



Department of Mechanical Engineering

MEC4045F

Assignment 3B

Date of submission: 26 June 2017

Plagiarism Declaration:

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. Each significant contribution to, and quotation in, this assignment from the work(s) of other people has been attributed, and has been cited and referenced.
3. This assignment is my own work.
4. I have not allowed, and will not allow anyone to copy my work with the intention of passing it off as his or her own work.


Student No.	Initials	First Name - Surname	Signature
MCKNEI009	NMM	Neil Mackenzie	

Table of Contents

Introduction	1
Question 1: Colour Plots	1
Question 2: Velocity Components along $x = 0.5$	5
Discussion.....	6
Conclusion.....	6
References	7
Appendix A: MeshGenerator.py	8
Appendix B: CFDSolver.py.....	10
Appendix C: Steps.py	13
Appendix D: Functions.py	15
Appendix E: Graphs.py	19

Introduction

The aim of this assignment was to generate a code to model the case of a lid-driven cavity. This was performed using python code on equi-spaced meshes of 21 by 21 and 51 by 51 nodes. The fluid was assumed to have a density $\rho = 1$ and viscosity $\mu = 0.01$, and the flow domain was assumed to be a unit square. This resulted in a Reynold's number of $Re = 100$.

The velocity was solved using the pressure projected method outlined in the course notes for MEC4045F. A fractional three-step semi-implicit method was used to solve for the velocity. First order upwinding was used for the convective term in the first step, and the pressure in the second step was solved implicitly using a coefficient matrix. This allowed for the velocity at the next time step to be solved explicitly in step three.

The discretisations for each of the terms was performed based on the method outlined in the course notes as well as that which was submitted in the author's Assignment 3A.

Question 1: Colour Plots

The code was iterated until the residual for pressure and velocity components fell below 1×10^{-5} . This required 981 iterations for the 21 by 21 node mesh and 2961 iterations for the 51 by 51 node mesh. Figure 1 and Figure 2 indicate the results of the code for a 21 by 21 node mesh and a 51 by 51 node mesh respectively. These plots were generated by Plotter.py, written by Bevan Jones.

The colour plots for the velocity magnitude for each mesh are repeated in Figure 3 and Figure 4 for clarity.

