Màster en Mètodes Numèrics **Aitor Bazán Escoda**

Universitat Politècnica de Catalunya

Finite Elements in Fluids

Homework 5: Incompressible flow: Stokes and Navier-Stokes

Stokes problem.

Introduction. Description of the problem based on BC.

$$-\nu\nabla^2 v + \nabla p = b \; in \; \Omega$$

$$\nabla \cdot v = 0 \; in \; \Omega$$

N-S equations are nonlinear due to inertial term, as it will be shown in the next problem. However, for very low Reynolds number cases, this is when low speed flows and/or highly viscous fluids are used, the inertial term is negligibly small to the viscous term so it drops from the equation resulting on a set of linear equations called Stokes equations.

The flow is inside a squared domain [0, 1]^2. Left, right and bottom walls of such cavity are stationary, while the top wall is moving to the right with a prescribed velocity. Motion of the upper wall puts the fluid inside the cavity into motion and then, a large clockwise rotating vertex forms.

Reynolds number can also be governed. In the case of Stokes, it is not appropriate to mention about Reynolds number since Stokes equations correspond to the limiting case in which the Reynolds number is 0. The physical meaning corresponds to an idealization of the equations but it is valid for creeping flows whose advective inertial forces are relatively small compared to viscous forces.

The provided BC is the divergence free velocity distribution. Moreover, as it is shown in *Fig. 1* and in figures shown next, there exists a discontinuity in the boundary conditions at the two upper corners of the cavity. Also, Dirichlet boundary conditions are imposed on every boundary. Pressure is known at an arbitrary point of the domain, in the current case it is set in the lower left corner of the cavity where P=0 is prescribed.

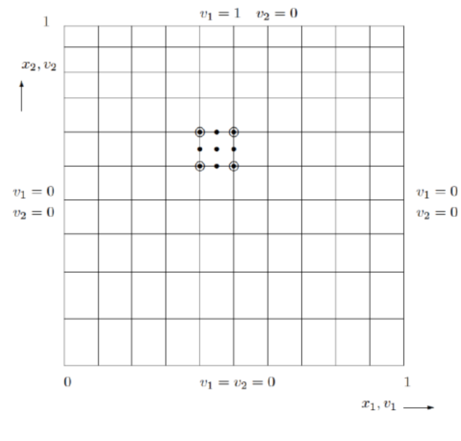


Figure 1. BC scheme.

First, the problem is solved for the Stokes problem with Galerkin formulation and these kinds of elements:

Q1Q1, Q2Q1;

P1P1, P2P1.

The stable ones are Q2Q1 and P2P1 as *Fig. 10 and Fig. 11* show. The key thing for this elements to show stability is the fact that they reach the compatibility condition known as ***Ladyzhenskaya-Babuska-Brezzi*** (LBB) or ***inf-sup*** condition so that unphysical node-to-node pressure oscillations can be avoided. According to this condition, pressure approximation must be at least one order lower than the velocity approximation over an element.

For the unstable ones, it is applied Galerkin Least-Squares method so as to overcome the problem of having zero entries on the main diagonal of the algebraic system of equations. The use of GLS not just enable non-oscillatory solution of high Reynolds number problems with reasonably fine meshes but also enable the use of equal order interpolation for velocity and pressure and notwithstanding they get rid of the zero diagonal entries of the global system of equations as aforementioned.

However, note that GLS is a residual based stabilization technique and if the original FEM solution is good enough, this is free from unphysical oscillations, then, the contribution of the method is negligible.

With 20 elements mesh it is observable in *Fig. 8* and *Fig. 9* the appearance of symmetry in the streamlines. Also, here below are shown some of the different types of mesh used for the study.

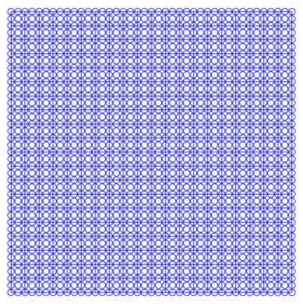


Figure 2. 20 elements Mesh. Quadrilaters, degreeV = 2.

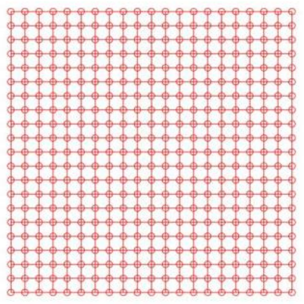


Figure 3. 20 elements Mesh. Quadrilaters, degreeP = 1.

