Finite Elements in Fluid

Homework 5a: Navier-Stokes numerical examples

Ye Mao

Ye Mao, mao.ye@estudiant.upc.edu
Master of Numerical methods on engineering - Universitat Politècnica de Catalunya

1. INTRODUCTION

   Stokes equations

$$-\nu \nabla^2 v + (v \bullet \nabla) v + \nabla p = b \qquad in \Omega$$
$$\nabla \bullet v = 0 \qquad in \Omega$$
$$v = v_D \qquad on \Gamma_D$$
$$n \bullet \boldsymbol{\sigma} = t \qquad on \Gamma_D$$

   Weak form:

$$\begin{cases} \nu(\nabla w, \nabla v) + \int_\Omega w \bullet (v \bullet \nabla) v d\Omega - b(w,p) = (w,b) + (w,t)_{\Gamma_N} & \forall w \in \boldsymbol{v} \\ -(q, \nabla \bullet v) = 0 & \forall q \in \boldsymbol{Q} \end{cases}$$

   Non-linear system of equations

$$\begin{pmatrix} K + C_{(v)} & G \\ G^T & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} f \\ h \end{pmatrix}$$

2. OBJECTIVE

   2.1 Complete the Piccard's method with the convective term C(v).
   2.2 Solve the problem with a N-R scheme.
   2.3 Describing discretization of each method and the results obtained for both of them and comment them.

3. METHODOLOGY AND RESULTS

   In Navier-Stokes formulation, there are two types of boundary conditions, Dirichlet and Neumann. We can consider Velocity is Dirichlet BC and traction is Neumann BC, where n is the unit outward normal vector of boundary, $\boldsymbol{\sigma}$ is the stress tensor which is the sum of normal and shear stresses and t is traction force applied by the boundary on the fluid. BC in fluid flows is known as no slip BC compare with the one at solid wall. Normal and tangential velocity components of fluid are equivalent to those of the wall. While there are stationary walls, both of these components are 0. In this example, it is imposed confined pressure to be 0 on the lower left corner of cavity. The tractions are usually not known at an outflow boundary in this assumption. In this case, it is just specified a constant arbitrary pressure at one-point approach.