MEC4045F Assignment 3B
Plots for a 21 by 21 Node Mesh

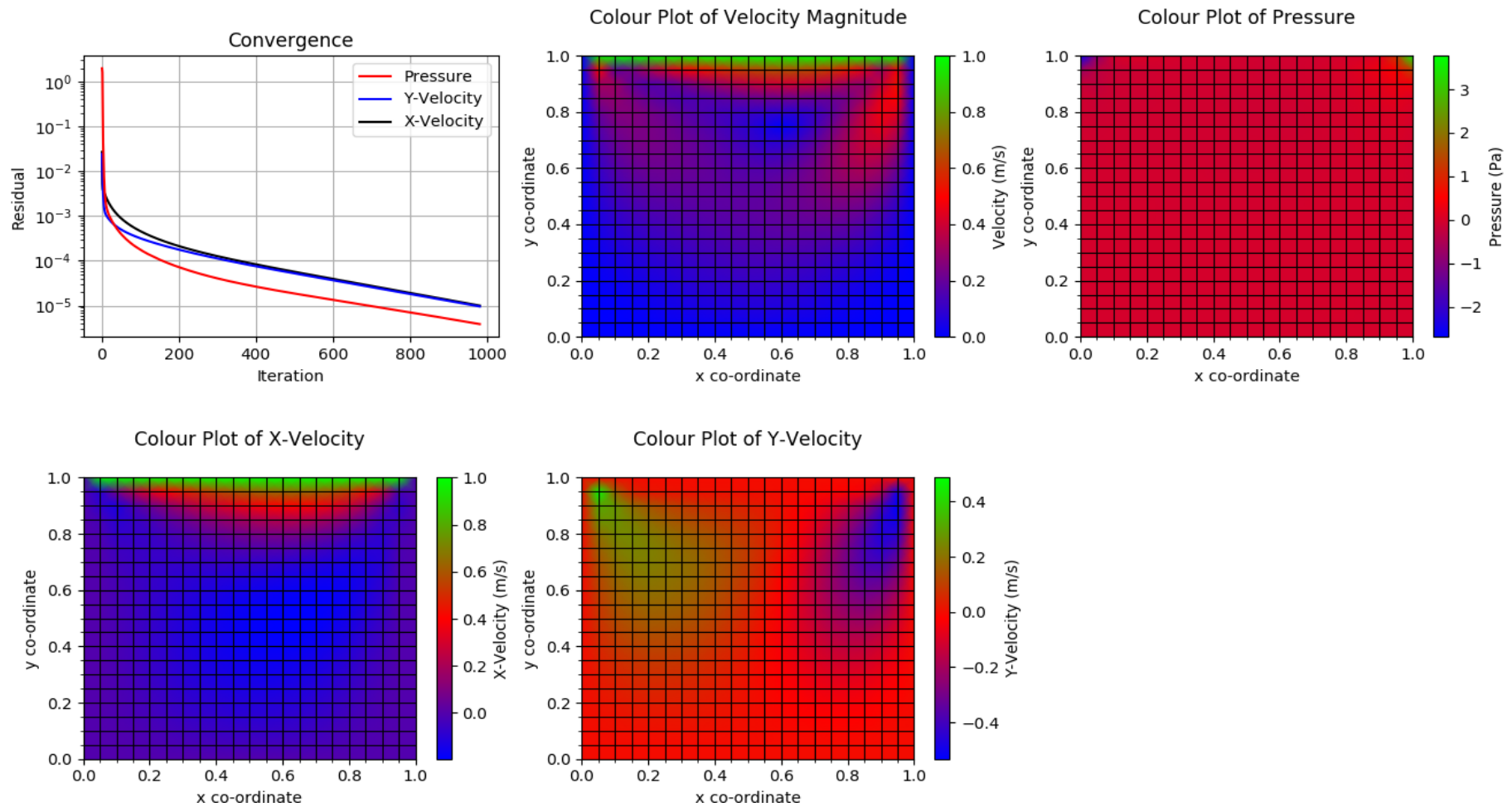


Figure 1: Results for 21 by 21 Node Mesh

MEC4045F Assignment 3B
Plots for a 51 by 51 Node Mesh

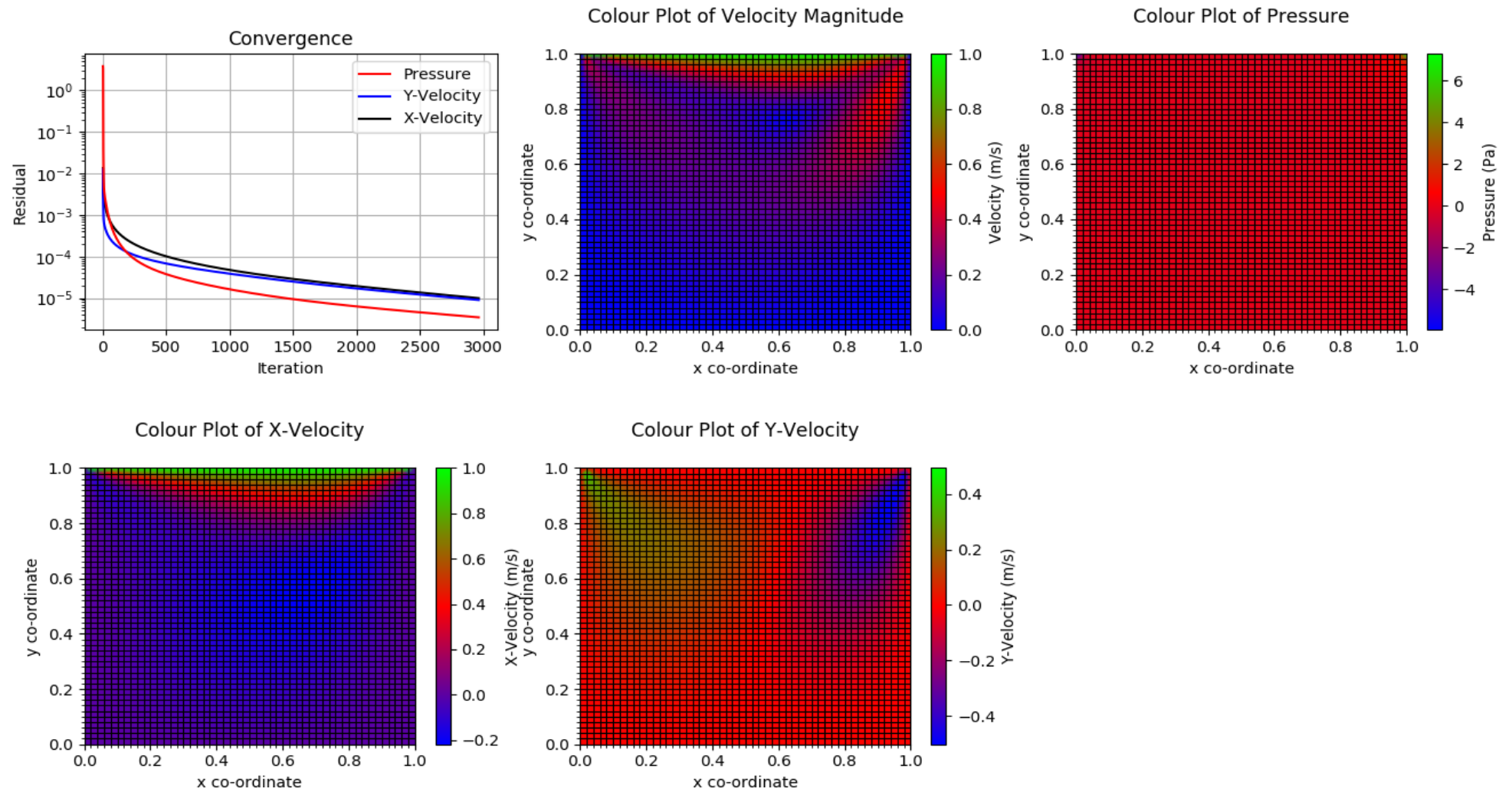


Figure 2: Results for 51 by 51 Node Mesh

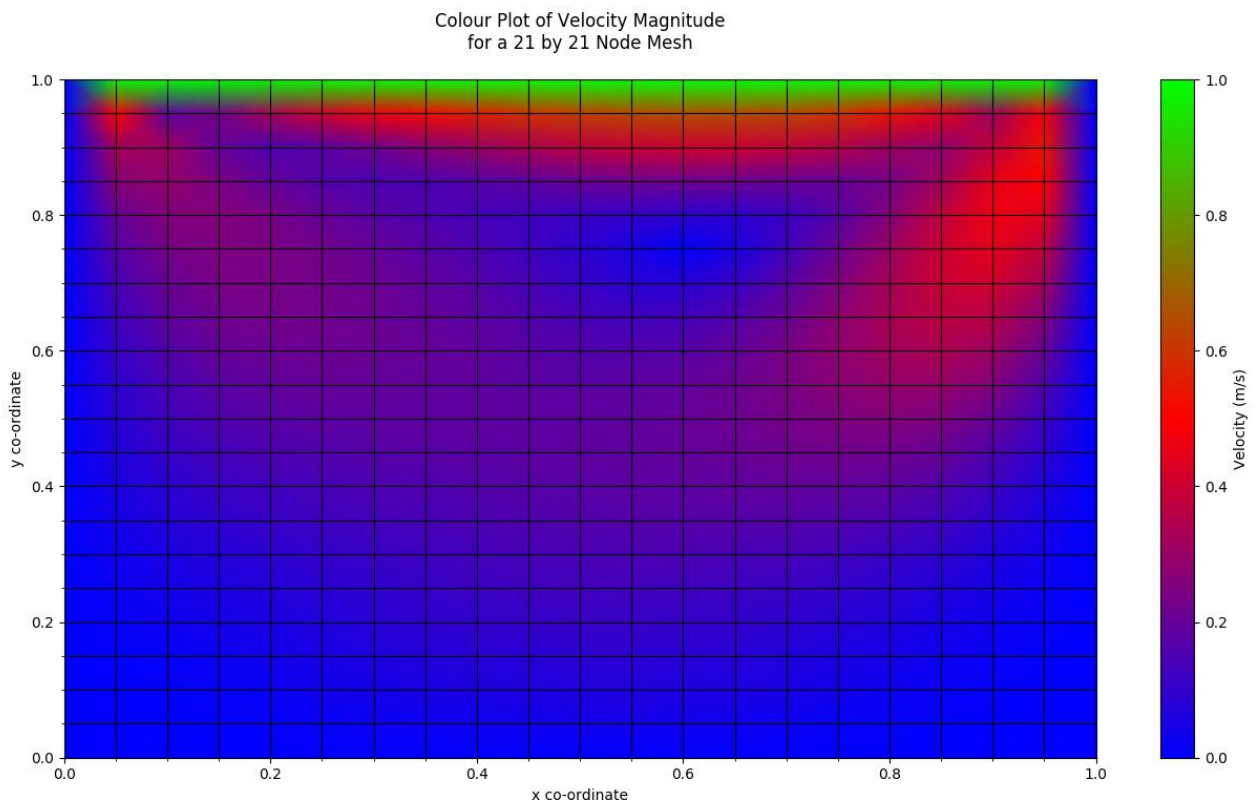