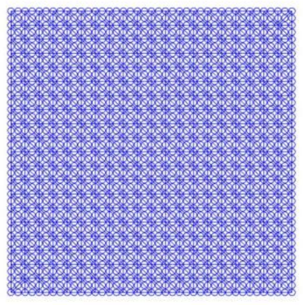


Figure 4. 20 elements Mesh. Triangles, degreeV = 2.

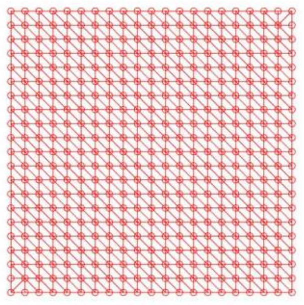


Figure 5. 20 elements Mesh. Triangles, degreeP = 1.

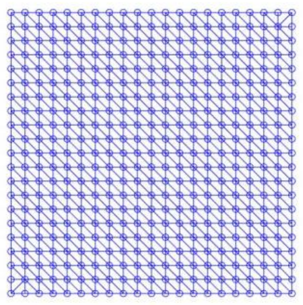


Figure 6. 20 elements Mesh. Bubble function, degreeV = 1

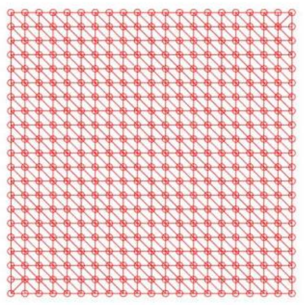


Figure 7. 20 elements Mesh. Bubble function, degreeP = 1.

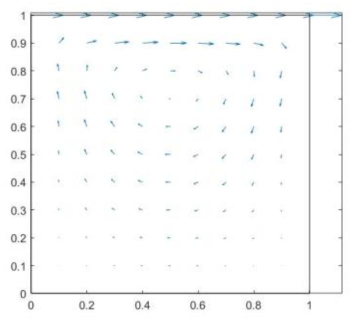


Figure 8. Velocity arrows.

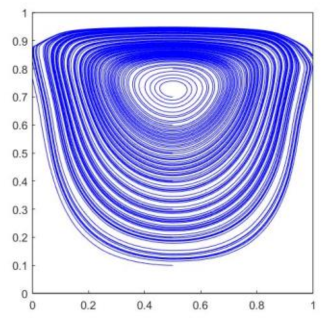


Figure 9. Velocity Streamlines.

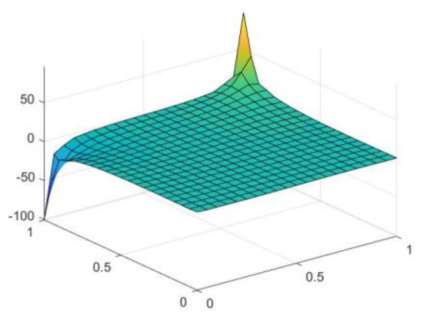


Figure 10. Stokes. elemV=0, degreeV=2, degreeP=1.

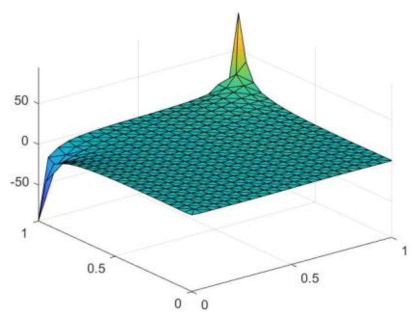


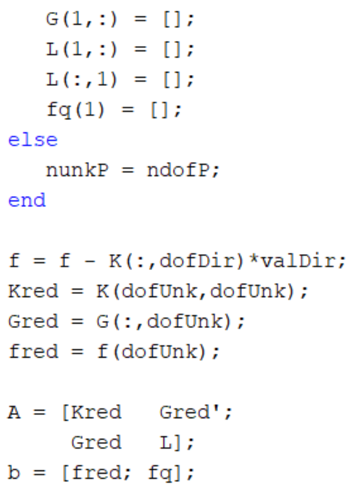
Figure 11. Stokes. elemV=1, degreeV=2, degreeP=1.

The images related to streamlines and the above ones determine the discontinuities on the upper corners of the domain.

<u>Methodology and Results</u>

GLS stabilization

The stabilization is added, in this case, for those cases where linear elements for pressure and velocity do not fulfil LBB compatibility condition. In such a case, the term $\bar{L}$ and $F_q$are added in the global system of equations.



```
   G(1,:) = [];
   L(1,:) = [];
   L(:,1) = [];
   fq(1) = [];
else
   nunkP = ndofP;
end

f = f - K(:,dofDir)*valDir;
Kred = K(dofUnk,dofUnk);
Gred = G(:,dofUnk);
fred = f(dofUnk);

A = [Kred   Gred';
     Gred   L];
b = [fred; fq];
```

As it is seen from the *APPENDIX A,* these $\bar{L}$ and F_q terms are multiplied by a stabilization term which corresponds to:

$$\tau = \frac{1}{3}\frac{{h_e}^2}{4\nu}$$

Where h_e is the element length, taken as the diameter of the circumcircle for triangular elements or largest length or diagonal for quadrilateral elements. The ν is the viscosity which can be avoided for Stokes formulation.

