```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Mar 30 22:22:35 2022

@author: juanmeriles
"""
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm

class element:
    NODE = []
    CON = []
    BOUN = []
    id_v = []

    def __init__(self):
        pass

def createBlock(origin,lenx,leny,numx,numy):
    xlocs = np.arange(origin[0]-lenx/2,origin[0]+lenx/2+.00000000000001,lenx/(numx-1))
    ylocs = np.arange(origin[1]-leny/2,origin[1]+leny/2+.00000000000001,leny/(numy-1))
    print(len(xlocs))
    print(len(ylocs))
    nodes = np.zeros((2,numx*numy))
    count = 0
    DOFcount = 0
    globalDOF = []
    for i in range(len(ylocs)):
        for j in range(len(xlocs)):
            nodes[0][count] = xlocs[j]
            nodes[1][count] = ylocs[i]
            globalDOF.append([DOFcount,DOFcount+1])
            count = count+1
            DOFcount = DOFcount+2
    con = []
    count = 0
    elid = []
    for i in range(len(ylocs)-1):
        for j in range(len(xlocs)-1):
            con.append([count,count+1,count+numx+1,count+numx])
```

```
elid.append(np.hstack([globalDOF[count],globalDOF[count+1],globalDOF[count+numx+1],globa
lDOF[count+numx]]))
            if (j == len(xlocs)-2):
                count = count+2
            else:
                count = count+1
            #print(count)


    return nodes,con,elid,xlocs,ylocs

def ShapeFcn(e,n):
    N1 = 1/4*(1-e)*(1-n)
    N2 = 1/4*(1+e)*(1-n)
    N3 = 1/4*(1+e)*(1+n)
    N4 = 1/4*(1-e)*(1+n)
    N = np.array([[N1,0,N2,0,N3,0,N4,0],
                  [0,N1,0,N2,0,N3,0,N4]])
    Nsmall = np.array([[N1,N2,N3,N4]])
    return N, Nsmall

def MakeB(e,n,el):
    dN1de = 1/4*(-1)*(1-n)
    dN2de = 1/4*(1)*(1-n)
    dN3de = 1/4*(1)*(1+n)
    dN4de = 1/4*(-1)*(1+n)
    dN1dn = 1/4*(1-e)*(-1)
    dN2dn = 1/4*(1+e)*(-1)
    dN3dn = 1/4*(1+e)*(1)
    dN4dn = 1/4*(1-e)*(1)
    J = Jacobian(e,n,el)
    temp = np.linalg.inv(J) @ np.array([[dN1de],[dN1dn]])
    dN1dx = temp[0][0]
    dN1dy = temp[1][0]
    temp = np.linalg.inv(J) @ np.array([[dN2de],[dN2dn]])
    dN2dx = temp[0][0]
    dN2dy = temp[1][0]
    temp = np.linalg.inv(J) @ np.array([[dN3de],[dN3dn]])
    dN3dx = temp[0][0]
    dN3dy = temp[1][0]
    temp = np.linalg.inv(J) @ np.array([[dN4de],[dN4dn]])
    dN4dx = temp[0][0]
    dN4dy = temp[1][0]
```

```
    B = np.array([[dN1dx,0,dN2dx,0,dN3dx,0,dN4dx,0],
              [0,dN1dy,0,dN2dy,0,dN3dy,0,dN4dy],
              [dN1dy,dN1dx,dN2dy,dN2dx,dN3dy,dN3dx,dN4dy,dN4dx]])
    return B

def Jacobian(e,n,el):
    dxde = 1/4*(-(1-n)*el.NODE[0][0]+(1-n)*el.NODE[1][0]+(1+n)*el.NODE[2][0]-
(1+n)*el.NODE[3][0])
    dyde = 1/4*(-(1-n)*el.NODE[0][1]+(1-n)*el.NODE[1][1]+(1+n)*el.NODE[2][1]-
(1+n)*el.NODE[3][1])
    dxdn = 1/4*(-(1-e)*el.NODE[0][0]-(1+e)*el.NODE[1][0]+(1+e)*el.NODE[2][0]+(1-
e)*el.NODE[3][0])
    dydn = 1/4*(-(1-e)*el.NODE[0][1]-(1+e)*el.NODE[1][1]+(1+e)*el.NODE[2][1]+(1-
e)*el.NODE[3][1])

    J = np.array([[dxde,dyde],
              [dxdn,dydn]])
    return J

def createKe(miu,el):
    gp = [-0.57735,0.57735]
    Ke = np.zeros((8,8))
    for i in gp:
        for j in gp:
            B = MakeB(i,j,el)
            miumat = np.array([[2*miu,0,0],
                        [0,2*miu,0],
                        [0,0,miu]])
            J = Jacobian(i,j,el)

            J = np.linalg.det(J)
            Ke = Ke+(B.T@miumat@B)*J