Figure 3: Colour Plot for 21 by 21 Node Mesh

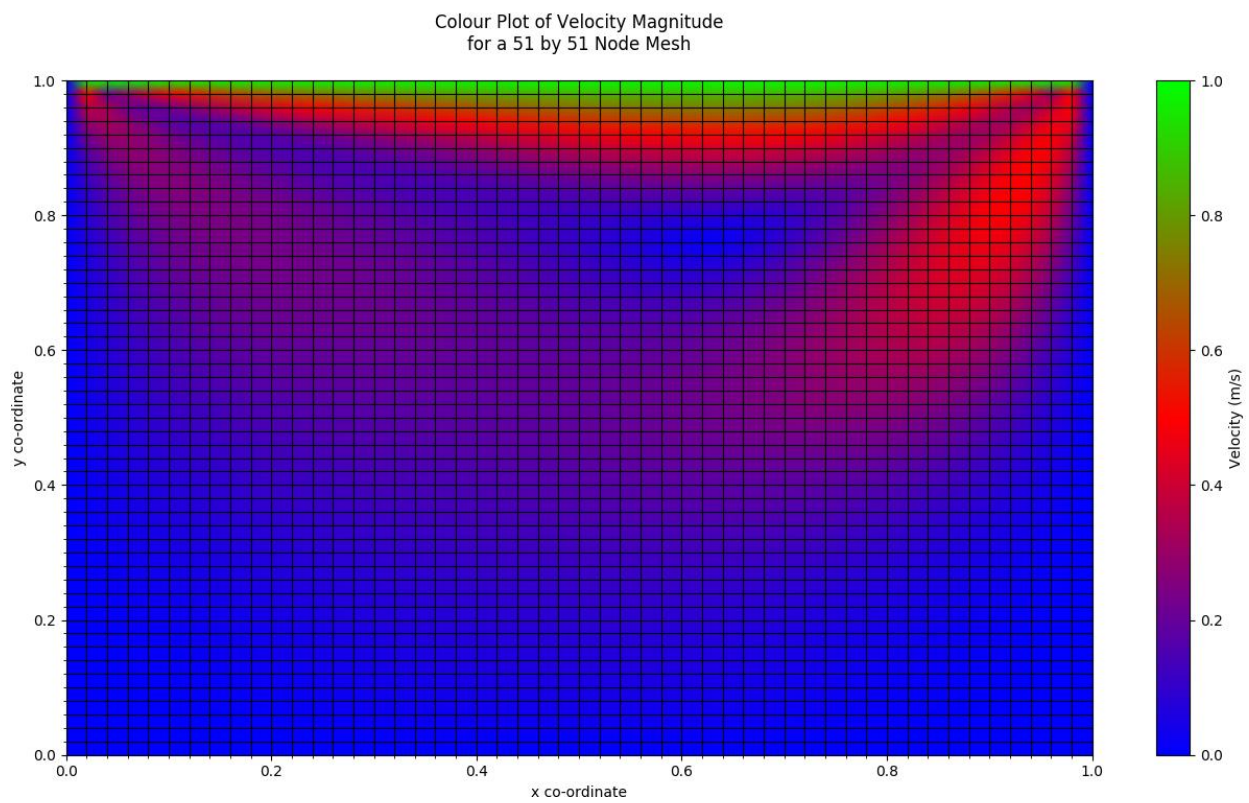


Figure 4: Colour Plot for 51 by 51 Node Mesh

Question 2: Velocity Components along $x = 0.5$

The x and y velocity components plotted along the line $x = 0.5$ are shown in Figure 5 and Figure 6 respectively.