The equation for the GLS method is discretized as shown:

$$\int_{\Omega^e} \tau L(w) \cdot R(\vec{V}, p) d\Omega$$

$$\text{Where } R = -\frac{1}{\rho}\nabla P + \frac{\mu}{\rho}\nabla^2 \vec{V} + \vec{f}$$

$$\text{And } L(w) = -\frac{1}{\rho}\nabla w_c + \frac{\mu}{\rho}\nabla^2 \boldsymbol{w}$$

$$\text{Where } w\ is\ a\ combination\ of\ w_x\ and\ w_y$$

The GLS contribution to the continuity equation reads as follows:

$$\int_{\Omega^e} \tau(\nabla w_c) \cdot (-\mu\nabla^2 \vec{V} + \nabla p - \rho\vec{f}) d\Omega$$

If the velocity approximation over the element is linear, the GLS will lead to

$$\int_{\Omega^e} \tau(\nabla w_c) \cdot (\nabla p - \rho\vec{f}) d\Omega$$

Therefore, the modified weak form of the continuity equation is:

$$\int_{\Omega^e} \left[(-w_c)\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) - \tau\left(\frac{\partial w_c}{\partial x}\frac{\partial p}{\partial x} + \frac{\partial w_c}{\partial y}\frac{\partial p}{\partial y}\right)\right] d\Omega = \int_{\Omega^e} -\tau\, w_c\left(\frac{\partial f_x}{\partial x} + \frac{\partial f_y}{\partial y}\right) d\Omega$$

Minus term is added to the continuity equation on purpose in order to get a symmetric stiffness matrix at the end.

The extra terms added to the continuity eq. are:

$$K_{ij} = \int_{\Omega^e} -\tau \left(\frac{\partial \hat{S}_i}{\partial x} \frac{\partial \hat{S}_j}{\partial x} + \frac{\partial \hat{S}_i}{\partial y} \frac{\partial \hat{S}_j}{\partial y} \right) d\Omega$$

$$F_i = \int_{\Omega^e} -\tau \hat{S}_i \left(\frac{\partial f_x}{\partial x} + \frac{\partial f_y}{\partial y} \right) d\Omega$$

Where $w_c = \hat{S}_i$

Being $p^h(x, y) = \sum_{j=1}^{nenp} p_j \hat{S}_j(x, y)$

From here and in order to program de discretized formulation of the problem, in Appendix A it is remarked.

It is important to note that as the mesh is refined, the stabilization parameter tends to 0 and, therefore, the GLS stabilization effect disappears.

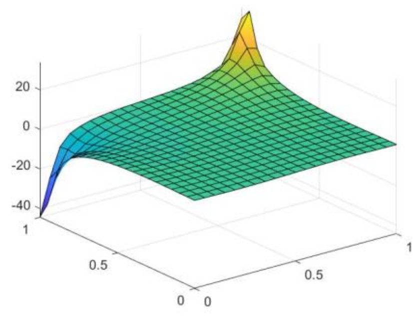


Figure 12. Stokes. elemV=0, degreeV=1, degreeP=1, 20elements.

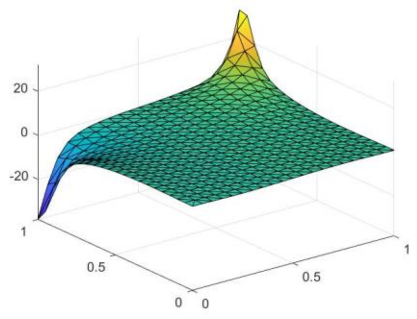


Figure 13. Stokes. elemV=1, degreeV=1, degreeP=1, 40elements.

Navier-Stokes problem.