    return Ke

def createMe(rho,el):

    Me = np.zeros((8,8))
    for i in range(len(Me)):
        J = Jacobian(0,0,el)
        J = np.linalg.det(J)
        Me[i,i] = 4*1/4*rho*J
```

```
    return Me

def createAe(vec,el):
    gp = [-0.57735,0.57735]
    Ae = np.zeros((8,1))
    for i in gp:
        for j in gp:
            N,Nsmall = ShapeFcn(i,j)
            B = MakeB(i,j,el)
            dN1dx = B[0,0]
            dN2dx = B[0,2]
            dN3dx = B[0,4]
            dN4dx = B[0,6]
            dN1dy = B[1,1]
            dN2dy = B[1,3]
            dN3dy = B[1,5]
            dN4dy = B[1,7]

            dNdx = np.array([[dN1dx,0,dN2dx,0,dN3dx,0,dN4dx,0],
                             [0,dN1dx,0,dN2dx,0,dN3dx,0,dN4dx]])
            dNdy = np.array([[dN1dy,0,dN2dy,0,dN3dy,0,dN4dy,0],
                             [0,dN1dy,0,dN2dy,0,dN3dy,0,dN4dy]])

            gamma = np.array([dNdx@vec,dNdy@vec])
            gamma = np.hstack(gamma)
            J = Jacobian(i,j,el)

            J = np.linalg.det(J)

            Ae += N.T @ gamma @ N @ vec * J
    return Ae

def createA(ReDOFS,v,el):
    A = np.zeros((len(v)))
    for i in range(len(el)):
        vec = np.array([[v[ReDOFS[elements[i].id_v[0]]]],
                        [v[ReDOFS[elements[i].id_v[1]]]],
                        [v[ReDOFS[elements[i].id_v[2]]]],
                        [v[ReDOFS[elements[i].id_v[3]]]],
                        [v[ReDOFS[elements[i].id_v[4]]]],
                        [v[ReDOFS[elements[i].id_v[5]]]],
                        [v[ReDOFS[elements[i].id_v[6]]]],
                        [v[ReDOFS[elements[i].id_v[7]]]]])
        #print(vec)
```

```
        Ae = createAe(vec,el[i])
        for j in range(len(Ae)):
            ind1 = ReDOFS[elements[i].id_v[j]]
            A[ind1] += Ae[j]
    return A

def createCe(el):
    gp = [-0.57735,0.57735]
    Ce = np.zeros((1,8))
    for i in gp:
        for j in gp:
            B = MakeB(i,j,el)
            J = Jacobian(i,j,el)
            J = np.linalg.det(J)
            Ce += np.array([B[0][0],B[1][1],B[0][2],B[1][3],B[0][4],B[1][5],B[0][6],B[1][7]])*J
    return Ce

#CODE for running predictor corrector
miu = .01
rho = 1
numx = 41
numy = 41
nodes,con,elid,xlocs,ylocs = createBlock([.5,.5],1,1,numx,numy)
nodes = nodes.T
boun = []
pboun = []
v = np.zeros((2*len(nodes)))
vstar = np.zeros((2*len(nodes)))
p = np.zeros((2*len(nodes)))


for i in range(len(nodes)):
    if (nodes[i][0] == xlocs[0]):
        boun.append([1,1])
    elif (nodes[i][0] == xlocs[-1]):
        boun.append([1,1])
    elif (nodes[i][1] == ylocs[0]):
        boun.append([1,1])
    elif (nodes[i][1] == ylocs[-1]):
        boun.append([1,1])
    else:
        boun.append([0,0])

    if (nodes[i][1] == ylocs[-1]):
```

```
        v[2*i] = 1

for i in range(len(con)):
    pboun.append([0])

elements = []
#boun[20] = [1,0]
boun[4] = [1,0]
pboun[35] = [1]
p[35] = 0;
for i in range(len(con)):
    elements.insert(i,element())
    elements[i].NODE = 
np.array([nodes[con[i][0]],nodes[con[i][1]],nodes[con[i][2]],nodes[con[i][3]]])
    elements[i].CON = con[i]
    elements[i].BOUN = 
np.array([boun[con[i][0]],boun[con[i][1]],boun[con[i][2]],boun[con[i][3]]])
    elements[i].nodeVec = 
np.array([[elements[i].NODE[0][0],elements[i].NODE[0][1],elements[i].NODE[1][0],elements[i].
