceElement: reference element properties (quadrature, shape ...
   functions...)
% velo: velocity field


elem = referenceElement.elemV;
ngaus = referenceElement.ngaus;
wgp = referenceElement.GaussWeights;
N = referenceElement.N;
Nxi = referenceElement.Nxi;
Neta = referenceElement.Neta;
ngeom = referenceElement.ngeom;

% Number of elements and number of nodes in each element
[nElem,nenV] = size(T);

% Number of nodes
nPt_V = size(X,1);
if elem == 11
    nPt_V = nPt_V + nElem;
end

% Number of degrees of freedom
nedofV = 2*nenV;
ndofV = 2*nPt_V;

C = zeros(ndofV,ndofV);

% Loop on elements
for ielem = 1:nElem
    % Global number of the nodes in element ielem
    Te = T(ielem,:);
    % Coordinates of the nodes in element ielem
    Xe = X(Te(1:ngeom),:);
    % Velocity at the nodes of the element
    Ve = velo(Te(1:ngeom),:);
    % Degrees of freedom in element ielem
    Te_dof = reshape([2*Te-1; 2*Te],1,nedofV);

    % Element matrices
    Ce = EleMat_Convection(Xe,ngeom,nedofV,ngaus,wgp,N,Nxi,Neta,Ve);

    % Assemble the element matrices
    C(Te_dof, Te_dof) = C(Te_dof, Te_dof) + Ce;
end
```

```
function Ce = EleMat_Convection(Xe,ngeom,nedofV,ngaus,wgp,N,Nxi,Neta,Ve)

Ce = zeros(nedofV,nedofV);

% Loop on Gauss points
for ig = 1:ngaus
    N_ig    = N(ig,:);
    Nxi_ig  = Nxi(ig,:);
    Neta_ig = Neta(ig,:);
    Jacob = [
        Nxi_ig(1:ngeom)*(Xe(:,1))   Nxi_ig(1:ngeom)*(Xe(:,2))
        Neta_ig(1:ngeom)*(Xe(:,1))  Neta_ig(1:ngeom)*(Xe(:,2))
        ];
    dvolu = wgp(ig)*det(Jacob);
    res = Jacob\[Nxi_ig;Neta_ig];
    nx = res(1,:);
    ny = res(2,:);

    Ngp = [reshape([1;0]*N_ig,1,nedofV); reshape([0;1]*N_ig,1,nedofV)];
    % Gradient
    Nx = [reshape([1;0]*nx,1,nedofV); reshape([0;1]*nx,1,nedofV)];
    Ny = [reshape([1;0]*ny,1,nedofV); reshape([0;1]*ny,1,nedofV)];

    v_ig = N_ig*Ve;

    Ce = Ce + Ngp'*(v_ig(1)*Nx+v_ig(2)*Ny)*dvolu;
end
```

B.3 New script `mainNavierStokes_NewtonRaphson.m`

This script solves Navier-Stokes equations using Newton-Raphson iterations.

```
% This program solves the Navier-Stokes cavity problem
clear; close all; clc

addpath('Func_ReferenceElement')

dom = [0,1,0,1];
Re = 100;
nu = 1/Re;

% Element type and interpolation degree
% (0: quadrilaterals, 1: triangles, 11: triangles with bubble function)
elemV = 0; degreeV = 2; degreeP = 1;
% elemV = 1; degreeV = 2; degreeP = 1;
% elemV = 11; degreeV = 1;  degreeP = 1;
if elemV == 11
    elemP = 1;
else
    elemP = elemV;
end
referenceElement = SetReferenceElementStokes(elemV,degreeV,elemP,degreeP);

nx = cinput('Number of elements in each direction',20);
```

```
meshType = 'adapted'; % 'uniform', 'adapted'
ny = nx;
[X,T,XP,TP] = CreateMeshes(dom,nx,ny,referenceElement,meshType);

figure; PlotMesh(T,X,elemV,'b-');
figure; PlotMesh(TP,XP,elemP,'r-');

% Matrices arising from the discretization
[K,G,f] = StokesSystem(X,T,XP,TP,referenceElement);
K = nu*K;
[ndofP,ndofV] = size(G);

% Prescribed velocity degrees of freedom
[dofDir,valDir,dofUnk,confined] = BC_red(X,dom,ndofV);
nunkV = length(dofUnk);
if confined
   nunkP = ndofP-1;
   disp(' ')
   disp('Confined flow. Pressure on lower left corner is set to zero');
   G(1,:) = [];
else
   nunkP = ndofP;
end

f = f - K(:,dofDir)*valDir;
Kred = K(dofUnk,dofUnk);
Gred = G(:,dofUnk);
fred = f(dofUnk);
A = [Kred   Gred'
     Gred   zeros(nunkP)];

% Initial guess
disp(' ')
IniVelo_file = input('.mat file with the initial velocity = ','s');
if isempty(IniVelo_file)
    velo = zeros(ndofV/2,2);
    y2 = dom(4);
    nodesY2 = find(abs(X(:,2)-y2) < 1e-6);
    velo(nodesY2,1) = 1;
else
    load(IniVelo_file);
end
pres = zeros(nunkP,1);
veloVect = reshape(velo',ndofV,1);
sol0  = [veloVect(dofUnk);pres(1:nunkP)];

iter = 0; tol = 0.5e-8;
while iter < 100
    fprintf('Iteration = %d\n',iter);

    C1 = ConvectionMatrix(X,T,referenceElement,velo);
    Cred1 = C1(dofUnk,dofUnk);