Introduction. Description of the problem based on BC.

$$-\nu\nabla^2 v + (v \cdot \nabla)v + \nabla p = b \ in \ \Omega$$

$$\nabla \cdot v = 0 \ in \ \Omega$$

$$v = v_D \ on \ \Gamma_D$$

$$n \cdot \sigma = t \ on \ \Gamma_N$$

For Navier-Stokes formulation, two types of BC are given. Velocity (Dirichlet) and traction (Neumann), where n is the unit outward normal vector of the boundary, σ is the stress tensor, the sum of normal and shear stresses and t is the traction force applied by the boundary on the fluid. BC condition at a solid wall, for fluid flows, is known as no slip BC. This is, normal and tangential velocity components of the fluid are equivalent to those of the wall. For the cases where there are stationary walls, both of these components are 0. For this part of the exercise it is also imposed confined pressure to be 0 on the lower left corner of the cavity. Under the assumption that tractions are usually not known at an outflow boundary, here it is just specified a constant arbitrary pressure at a point approach.

Methodology and results

Navier-Stokes Picard's method resolution

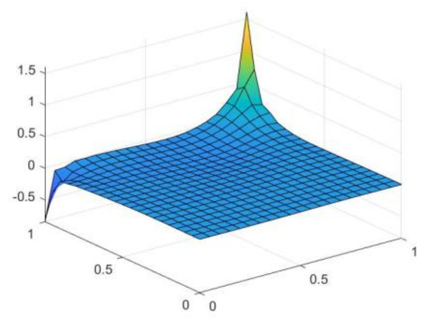


Figure 14. N-S Re=100. elemV=0, degreeV=2, degreeP=1, 20elements.

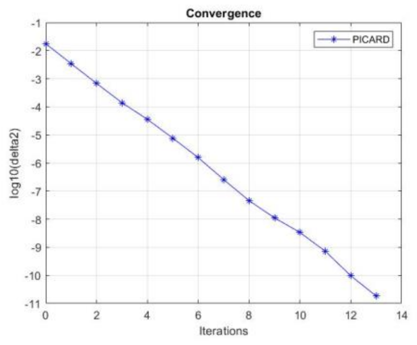


Figure 15. N-S Re=100. Convergence plot.

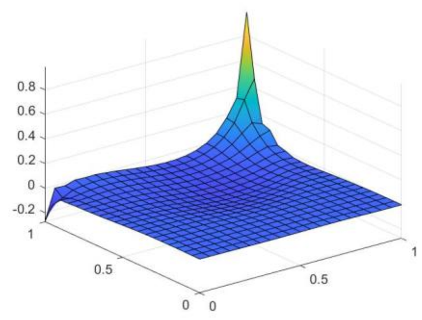


Figure 16. N-S Re=250. elemV=0, degreeV=2, degreeP=1, 20elements.

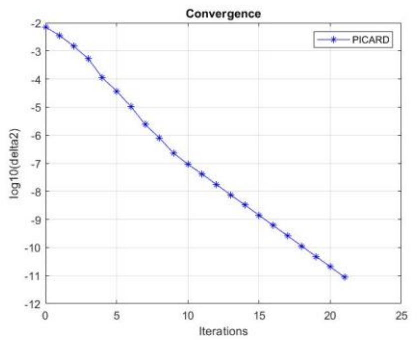


Figure 17. N-S Re=250. Convergence plot.

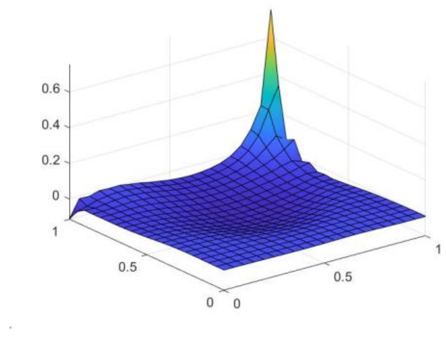


Figure 18. N-S Re=500. elemV=0, degreeV=2, degreeP=1, 20elements.

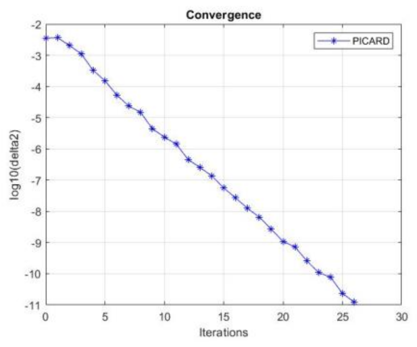


Figure 19. N-S Re=500. Convergence plot.

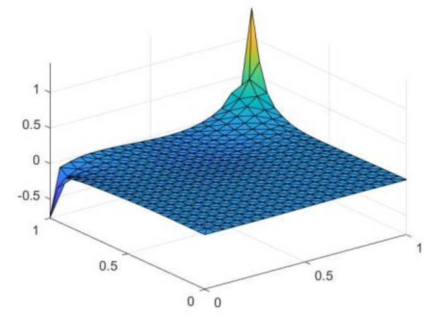


Figure 20. N-S Re=100. elemV=1, degreeV=2, degreeP=1, 20elements.