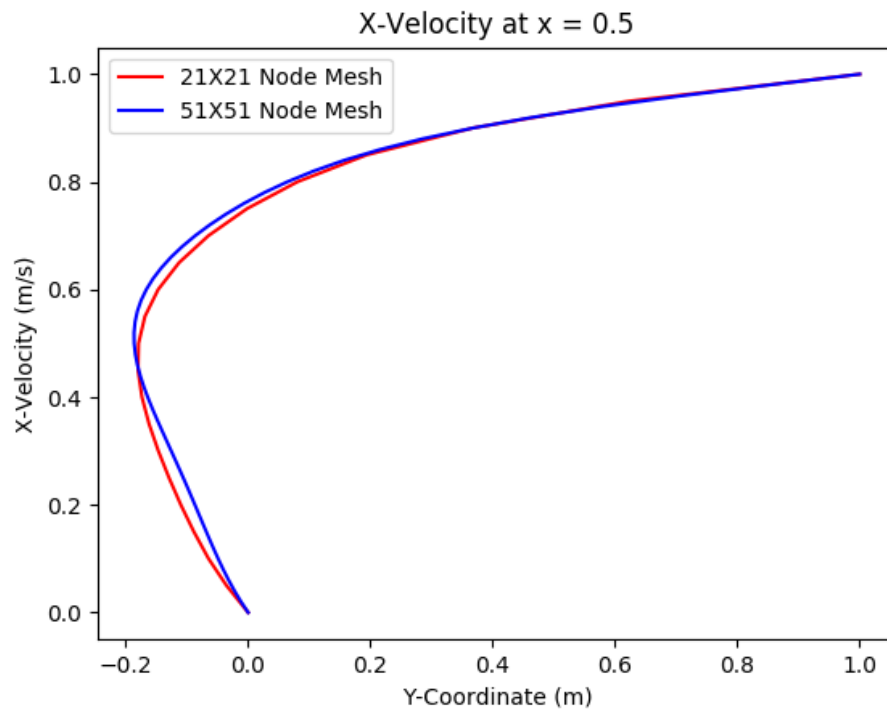


Figure 5: X-Velocity Component along $x=0.5$

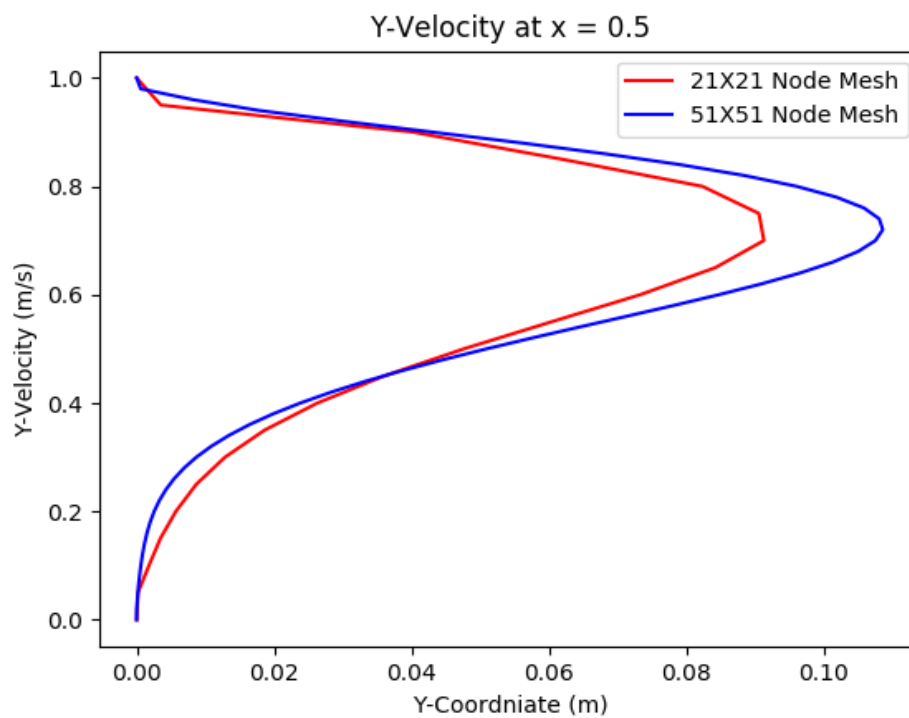


Figure 6: Y-Velocity Component along $x=0.5$

Discussion

The results of the x-component of velocity along the line $x = 0.5$ are similar to results reported by Omari [1] and Marchi [2]. The results of the y-component could not be compared as Omari reports these along the line $y=0.5$.

The overall velocity magnitude determined in this assignment is similar to the horizontal velocity plot provided by the benchmark for a Reynold's number of 100 [3]. Since the x-velocity is dominant in the top of the cavity, the plot can be considered to match the overall benchmark closely.

Since the code successfully matches the results of values reported in literature, it can be considered to successfully solve the lid-driven cavity problem.

Whilst the code may run successfully, it is not as efficient as it could be. This is especially true in "Functions.py", where each type of node has to be handled individually. The code could be improved by assigning attributes to each node within a class and using a node connectivity table to solve each step faster.

Conclusion

The python code generated for this assignment successfully solved the 2D lid-driven cavity problem using the pressure projection method for a fluid with a Reynold's Number of 100 in a unit domain. This was confirmed by comparing the results obtained to values reported in literature. The discretisation of each step, submitted in Assignment 3A, was therefore correct. Each step of the pressure projection method was also correctly performed.

Although successful, the code can be condensed with better techniques that will allow the processing time to be reduced.

References

- [1] R. Omari, "CFD Simulations of Lid Driven Cavity Flow at Moderate Reynolds Number," *European Scientific Journal*, vol. 9, no. 15, 2013.
- [2] C. H. Machi, R. Suero, and L. K. Araki, "The Lid-Driven Square Cavity Flow: Numerical Solution with a 1024 x 1024 Grid," *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, vol. 31, no. 3, 2009.
- [3] N. Technologies. *Benchmark: Lid-driven Cavity (2d)*. Available: <http://www.zetacomp.com/benchmarks/lid-driven-cavity-2d.asp>. [Accessed on 26 June, 2017]

Appendix A: MeshGenerator.py

```
1. #MeshGenerator.py
2. #Description: Generate 2D Mesh with identifiers for each node
3. #Author: Neil Mackenzie
4.
5. import numpy as np
6. def NodeTable(n,m):
7.
8.     #Define number of edges and edge length in each direction
9.     EdgesX=n-1
10.    EdgesY=m-1
11.    DeltaX=1/EdgesX
12.    DeltaY=1/EdgesY
13.    #NoNodes = number nodes in x X number nodes in y
14.    NoNodes=n*m
15.
16.    #Determine x and y co-ordinates along each edge
17.    XCoord=[0]
18.    YCoord=[0]
19.    for i in range (EdgesX):
20.        XCoord.append(DeltaX*(i+1))
21.    for j in range(EdgesY):
22.        YCoord.append(DeltaY*(j+1))
23.
24.    #Generate co-ordinate list with identifier, face normal, control volume
25.    #and edge length for each node
26.    #Identifier is used to tell whether a node is a corner, edge or internal node
27.    #If it is a corner/edge node, Identifier shows which one
28.    XYCoord = np.zeros((NoNodes,2))
29.    Identifier=np.zeros(NoNodes)
30.    C=np.zeros((NoNodes,4))
31.    ControlVol=np.zeros(NoNodes)
32.    NodeEdgeLength=np.zeros((NoNodes,2))
33.    #Loop variable to identify location in single-row node list
34.    Loop=0
35.    #Loop through m number of nodes in x-direction to assign values to nodes
36.    #Then step up 1 in y-direction and repeat until y=n
37.    for k in range (n):
38.        for l in range (m):
39.
40.            XYCoord[Loop]=XCoord[l],YCoord[k]
41.            if k==0 and l ==0:                                #Bottom Left Corner
42.                Identifier[Loop] = 1
43.                ControlVol[Loop] = DeltaX*DeltaY/4
44.                C[Loop]=DeltaX/2,DeltaY/2,-DeltaX/2,-DeltaY/2
45.            if k==0 and l!=0 and l!= EdgesX:                #Bottom Edge excl. corn
46.                Identifier[Loop] = 2
47.                ControlVol[Loop] = DeltaX*DeltaY/2
48.                C[Loop]=DeltaX,DeltaY/2,-DeltaX,-DeltaY/2
49.            if k==0 and l==EdgesX:                            #Bottom Right Corner
50.                Identifier[Loop] = 3
51.                ControlVol[Loop] = DeltaX*DeltaY/4
52.                C[Loop]=DeltaX/2,DeltaY/2,-DeltaX/2,-DeltaY/2
53.            if k!=0 and l==0:                                #Left Boundary excl. co
54.                Identifier[Loop] = 4
55.                ControlVol[Loop] = DeltaX*DeltaY/2
56.                C[Loop]=DeltaX/2,DeltaY,-DeltaX/2,-DeltaY
57.            if k!=0 and k!= EdgesY and l!= 0 and l!= EdgesX:#Internal Nodes
58.                Identifier[Loop] = 5
59.                ControlVol[Loop] = DeltaX*DeltaY
60.                C[Loop]=DeltaX,DeltaY,-DeltaX,-DeltaY
```

```

61.         if k!= 0 and k!= EdgesY and l==EdgesX:           #Right Boundary excl. c
        orners
62.             Identifier[Loop] = 6
63.             ControlVol[Loop] = DeltaX*DeltaY/2
64.             C[Loop]=DeltaX/2,DeltaY,-DeltaX/2,-DeltaY
65.             if k==EdgesY and l==0:                         #Top left corner
66.                 Identifier[Loop] = 7
67.                 ControlVol[Loop] = DeltaX*DeltaY/4
68.                 C[Loop]=DeltaX/2,DeltaY/2,-DeltaX/2,-DeltaY/2
69.             if k==EdgesY and l!=0 and l!=EdgesY:           #Top boundary excl. cor
        ners
70.                 Identifier[Loop] = 8
71.                 ControlVol[Loop] = DeltaX*DeltaY/2
72.                 C[Loop]=DeltaX,DeltaY/2,-DeltaX,-DeltaY/2
73.                 if k==EdgesY and l==EdgesY:                 #Top right corner
74.                     Identifier[Loop] = 9
75.                     ControlVol[Loop] = DeltaX*DeltaY/4
76.                     C[Loop]=DeltaX/2,DeltaY/2,-DeltaX/2,-DeltaY/2
77.
78.                 #Store Edge length of node. This is used to calculate time step
79.                 if Identifier[Loop]==1 or Identifier[Loop]==3 or Identifier[Loop]==7 or
Identifier[Loop]==9:
80.                     NodeEdgeLength[Loop]=DeltaX/2,DeltaY/2
81.                 if Identifier[Loop]==2 or Identifier[Loop]==8:
82.                     NodeEdgeLength[Loop]=DeltaX,DeltaY/2
83.                 if Identifier[Loop]==4 or Identifier[Loop]==6:
84.                     NodeEdgeLength[Loop]=DeltaX/2,DeltaY
85.                 if Identifier[Loop]==5:
86.                     NodeEdgeLength[Loop]=DeltaX,DeltaY
87.                 #Add 1 to loop variable to go to next co-ordinate in list
88.                 Loop=Loop+1
89.
90.
91.     return Identifier,ControlVol,C,NoNodes,DeltaX,DeltaY,NodeEdgeLength