    Atot = A;
    Atot(1:nunkV,1:nunkV) = A(1:nunkV,1:nunkV) + Cred1;
    btot = [fred - C1(dofUnk,dofDir)*valDir; zeros(nunkP,1)];

    % Computation of residual
```

```
        res = btot - Atot*sol0;

        % linearized stiffness matrix
        C2 = ConvectionMatrix2(X,T,referenceElement,velo);
        Cred2 = C2(dofUnk,dofUnk);
        Atot_linearised = Atot;
        Atot_linearised(1:nunkV,1:nunkV) = Atot(1:nunkV,1:nunkV) + Cred2;

        % Computation of velocity and pressure increment
        solInc = Atot_linearised\res;

        % Update the solution
        veloInc = zeros(ndofV,1);
        veloInc(dofUnk) = solInc(1:nunkV);
        presInc = solInc(nunkV+1:end);
        velo = velo + reshape(veloInc,2,[])';
        pres = pres + presInc;

        % Check convergence
        delt1 = max(abs(veloInc));
        dela2 = max(abs(res));
        fprintf('Velocity increment=%8.6e, Residue max=%8.6e\n',delt1,dela2);
        if delt1 < tol*max(max(abs(velo))) && dela2 < tol
            fprintf('\nConvergence achieved in iteration number %g\n',iter);
            break
        end

        % Update variables for next iteration
        veloVect = reshape(velo',ndofV,1);
        sol0 = [veloVect(dofUnk); pres];
        iter = iter + 1;
    end

    if confined
        pres = [0; pres];
    end

    nPt = size(X,1);
    figure;
    quiver(X(1:nPt,1),X(1:nPt,2),velo(1:nPt,1),velo(1:nPt,2));
    hold on
    plot(dom([1,2,2,1,1]),dom([3,3,4,4,3]),'k')
    axis equal; axis tight

    PlotStreamlines(X,velo,dom);

    if degreeP == 0
        PlotResults(X,T,pres,referenceElement.elemP,referenceElement.degreeP)
    else
        PlotResults(XP,TP,pres,referenceElement.elemP,referenceElement.degreeP)
    end
```

B.4 New function `ConvectionMatrix2.m`

This function gives the convection matrix used in the linearisation of residual form to be used in Newton-Raphson iterations.

```
function C = ConvectionMatrix2(X,T,referenceElement,velo)
% C = ConvectionMatrix2(X,T,referenceElement,velo)
% Matrix C obtained after discretizing the convection term in NSE
%
% X,T: nodal coordinates and connectivities for velocity
% referenceElement: reference element properties (quadrature, shape ...
   functions...)
% velo: velocity field


elem = referenceElement.elemV;
ngaus = referenceElement.ngaus;
wgp = referenceElement.GaussWeights;
N = referenceElement.N;
Nxi = referenceElement.Nxi;
Neta = referenceElement.Neta;
ngeom = referenceElement.ngeom;

% Number of elements and number of nodes in each element
[nElem,nenV] = size(T);

% Number of nodes
nPt_V = size(X,1);
if elem == 11
    nPt_V = nPt_V + nElem;
end

% Number of degrees of freedom
nedofV = 2*nenV;
ndofV = 2*nPt_V;

C = zeros(ndofV,ndofV);

% Loop on elements
for ielem = 1:nElem
    % Global number of the nodes in element ielem
    Te = T(ielem,:);
    % Coordinates of the nodes in element ielem
    Xe = X(Te(1:ngeom),:);
    % Velocity at the nodes of the element
    Ve = velo(Te(1:ngeom),:);
    % Degrees of freedom in element ielem
    Te_dof = reshape([2*Te-1; 2*Te],1,nedofV);

    % Element matrices
    Ce = EleMat_Convection(Xe,ngeom,nedofV,ngaus,wgp,N,Nxi,Neta,Ve);

    % Assemble the element matrices
    C(Te_dof, Te_dof) = C(Te_dof, Te_dof) + Ce;
end
```

```
function Ce = EleMat_Convection(Xe,ngeom,nedofV,ngaus,wgp,N,Nxi,Neta,Ve)

Ce = zeros(nedofV,nedofV);

% Loop on Gauss points
for ig = 1:ngaus
    N_ig    = N(ig,:);
    Nxi_ig  = Nxi(ig,:);
    Neta_ig = Neta(ig,:);
    Jacob = [
        Nxi_ig(1:ngeom)*(Xe(:,1))   Nxi_ig(1:ngeom)*(Xe(:,2))
        Neta_ig(1:ngeom)*(Xe(:,1))  Neta_ig(1:ngeom)*(Xe(:,2))
        ];
    dvolu = wgp(ig)*det(Jacob);
    res = Jacob\[Nxi_ig;Neta_ig];
    nx = res(1,:);
    ny = res(2,:);

    Ngp = [reshape([1;0]*N_ig,1,nedofV); reshape([0;1]*N_ig,1,nedofV)];

    vx_ig = nx*Ve;
    vy_ig = ny*Ve;

    aux = [Ngp(1,:)*vx_ig(1) + Ngp(2,:)*vy_ig(1);
           Ngp(1,:)*vx_ig(2) + Ngp(2,:)*vy_ig(2)];

    Ce = Ce + Ngp'*aux*dvolu;
end
```

References

[1] Donea, J., Huerta, A. (2004). *Finite Element Methods for Flow Problems.* Chichester: Wiley, pp.287-288.

[2] Donea, J., Huerta, A. (2004). *Finite Element Methods for Flow Problems.* Chichester: Wiley, pp.314.