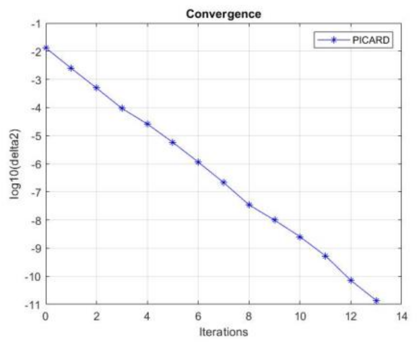


Figure 21. N-S Re=100. Convergence plot.

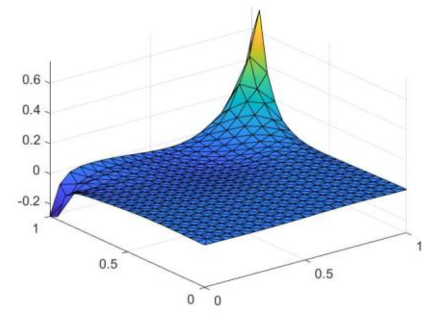


Figure 22. N-S Re=250. elemV=11, degreeV=2, degreeP=1, 20elements.

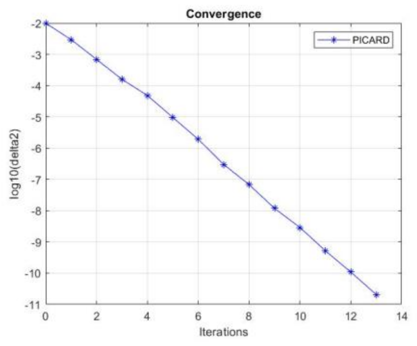


Figure 23. N-S Re=250. Convergence plot.

Here above it is shown how the Picard's method behaves linearly with the number of iterations.

The discretization of the convective matrix is as follows knowing that the final convective matrix is of the form:

$$C = \begin{bmatrix} v_x \dfrac{\partial v_x}{\partial x} + v_y \dfrac{\partial v_y}{\partial y} \\ v_x \dfrac{\partial v_x}{\partial y} + v_y \dfrac{\partial v_y}{\partial y} \end{bmatrix}$$

It is obtained the following for the convective term.

$$\xi(v) = \begin{bmatrix} V_x & 0 & V_y & 0 \\ 0 & V_x & 0 & V_y \end{bmatrix}$$

$$grad\ N = \ g(v) = \begin{bmatrix} \frac{\partial v_x}{\partial x} \\ \frac{\partial v_y}{\partial x} \\ \frac{\partial v_x}{\partial y} \\ \frac{\partial v_y}{\partial y} \end{bmatrix}$$

C1 = [mat N]^T $\xi(v)$[gradN]

Check Appendix B for further details on how it is programmed.

Navier-Stokes Newton-Raphson's resolution

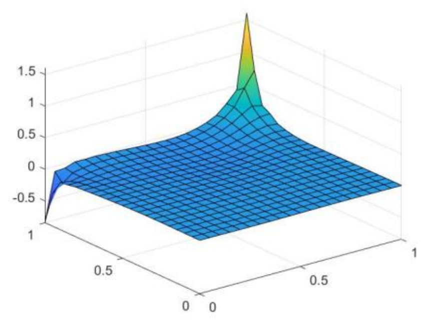


Figure 24. N-S Re=100. elemV=11, degreeV=2, degreeP=1, 20elements, Newton-Raphson.

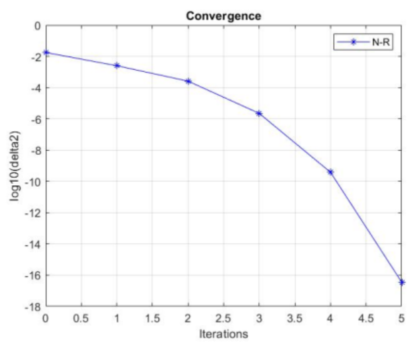


Figure 25. N-S Re=100. Convergence plot. Newton-Raphson.

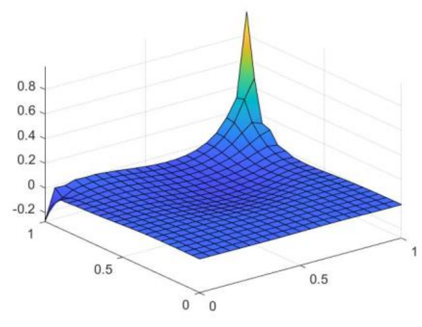


Figure 26. N-S Re=250. elemV=11, degreeV=2, degreeP=1, 20elements, Newton-Raphson.