```

Appendix B: CFDsolver.py

```
1. #CFDSolver.py
2. #Description: Solve Flow for 2D Mesh
3. #Author: Neil Mackenzie
4. import numpy as np
5. import MeshGenerator as mg
6. import Steps as st
7. import Graphs as gr
8. import Functions as f
9.
10. def main(n,m):
11.     #Input Variables
12.     Mu=0.01
13.     rho=1
14.     CFL=0.5
15.     #Run Solver using input variables
16.     CFDSolver(n,m,rho,Mu,CFL)
17.
18. def InitialConditions(U,rho,Identifier,NoNodes):
19.     #Run for loop to force velocities along top boundary (excl corners) to be 1
20.     #Boundary and corner velocities everywhere else are 0
21.     for i in range (NoNodes):
22.         if Identifier[i]==8:
23.             U[i,0]=1
24.             U[i,1]=0
25.         if Identifier[i]==1 or Identifier[i]==2 or Identifier[i]==3 or Identifier[i]
26.         ]==4 or Identifier[i]==6 or Identifier[i]==7 or Identifier[i]==9:
27.             U[i,0]=0
28.             U[i,1]=0
29.
30.     #W = density X velocity at every node
31.     W=rho*U
32.     return U,W
33.
34. def TimeStep(NoNodes,NodeEdgeLength,CFL,rho,Mu,U):
35.     #Start with large initial DeltaT
36.     DeltaT=10000
37.     #Check DeltaT at each node.
38.     for i in range (NoNodes):
39.         DeltaXSquared = (NodeEdgeLength[i,0]**2)
40.         DeltaYSquared = (NodeEdgeLength[i,1]**2)
41.         DeltaXEff=(1/(1/DeltaXSquared+1/DeltaYSquared))*0.5
42.         U_diff=((U[i,0]**2+U[i,1]**2)*0.5)+Mu/(rho*0.45*DeltaXEff)
43.         T=CFL*DeltaXEff/U_diff
44.         #If DeltaT at current node is smaller than current DeltaT, store new DeltaT
45.         if T<DeltaT:
46.             DeltaT=T
47.     return DeltaT
48.
49. def Residual(U_old,U_new,Pressure_New,Pressure_Old,NoNodes):
50.     #Calculate residual of each velocity component as well as pressure
51.     #using residual formula provided on vlua
52.     Resi_XVelo=((sum(((U_new[:,0])-(U_old[:,0]))**2))/NoNodes)**0.5
53.     Resi_YVelo=((sum(((U_new[:,1])-(U_old[:,1]))**2))/NoNodes)**0.5
54.     Resi_Press=((sum(((Pressure_New[:,0])-(Pressure_Old[:,0]))**2))/NoNodes)**0.5
55.
56.     #If residual is small enough, return 'False' to terminate for loop
57.     #Otherwise, return True and add 1 to no. of iterations
58.     if Resi_XVelo <=1*10**-5 and Resi_YVelo <=1*10**-5 and Resi_Press <=1*10**-5:
59.         return False,0,Resi_XVelo,Resi_YVelo,Resi_Press
60.     else:
61.         return True,1,Resi_XVelo,Resi_YVelo,Resi_Press
```

```

62. def CFDSolver(n,m,rho,Mu,CFL):
63.     #Run MeshGenerator to obtain Identifiers, Control Volumes, Face Normals etc
64.     #for each node
65.     Identifier,ControlVol,C,NoNodes,DeltaX,DeltaY,NodeEdgeLength=mg.NodeTable(n,m)

66.
67.     #Initialise U and Solve initial conditions for lid-driven cavity
68.     U_zero=np.zeros((NoNodes,2))
69.     U,W=InitialConditions(U_zero,rho,Identifier,NoNodes)
70.
71.     #Calculate Minimum Time Step
72.     DeltaT = TimeStep(NoNodes,NodeEdgeLength,CFL,rho,Mu,U)
73.
74.     #Pre-Compute A Matrix for Pressures (and its inverse) to save time in while-
    loop
75.     AMatrix=f.MatrixA(NodeEdgeLength,DeltaX,DeltaY,ControlVol,NoNodes,n,m,Identifie
    r)
76.     A_inv=np.linalg.inv(AMatrix)
77.
78.     #Start iteration and residual lists and begin with 1st iteration
79.     Iteration = 1
80.     IterationList=[]
81.     ResidualList=[[[]],[[]],[[]]]
82.     Run = True
83.     #Enter while loop to continue calculation while Run is 'True'
84.     #Run will be set to 'False' when the residual calculated at the end of the
85.     #while loop is small enough
86.     while Run:
87.         IterationList.append(Iteration)
88.
89.         #Run Step 1 to calculate DeltaWStar
90.         DeltaWStar = st.Step1(U,W,NoNodes,n,m,ControlVol,Mu,DeltaX,DeltaY,Identifie
    r,C,DeltaT)
91.         #Store current U as U_Old for residual comparison
92.         U_Old = U
93.
94.         #Run Step 2
95.         #Set pressure = 0 if first iteration or store previous iteration pressure
96.         #for residual comparison
97.         if Iteration == 1:
98.             Pressure_Old=np.zeros((NoNodes,1))
99.         else:
100.             Pressure_Old = Pressure_New
101.             #Calculate pressure implicitly using inverse of Matrix A calculated
    above
102.             Pressure_New=st.Step2(U,DeltaWStar,NoNodes,n,m,Identifier,ControlVol
    ,C,NodeEdgeLength,DeltaX,DeltaY,rho,DeltaT,A_inv)
103.
104.             #Run Step 3
105.             #Calculate velocity at next time step
106.             U_New=st.Step3(Pressure_New,Identifier,DeltaT,U,rho,n,m,C,NoNodes,Co
    ntrolVol,DeltaWStar)
107.
108.             #Check Residual
109.             #Re-Apply Boundary Conditions
110.             U,W=InitialConditions(U_New,rho,Identifier,NoNodes)
111.             #Check residual of velocity components and pressure
112.             #This will return 'Run' as 'True' or 'False' depending on size of re
    siduals
113.             Run,AddIteration,ResX,ResY,ResP = Residual(U_Old,U,Pressure_New,Pres
    sure_Old,NoNodes)
114.             #Add 1 to iteration if it must continue, add 0 if not
115.             Iteration = Iteration + AddIteration
116.
117.             #Populate residual lists for plotting
118.             ResidualList[0].append(ResX)