NODE[1][1],\

elements[i].NODE[2][0],elements[i].NODE[2][1],elements[i].NODE[3][0],elements[i].NODE[3][1]
]])
    elements[i].id_v = elid[i]


BDOFS = []
FDOFS = []
pBDOFS = []
pFDOFS = []
count = 0


for i in range(len(nodes)):
    if (boun[i][0] == 1):
        BDOFS.append(count)
        count = count+1
    else:
        FDOFS.append(count)
        count = count+1
    if (boun[i][1] == 1):
        BDOFS.append(count)
        count = count+1
    else:
```

```
        FDOFS.append(count)
        count = count+1

count = 0
for i in range(len(elements)):
    if (pboun[i][0] == 1):
        pBDOFS.append(count)
    else:
        pFDOFS.append(count)
    count = count+1

#Creates the ReDOF and RepDOF vectors which map the old global dofs to new positions
DOForder = np.hstack([FDOFS,BDOFS])
pDOForder = np.hstack([pFDOFS,pBDOFS])
ReDOFS = [0]*(2*len(nodes))
RepDOFS = [0]*(len(elements))
for i in range(len(DOForder)):
    ReDOFS[DOForder[i]] = i

for i in range(len(pDOForder)):
    RepDOFS[pDOForder[i]] = i


Ke = createKe(miu,elements[0])
Me = createMe(rho,elements[0])
Ce = createCe(elements[0])


#assemble global matrices
K = np.zeros((2*len(nodes),2*len(nodes)))
M = np.zeros((2*len(nodes),2*len(nodes)))
C = np.zeros((2*len(nodes),len(elements)))
for i in range(len(elements)):
    for j in range(8):
        C[ReDOFS[elements[i].id_v[j]],RepDOFS[i]] += Ce[0][j]
        for k in range(8):
            ind1 = ReDOFS[elements[i].id_v[j]]
            ind2 = ReDOFS[elements[i].id_v[k]]
            K[ind1,ind2] += Ke[j,k]
            M[ind1,ind2] += Me[j,k]


vfree = v[FDOFS]
vbound = v[BDOFS]
```

```
pfree = p[pFDOFS]
pbound = p[pBDOFS]


#Splitting up matrices into known and unknown parts
Kff = K[0:len(FDOFS),0:len(FDOFS)]
Mff = M[0:len(FDOFS),0:len(FDOFS)]
Mffinv = np.linalg.inv(Mff)
Cff_v = C[0:len(FDOFS),0:len(pFDOFS)].T
Cff_p = Cff_v.T
CTMC = Cff_p.T @ np.linalg.inv(Mff) @ Cff_p
CTMCinv = np.linalg.inv(CTMC)
#CTMCinv = CTMCinv[0:len(pFDOFS),0:len(pFDOFS)]

Kfb = K[0:len(FDOFS),len(FDOFS):]
Mfb = M[0:len(FDOFS),len(FDOFS):]
Cfb_p = C[0:len(FDOFS),len(pFDOFS):]
Cfb_v = C.T[0:len(pFDOFS),len(FDOFS):]


#Implement predictor corrector
dt = .001
t = np.arange(0,10,dt)

for i in tqdm(range(len(t))):
   A = createA(ReDOFS,v,elements)
   A2 = A[0:len(FDOFS)]
   #print(A)
   F = -Kfb @ vbound
   vfree = vfree-dt*Mffinv@(A2+Kff@vfree)+dt*Mffinv@F
   #vfree = vfree-dt*Mffinv@(Kff@vfree)+dt*Mffinv@F
   G = -Cfb_v @ vbound
   pfree =(1/dt)* CTMCinv @ (G-Cff_v@vfree)
   vfree = vfree+dt*Mffinv@(Cff_p@pfree)
   v = np.hstack([vfree,vbound])
   p = np.hstack([pfree,pbound])

v = v[ReDOFS]
p = p[RepDOFS]
vx = v[0::2]
vy = v[1::2]
vt = np.sqrt(vx**2+vy**2)

vx = vx.reshape(numx,numy)
```

```
vy = vy.reshape(numx,numy)
vt = vt.reshape(numx,numy)
X,Y = np.meshgrid(xlocs,ylocs)

plt.figure(50)
plt.contourf(X,Y, vt)
plt.colorbar()

plt.figure(51)
plt.contourf(X,Y, vx)
plt.colorbar()

plt.figure(52)
plt.contourf(X,Y, vy)
plt.colorbar()


pxlocs = xlocs-(xlocs[1]-xlocs[0])/2
pxlocs = pxlocs[1:]
pylocs = ylocs-(ylocs[1]-ylocs[0])/2
pylocs = pylocs[1:]
p = p.reshape(numx-1,numy-1)
pX,pY = np.meshgrid(pxlocs,pylocs)

plt.figure(90)
plt.contourf(pX,pY, p)
plt.colorbar()
```