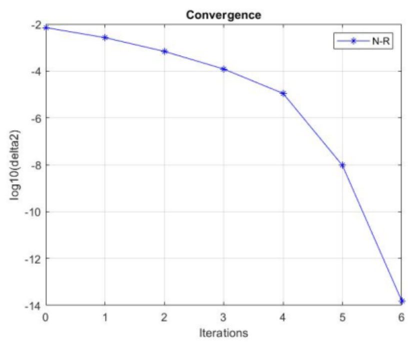


Figure 27. N-S Re=250. Convergence plot. Newton-Raphson.

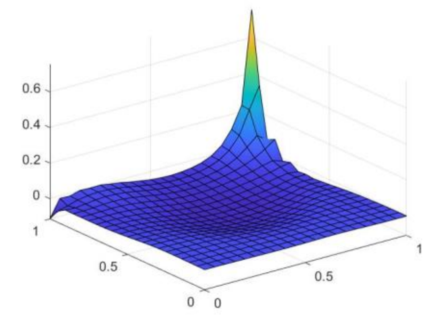


Figure 28. N-S Re=500. elemV=11, degreeV=2, degreeP=1, 20elements, Newton-Raphson.

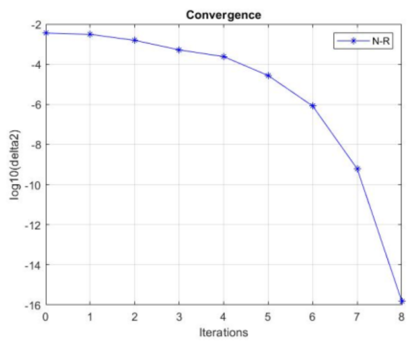


Figure 29. N-S Re=500. Convergence plot. Newton-Raphson.

Newton-Raphson behaves quadratically as expected.

An observation is that the time required per iteration increases as the mesh gets finer, as expected. But also note that for high Reynolds numbers coarse mesh solutions require more iterations to converge.

The method is discretized in such a way it is first computed the residual as it would usually do and then, the Jacobian is computed as:

$$\mathbf{r} = \begin{bmatrix} \underbrace{\big(\mathbf{K}+\mathbf{C}(\mathbf{v})\big)\mathbf{v}}_{\mathbf{r}_1(\mathbf{v})\ \text{non-linear}} + \mathbf{G}^T\mathbf{p} - \mathbf{f} \\ \mathbf{G}\mathbf{v} \end{bmatrix} \quad \mathbf{J} = \begin{pmatrix} \dfrac{\mathrm{d}\mathbf{r}_1}{\mathrm{d}\mathbf{v}} & \mathbf{G}^T \\ \mathbf{G} & \mathbf{0} \end{pmatrix}$$

For the derivative of the residual, it comes to the linearization of the convective term.

$$\frac{dr_1}{dv} = K + C_1(v) + C_2(v)$$

Where C1 is previously defined.

And C2 is:

$$\xi(v) = \begin{bmatrix} \dfrac{\partial v_x}{\partial x} & \dfrac{\partial v_y}{\partial y} \\ \dfrac{\partial v_x}{\partial y} & \dfrac{\partial v_y}{\partial y} \end{bmatrix}$$

$$v(x) = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

C2 = [mat N]^T $\xi(v)$[mat N]

Check for Appendix C for further details on how it is programmed.

<u>Conclusions</u>

Picard method reaches the solution in a higher number of iterations than Newton-Rapshon one. As commented before, as the Reynolds number is increased, the methods require more iterations to converge.

LBB-stable elements with quadratic velocity and linear pressure approximation provide non-oscillatory solutions with no stabilization.