```

```

119.         ResidualList[1].append(ResY)
120.         ResidualList[2].append(ResP)
121.
122.         #Print confirmation with number of iterations required
123.         print("=====
=====
=====")
124.         print("Calculation Completed for " +str(m) + " by " + str (n) + " node m
esh after " + str(Iteration)+ " iterations")
125.
126.         #Graphing Information
127.         U_Final=U_New
128.         Pressure_Final=Pressure_New[:,0]
129.         #Calculate magnitude using sqrt(x^2+y^2)
130.         U_Final_Magnitude=(U_Final[:,0]**2+U_Final[:,1]**2)**0.5
131.         #Colour Plots using Plotter.py
132.         gr.Question1Plot(U_Final_Magnitude,U_Final,Pressure_Final,n,m,IterationL
ist,ResidualList)
133.
134.         #Determine Velocities along x = 0.5 for Question 2
135.         U_X_MidPt=[]
136.         U_Y_MidPt=[]
137.         Y_Coord=[0]
138.         for i in range (n):
139.             U_X_MidPt.append(U_Final[m//2+i*n,0])
140.             if i!=0:
141.                 Y_Coord.append(i/(n-1))
142.         for j in range (m):
143.             U_Y_MidPt.append(U_Final[n//2+j*m,1])
144.         #Plot graphs for question 2
145.         gr.Question2Plot(U_X_MidPt,U_Y_MidPt,Y_Coord,n,m)
146.
147.
148.         #Run the main function for 21 X 21 and 51 X 51
149.         main(21,21)
150.         main(51,51)