   3.1 Navier-Stokes Picard's method resolution

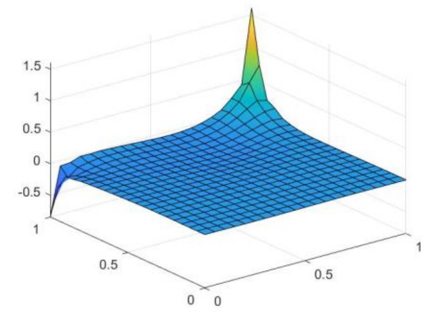


Figure 1. N-S Re=100 Q2Q1,20 element, Pressure

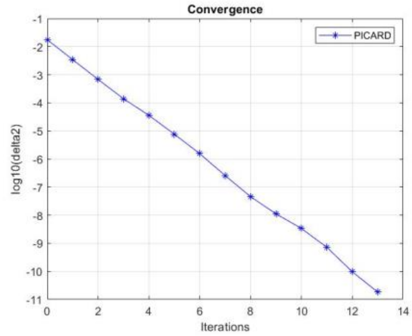


Figure 2. N-S Re=100 Q2Q1,20 element, convergence

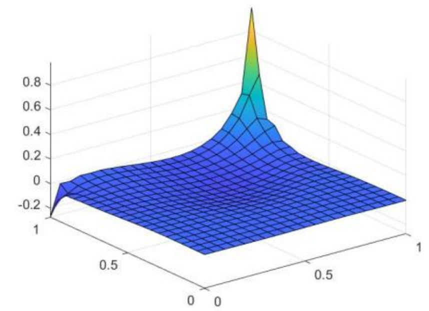


Figure 3. N-S Re=250 Q2Q1,20 element, Pressure

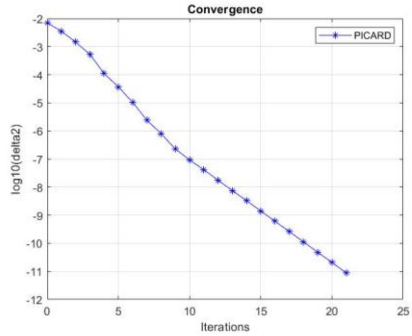


Figure 4. N-S Re=250 Q2Q1,20 element, convergence

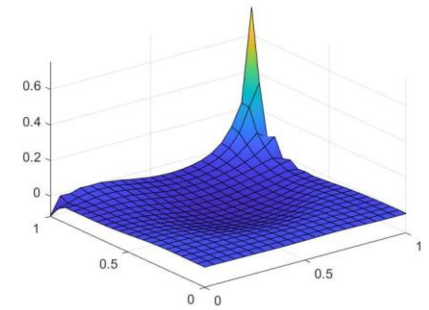


Figure 5. N-S Re=500 Q2Q1,20 element, Pressure

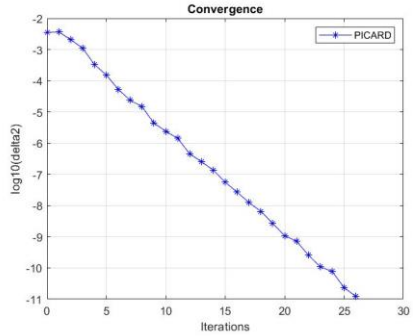


Figure 6. N-S Re=500 Q2Q1,20 element, convergence

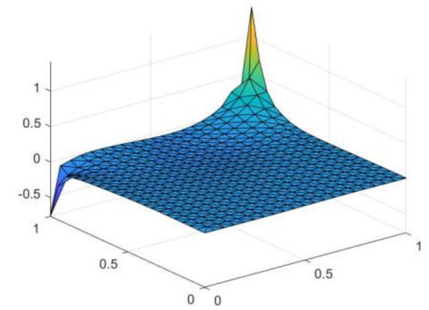


Figure 7. N-S Re=100 P2P1,20 element, Pressure

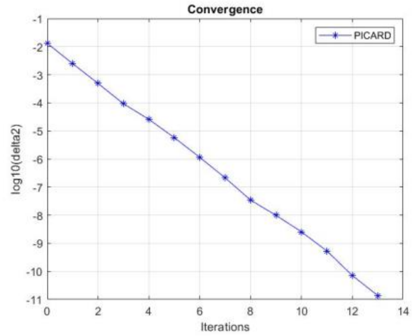


Figure 8. N-S Re=100 P2P1,20 element, convergence

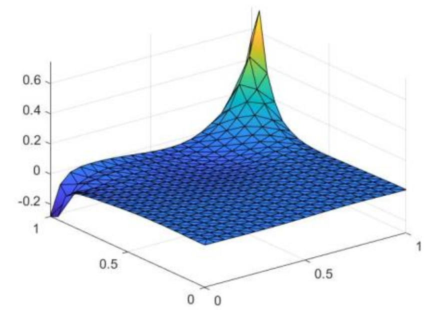


Figure 9. N-S Re=100 Mini,20 element, Pressure

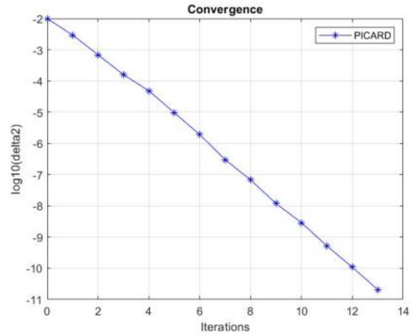


Figure 10. N-S Re=100 Mini,20 element, convergence

The discretization of the convective matrix is according the following. The final convective matrix form is:

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

Yield the following,

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

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

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

3.2 Navier-Stokes Newton-Raphson's method resolution

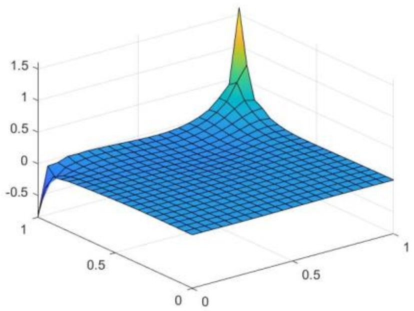


Figure 11. N-S Re=100 Q2Q1,20 element, Pressure

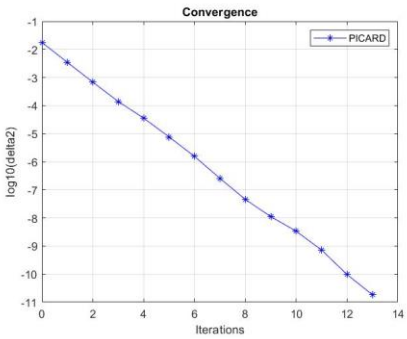


Figure 12. N-S Re=100 Q2Q1,20 element, convergence

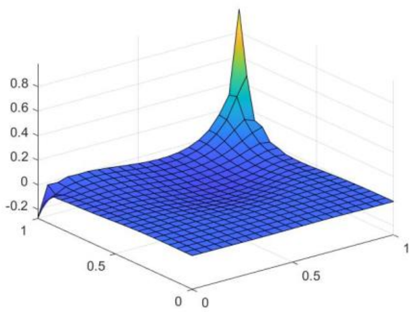


Figure 13. N-S Re=250 Q2Q1,20 element, Pressure

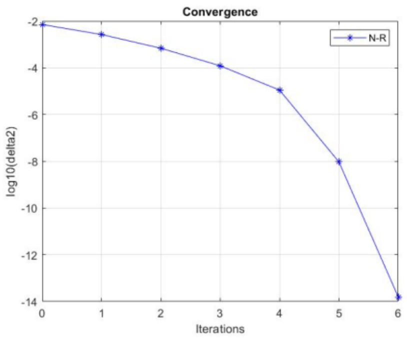
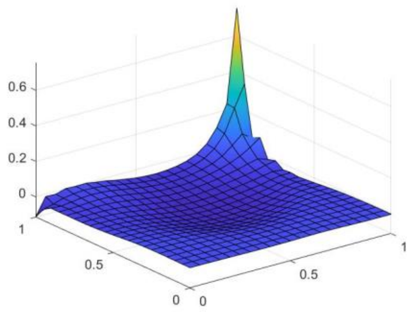


Figure 14. N-S Re=250 Q2Q1,20 element, convergence



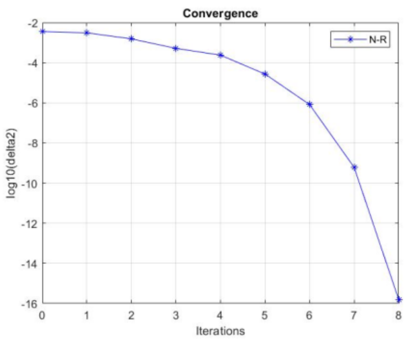


Figure 15. N-S Re=500 Q2Q1,20 element, Pressure

Figure 16. N-S Re=500 Q2Q1,20 element, convergence

Newton-Raphson works well with quadratically. The required time per iteration increases as the mesh become better. All of these are expected. It is also noted that the mesh solutions with high Reynold number require more iterations to converge.

In this method, the discretization can be understood as that we take computation of residual as usually, then obtain the Jacobian:

$$r = \begin{bmatrix} K + C(v)v + G^T p - f \\ Gv \end{bmatrix}$$

$$J = \begin{pmatrix} \frac{dr_1}{dv} & G^T \\ G & 0 \end{pmatrix}$$

Take the derivative of residual and obtain the linearization of the convective term.

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

Where C_1 is defined previously.
C2 is the following,

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

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

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

4. CONCLUSIONS

Compare with Newton-Rapshon method, Picard method need a higher number of iterations to get the solution. While the Reynold number is increased, both of methods require more iteration to converge. LBB-stable element with no stabilization that is quadratic velocity and linear pressure provides no-oscillation solutions. While GLS is used, LBB-non stable element can obtain acceptable solutions.

5. REFERENCE

[1] Lecture slides in Finite elements in fluid.

[2] Finite Element Methods for Flow Problems, Jean Donea and Antonio Huerta.

6. APPENDIX

6.1 Picard convection matrix codes

```
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
```

2.1 Newton-Raphson codes

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

addpath('Func_ReferenceElement')

dom = [0,1,0,1];
Re = 500;
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;
global elemV;
 elemV = 0; 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);
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.de
greeP)
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
```