If GLS is used, acceptable solutions can be obtained when using LBB-non stable elements.

```
function [K,G,f,L,fq] = StokesSystem(X,T,XP,TP,referenceElement)
% [K,G,f] = StokesSystem(X,T,XP,TP,referenceElement)
% Matrices K, G and r.h.s vector f obtained after discretizing a Stokes problem
%
% X,T: nodal coordinates and connectivities for velocity
% XP,TP: nodal coordinates and connectivities for pressure
% referenceElement: reference element properties (quadrature, shape functions...)


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

% Number of elements and number of nodes in each element
[nElem,nenV] = size(T);
nenP = size(TP,2);

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

% Number of degrees of freedom
nedofV = 2*nenV;
nedofP = nenP;
ndofV = 2*nPt_V;
ndofP = nPt_P;

K = zeros(ndofV,ndofV);
G = zeros(ndofP,ndofV);
f = zeros(ndofV,1);
L = zeros(ndofP,ndofP);
fq = zeros(ndofP,1);
% Loop on elements
for ielem = 1:nElem
    % Global number of the nodes in element ielem
    Te = T(ielem,:);
    TPe = TP(ielem,:);
    % Coordinates of the nodes in element ielem
    Xe = X(Te(1:ngeom),:);
    % Degrees of freedom in element ielem
    Te_dof = reshape([2*Te-1; 2*Te],1,nedofV);
    TPe_dof = TPe;

    % Element matrices
    [Ke,Ge,fe,Le,fqe] = EleMatStokes(Xe,ngeom,nedofV,nedofP,ngaus,wgp,N,Nxi,Neta,NP);

    % Assemble the element matrices
    K(Te_dof, Te_dof) = K(Te_dof, Te_dof) + Ke;
    G(TPe_dof,Te_dof) = G(TPe_dof,Te_dof) + Ge;
    f(Te_dof) = f(Te_dof) + fe;
    L(TPe_dof,TPe_dof) = L(TPe_dof,TPe_dof) + Le;
    fq(TPe_dof) = fq(TPe_dof) + fqe;
end

function [Ke,Ge,fe,Le,fqe] =
EleMatStokes(Xe,ngeom,nedofV,nedofP,ngaus,wgp,N,Nxi,Neta,NP)
%
global tau;
global degreeP;
global degreeV;
Ke = zeros(nedofV,nedofV);
Ge = zeros(nedofP,nedofV);
fe = zeros(nedofV,1);
Le = zeros(nedofP,nedofP);
fqe = zeros(nedofP,1);
% Loop on Gauss points
for ig = 1:ngaus
    N_ig    = N(ig,:);
    Nxi_ig  = Nxi(ig,:);
    Neta_ig = Neta(ig,:);
    NP_ig = NP(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)];
    % Divergence
    dN = reshape(res,1,nedofV);
    Ke = Ke + (Nx'*Nx+Ny'*Ny)*dvolu;
    Ge = Ge - NP_ig'*dN*dvolu;
    x_ig = N_ig(1:ngeom)*Xe;
    f_igaus = SourceTerm(x_ig);
    fe = fe + Ngp'*f_igaus*dvolu;
    if degreeV ~= degreeP
        Le = zeros(nedofP,nedofP);
        fqe = zeros(nedofP,1);
    else
    Le = Le - tau*([nx;nx;ny;ny]'*[nx;ny;nx;ny])*dvolu;
    fqe = fqe - tau*([nx;ny]'*f_igaus)*dvolu;
    end
end
```

```matlab
% This program solves the cavity flow problem
clear; close all; clc

addpath('Func_ReferenceElement')

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

% Element type and interpolation degree
% (0: quadrilaterals, 1: triangles, 11: triangles with bubble function)
global degreeV;
global degreeP;
% Inf-Sup compliant
% elemV = 0; degreeV = 2; degreeP = 1;
% elemV = 1; degreeV = 2; degreeP = 1;
% Inf-Sup non-compliant
% elemV = 0; degreeV = 1; degreeP = 1;
% elemV = 1; degreeV = 1; 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);
ny = nx;
[X,T,XP,TP] = CreateMeshes(dom,nx,ny,referenceElement);
global h;
h = XP(2)-XP(1);
global tau;
tau = 1/3*h^2/(4*mu);
figure; PlotMesh(T,X,elemV,'b-');
figure; PlotMesh(TP,XP,elemP,'r-');

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

% Prescribed velocity degrees of freedom
[dofDir,valDir,dofUnk,confined] = BC_red(X,dom,ndofV);
nunkV = length(dofUnk);

% Total system of equations
if confined

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

```
else
   nunkP = ndofP;
end

f = f - K(:,dofDir)*valDir;
Kred = K(dofUnk,dofUnk);
Gred = G(:,dofUnk);
fred = f(dofUnk);

fqred = fq(1:nunkP);

A = [Kred   Gred';
     Gred   L];
b = [fred; fq];

sol = A\b;

velo = zeros(ndofV,1);
velo(dofDir) = valDir;
velo(dofUnk) = sol(1:nunkV);
velo = reshape(velo,2,[])';
pres = sol(nunkV+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
```

APPENDIX B. PICARD CONVECTION MATRIX IMPLEMENTATION.