```

Appendix C: Steps.py

```
1. #Steps.py
2. #Description: Perform 3 steps for 2D flow solver
3. #Author: Neil Mackenzie
4. import MeshGenerator as mg
5. import numpy as np
6. import Functions as f
7.
8.
9. def Step1(U,W,NoNodes,n,m,ControlVol,Mu,DeltaX,DeltaY,Identifier,C,DeltaT):
10.     #Initialise matrices for step 1
11.     Convective = np.zeros((NoNodes,2))
12.     Diffusive = np.zeros((NoNodes,2))
13.     DeltaWStar=np.zeros((NoNodes,2))
14.     #Loop through each node to determine convective and diffusive term before
15.     #using these to determine DeltaWStar for each node
16.     for i in range (NoNodes):
17.         #Convective
18.         #Determine central differenced Wstar excluding boundary face contributions
19.         U_CD_NdotC,U_CD_EdotC,U_CD_SdotC,U_CD_WdotC=f.NablaDotPhi(U,C,Identifier[i]
20.         ,i,n,m)
21.         #Find Upwind node for W
22.         Upwind_N=Upwind(U_CD_NdotC,'N')
23.         Upwind_E=Upwind(U_CD_EdotC,'E')
24.         Upwind_S=Upwind(U_CD_SdotC,'S')
25.         Upwind_W=Upwind(U_CD_WdotC,'W')
26.         #Convective in x using W_x
27.         Convective[i,0]=1/ControlVol[i]*((W[i+Upwind_E,0])*(U_CD_EdotC)+(W[i+Upwind
28.         _W,0])*(U_CD_WdotC)+W[i+Upwind_S*n,0]*(U_CD_SdotC)+(W[i+Upwind_N*n,0])*(U_CD_NdotC)
29.         )
30.         #Convective in y using W_y
31.         Convective[i,1]=1/ControlVol[i]*((W[i+Upwind_E,1])*(U_CD_EdotC)+(W[i+Upwind
32.         _W,1])*(U_CD_WdotC)+W[i+Upwind_S*n,1]*(U_CD_SdotC)+(W[i+Upwind_N*n,1])*(U_CD_NdotC)
33.         )
34.         #=====
35.         #Diffusive
36.         #Corners are not included here because they will always be 0
37.         if Identifier[i]==5:
38.             #Internal Nodes include all terms
39.             Diffusive[i,0]=Mu*((U[i-1,0]-2*U[i,0]+U[i+1,0])/(DeltaX**2)+(U[i-n,0]-
40.             2*U[i,0]+U[i+n,0])/(DeltaY**2))
41.             Diffusive[i,1]=Mu*((U[i-1,1]-2*U[i,1]+U[i+1,1])/(DeltaX**2)+(U[i-n,1]-
42.             2*U[i,1]+U[i+n,1])/(DeltaY**2))
43.         if Identifier[i]==4 or Identifier[i]==6:
44.             #East/West boundaries exclude west and east terms since these cancel
45.             Diffusive[i,0]=Mu/(DeltaY**2)*(U[i-n,0]-2*U[i,0]+U[i+n,0])
46.             Diffusive[i,1]=2*Mu/(DeltaX**2)*(U[i-n,1]-2*U[i,1]+U[i+n,1])
47.         if Identifier[i]==2 or Identifier[i]==8:
48.             #North/South boundaries exclude north and west terms since these cancel
49.             Diffusive[i,0]=Mu/(DeltaY**2)*(U[i-1,0]-2*U[i,0]+U[i+1,0])
50.             Diffusive[i,1]=2*Mu/(DeltaX**2)*(U[i+1,1]-2*U[i,1]+U[i-1,1])
51.         #Compute DeltaWStar
52.         DeltaWStar[i,0]=DeltaT*(Diffusive[i,0]-Convective[i,0])
53.         DeltaWStar[i,1]=DeltaT*(Diffusive[i,1]-Convective[i,1])
```



```

53.     return DeltaWStar
54.
55. def Step2(U,WStar,NoNodes,n,m,Identifier,ControlVol,C,NodeEdgeLength,DeltaX,DeltaY,
    rho,DeltaT,A_inv):
56.     #Initialise matrices for Step 2
57.     GradU=np.zeros((NoNodes,1))
58.     GradWStar=np.zeros((NoNodes,1))
59.     BMatrix=np.zeros((NoNodes,1))
60.     for i in range (NoNodes):
61.         #GradU Term
62.         #Determine central differenced velocities
63.         U_CD_NdotC,U_CD_EdotC,U_CD_SdotC,U_CD_WdotC=f.UCDdotC(U,C,Identifier[i],i,n
    ,m)
64.         GradU[i]=1/ControlVol[i]*((U_CD_EdotC)+(U_CD_WdotC)+(U_CD_NdotC)+(U_CD_Sdot
    C))
65.
66.         #Grad DeltaWStar Term
67.         #Determine central differenced Wstar excluding boundary face contributions
68.         WStar_CD_NdotC,WStar_CD_EdotC,WStar_CD_SdotC,WStar_CD_WdotC=f.NablaDotPhi(W
    Star,C,Identifier[i],i,n,m)
69.         GradWStar[i]=1/ControlVol[i]*((WStar_CD_EdotC)+(WStar_CD_WdotC)+(WStar_CD_N
    dotC)+(WStar_CD_SdotC))
70.
71.         #Calculate B Matrix using GradU and GradDeltaWStar
72.         BMatrix[i]=rho/DeltaT*(GradU[i]+GradWStar[i])
73.         #Reset top line of B matrix for reference pressure
74.         BMatrix[0]=0
75.         #Calculate pressure at each node using A_inverse dotted with B
76.         Pressure=A_inv.dot(BMatrix)
77.
78.     return Pressure
79.
80. def Step3(Pressure,Identifier,DeltaT,U,rho,n,m,C,NoNodes,ControlVol,DeltaWStar):
81.     #Initialise Matrices for Step 3
82.     U_New=np.zeros((NoNodes,2))
83.     GradP=np.zeros((NoNodes,2))
84.     #Calculate GradP in x and y-directions for each node
85.     for i in range(NoNodes):
86.         #Obtain central differenced pressures for each face
87.         Press_N,Press_E,Press_S,Press_W=f.PDotC(Pressure,C,Identifier[i],i,n,m)
88.         GradP[i,0]=1/ControlVol[i]*((Press_E)+(Press_W))
89.         GradP[i,1]=1/ControlVol[i]*((Press_N)+(Press_S))
90.         #Determine next velocity implicitly
91.         U_New = U - (DeltaT/rho)*GradP+(1/rho)*DeltaWStar
92.
93.     return U_New
94.
95. def Upwind(U_CDdotC,Face):
96.     #Determine Upwind node by checking if U dot C is positive or negative
97.     #where C is the area face normal
98.     if U_CDdotC <0:
99.         #Take node above or to the right if north or east face
100.         if Face=="N" or Face=="E":
101.             return 1
102.         #Take node below or to the left if south or west face
103.         if Face=="S" or Face=="W":
104.             return -1
105.         #Use internal node for all faces if U dot C is greater than 0
106.     if U_CDdotC >=0:
107.         return 0