```
function C = ConvectionMatrix(X,T,referenceElement,velo)

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

% Total number of elements and element node's number
[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;

%Preallocation
C = zeros(ndofV,ndofV);

% Loop on elements
for ielem = 1:nElem
    % Te: current velocity element
    Te = T(ielem,:);
    Te_dof = reshape([2*Te-1; 2*Te],1,nedofV);

    % Xe: element nodes' coordinates
    Xe = X(Te(1:ngeom),:);

    % Ve: element nodes' velocity
    Ve = velo(T(ielem,:),:);

    % element matrix
    Cve = zeros(nedofV,nedofV);

    % Loop on Gauss points
    for ig = 1:ngaus
        % Shape functions on Gauss point igaus
        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)];
        % Velocity on point ig
        V_ig = N_ig*Ve;
        % Contribution to element matrix
        Cve = Cve + Ngp'*(V_ig(1)*Nx+V_ig(2)*Ny)*dvolu;
    end
    % Assembly
    C(Te_dof,Te_dof) =C(Te_dof,Te_dof) + Cve;
end
clear Cve;
C = sparse(C);
end
```

APPENDIX C. NEWTON-RAPHSON IMPLEMENTATION.

```
% 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;
global mu;
mu = 1;
% Element type and interpolation degree
% (0: quadrilaterals, 1: triangles, 11: triangles with bubble function)
global degreeV;
global degreeP;
elemV = 1; degreeV = 2; degreeP = 1;
% elemV = 0; degreeV = 1; 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);
ny = nx;
[X,T,XP,TP] = CreateMeshes(dom,nx,ny,referenceElement);
global h;
h = XP(2)-XP(1);
global tau;
tau = 1/3*h^2/(4*mu);
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);
```

```matlab
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)];
j = 1;
deltai = [];
iter = 0; tol = 0.5e-08;
while iter < 100

    fprintf('Iteration = %d\n',iter);

    [C,C2] = ConvectionMatrix(X,T,referenceElement,velo);
    Cred = C(dofUnk,dofUnk);
    Cred2 = C2(dofUnk,dofUnk);
    Atot = A;
    Atot(1:nunkV,1:nunkV) = A(1:nunkV,1:nunkV) + Cred;
    btot = [fred - C(dofUnk,dofDir)*valDir; zeros(nunkP,1)];

    % Computation of residual
    res = btot - Atot*sol0;

    % Computation of Jacobian
    dr1dv = Kred + Cred + Cred2;
    Jac = [dr1dv   Gred'
           Gred    zeros(nunkP)];
    % Computation of velocity and pressure increment
    solInc = Jac\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
    delta1 = max(abs(veloInc));
    delta2 = max(abs(res));
    deltai(j) = delta2;
    j = j+1;
    fprintf('Velocity increment=%8.6e, Residue max=%8.6e\n',delta1,delta2);
    if delta1 < tol*max(max(abs(velo))) && delta2 < 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;
    vect(j) = iter;

    cputime
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);

figure(2);
plot(vect,log10(deltai),'-*b');
title('Convergence');
ylabel('log10(delta2)');
xlabel('Iterations');
legend('N-R');
grid on
if degreeP == 0
    PlotResults(X,T,pres,referenceElement.elemP,referenceElement.degreeP)
else
    PlotResults(XP,TP,pres,referenceElement.elemP,referenceElement.degreeP)
end

function [C,C2] = ConvectionMatrix(X,T,referenceElement,velo)

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

% Total number of elements and element node's number
[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;

%Preallocation
C = zeros(ndofV,ndofV);
C2 = zeros(ndofV,ndofV);
% Loop on elements
for ielem = 1:nElem
    % Te: current velocity element
    Te = T(ielem,:);
    Te_dof = reshape([2*Te-1; 2*Te],1,nedofV);

    % Xe: element nodes' coordinates
    Xe = X(Te(1:ngeom),:);

    % Ve: element nodes' velocity
    Ve = velo(T(ielem,:),:);

    % element matrix
    Cve = zeros(nedofV,nedofV);
    Cve2 = zeros(nedofV,nedofV);
    % Loop on Gauss points
    for ig = 1:ngaus
        % Shape functions on Gauss point igaus
        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)];
        % Divergence
        dN = reshape(res,1,nedofV);
        % Velocity on point ig
        V_ig = N_ig*Ve;
        % Contribution to element matrix
        Cve = Cve + Ngp'*(V_ig(1)*Nx+V_ig(2)*Ny)*dvolu;
        Cve2 = Cve2 + Ngp'*([nx;ny]*Ve)'*Ngp*dvolu;
    end
    % Assembly
    C(Te_dof,Te_dof) =C(Te_dof,Te_dof) + Cve;
    C2(Te_dof,Te_dof) =C2(Te_dof,Te_dof) + Cve2;
end
clear Cve;
C = sparse(C);
clear Cve2;
C2 = sparse(C2);
end
```