```

Appendix D: Functions.py

```
1. #Functions.py
2. #Description: Perform discretisations for 3-step methodology
3. #Author: Neil Mackenzie
4. import numpy as np
5.
6. def UCDDotC(U,C,Identifier,i,n,m):
7.     #Calculate Central Differenced velocity at each type of node
8.     if Identifier==1:                                #Bottom left corner
9.         U_CD_N=(U[i,1]+U[i+n,1])/2
10.        U_CD_S=U[i,1]
11.        U_CD_W=U[i,0]
12.        U_CD_E=(U[i,0]+U[i+1,0])/2
13.    if Identifier ==2:                                #Bottom boundary
14.        U_CD_N=(U[i,1]+U[i+n,1])/2
15.        U_CD_S=U[i,1]
16.        U_CD_W=(U[i,0]+U[i-1,0])/2
17.        U_CD_E=(U[i,0]+U[i+1,0])/2
18.    if Identifier ==3:                                #Bottom right corner
19.        U_CD_N=(U[i,1]+U[i+n,1])/2
20.        U_CD_S=U[i,1]
21.        U_CD_W=(U[i,0]+U[i-1,0])/2
22.        U_CD_E=U[i,0]
23.    if Identifier ==4:                                #Left Boundary
24.        U_CD_N=(U[i,1]+U[i+n,1])/2
25.        U_CD_S=(U[i,1]+U[i-n,1])/2
26.        U_CD_W=U[i,0]
27.        U_CD_E=(U[i,0]+U[i+1,0])/2
28.    if Identifier==5:                                #Internal Node
29.        U_CD_N=(U[i,1]+U[i+n,1])/2
30.        U_CD_S=(U[i,1]+U[i-n,1])/2
31.        U_CD_W=(U[i,0]+U[i-1,0])/2
32.        U_CD_E=(U[i,0]+U[i+1,0])/2
33.    if Identifier==6:                                #Right Boundary
34.        U_CD_N=(U[i,1]+U[i+n,1])/2
35.        U_CD_S=(U[i,1]+U[i-n,1])/2
36.        U_CD_W=(U[i,0]+U[i-1,0])/2
37.        U_CD_E=U[i,0]
38.    if Identifier==7:                                #Top left corner
39.        U_CD_N=U[i,1]
40.        U_CD_S=(U[i,1]+U[i-n,1])/2
41.        U_CD_W=U[i,0]
42.        U_CD_E=(U[i,0]+U[i+1,0])/2
43.    if Identifier==8:                                #Top boundary
44.        U_CD_N=U[i,1]
45.        U_CD_S=(U[i,1]+U[i-n,1])/2
46.        U_CD_W=(U[i,0]+U[i-1,0])/2
47.        U_CD_E=(U[i,0]+U[i+1,0])/2
48.    if Identifier==9:                                #Top right corner
49.        U_CD_N=U[i,1]
50.        U_CD_S=(U[i,1]+U[i-n,1])/2
51.        U_CD_W=(U[i,0]+U[i-1,0])/2
52.        U_CD_E=U[i,0]
53.
54.    #Perform dot product with face normal
55.    U_CD_NdotC= U_CD_N*(C[i,0])
56.    U_CD_EdotC= U_CD_E*(C[i,1])
57.    U_CD_SdotC= U_CD_S*(C[i,2])
58.    U_CD_WdotC= U_CD_W*(C[i,3])
59.
60.    return U_CD_NdotC,U_CD_EdotC,U_CD_SdotC,U_CD_WdotC
61.
62. def NablaDotPhi(U,C,Identifier,i,n,m):
```

```

63.     #This function is the same as UCDotC, but excludes boundary face
64.     #contributions for the convective and DeltaW* terms
65.     if Identifier==1:
66.         U_CD_N=(U[i,1]+U[i+n,1])/2
67.         U_CD_S=0
68.         U_CD_W=0
69.         U_CD_E=(U[i,0]+U[i+1,0])/2
70.     if Identifier ==2:
71.         U_CD_N=(U[i,1]+U[i+n,1])/2
72.         U_CD_S=0
73.         U_CD_W=(U[i,0]+U[i-1,0])/2
74.         U_CD_E=(U[i,0]+U[i+1,0])/2
75.     if Identifier ==3:
76.         U_CD_N=(U[i,1]+U[i+n,1])/2
77.         U_CD_S=0
78.         U_CD_W=(U[i,0]+U[i-1,0])/2
79.         U_CD_E=0
80.     if Identifier ==4:
81.         U_CD_N=(U[i,1]+U[i+n,1])/2
82.         U_CD_S=(U[i,1]+U[i-n,1])/2
83.         U_CD_W=0
84.         U_CD_E=(U[i,0]+U[i+1,0])/2
85.     if Identifier==5:
86.         U_CD_N=(U[i,1]+U[i+n,1])/2
87.         U_CD_S=(U[i,1]+U[i-n,1])/2
88.         U_CD_W=(U[i,0]+U[i-1,0])/2
89.         U_CD_E=(U[i,0]+U[i+1,0])/2
90.     if Identifier==6:
91.         U_CD_N=(U[i,1]+U[i+n,1])/2
92.         U_CD_S=(U[i,1]+U[i-n,1])/2
93.         U_CD_W=(U[i,0]+U[i-1,0])/2
94.         U_CD_E=0
95.     if Identifier==7:
96.         U_CD_N=0
97.         U_CD_S=(U[i,1]+U[i-n,1])/2
98.         U_CD_W=0
99.         U_CD_E=(U[i,0]+U[i+1,0])/2
100.         if Identifier==8:
101.             U_CD_N=0
102.             U_CD_S=(U[i,1]+U[i-n,1])/2
103.             U_CD_W=(U[i,0]+U[i-1,0])/2
104.             U_CD_E=(U[i,0]+U[i+1,0])/2
105.         if Identifier==9:
106.             U_CD_N=0
107.             U_CD_S=(U[i,1]+U[i-n,1])/2
108.             U_CD_W=(U[i,0]+U[i-1,0])/2
109.             U_CD_E=0
110.
111.         U_CD_NdotC= U_CD_N*C[i,0]
112.         U_CD_EdotC= U_CD_E*C[i,1]
113.         U_CD_SdotC= U_CD_S*C[i,2]
114.         U_CD_WdotC= U_CD_W*C[i,3]
115.
116.         return U_CD_NdotC,U_CD_EdotC,U_CD_SdotC,U_CD_WdotC
117.
118.     def MatrixA(NodeEdgeLength,DeltaX,DeltaY,ControlVol,NoNodes,n,m,Identifier):
119.
120.         #This function generates Matrix A for pressure calculationusing the
121.         #vertical and horizontal coefficients determined at each node
122.         A=np.zeros((NoNodes,NoNodes))
123.         for i in range (NoNodes):
124.             CoeffVert=NodeEdgeLength[i,1]/(ControlVol[i]*DeltaX) #East and West
125.             CoeffHori=NodeEdgeLength[i,0]/(ControlVol[i]*DeltaY) #North and Sout

```

```

126.         if Identifier[i]==1:
127.             #Store as 1 to use as reference pressure
128.             A[i,i]=1
129.         if Identifier[i]==2:
130.             A[i,i-1]=CoeffVert
131.             #Double vertical component due to absence of south face
132.             A[i,i]=-2*CoeffVert-CoeffHori
133.             A[i,i+1]=CoeffVert
134.             A[i,i+n]=CoeffHori
135.         if Identifier[i]==3:
136.             A[i,i-1]=CoeffVert
137.             A[i,i]=-CoeffVert-CoeffHori
138.             A[i,i+n]=CoeffHori
139.         if Identifier[i]==4:
140.             #Double horizontal component due to absence of west face
141.             A[i,i]=-CoeffVert-2*CoeffHori
142.             A[i,i+1]=CoeffVert
143.             A[i,i+n]=CoeffHori
144.             A[i,i-n]=CoeffHori
145.         if Identifier[i]==5:
146.             A[i,i-1]=CoeffVert
147.             A[i,i]=-2*CoeffVert-2*CoeffHori
148.             A[i,i+1]=CoeffVert
149.             A[i,i+n]=CoeffHori
150.             A[i,i-n]=CoeffHori
151.         if Identifier[i]==6:
152.             A[i,i-1]=CoeffVert
153.             #Double horizontal component due to absence of east face
154.             A[i,i]=-CoeffVert-2*CoeffHori
155.             A[i,i+n]=CoeffHori
156.             A[i,i-n]=CoeffHori
157.         if Identifier[i]==7:
158.             A[i,i]=-CoeffVert-CoeffHori
159.             A[i,i+1]=CoeffVert
160.             A[i,i-n]=CoeffHori
161.         if Identifier[i]==8:
162.             #Double vertical component due to absence of south face
163.             A[i,i-1]=CoeffVert
164.             A[i,i]=-2*CoeffVert-CoeffHori
165.             A[i,i+1]=CoeffVert
166.             A[i,i-n]=CoeffHori
167.         if Identifier[i]==9:
168.             A[i,i-1]=CoeffVert
169.             A[i,i]=-CoeffVert-CoeffHori
170.             A[i,i-n]=CoeffHori
171.
172.     return A
173.
174. def PDotC(P,C,Identifier,i,n,m):
175.     #Same as above functions, but pressure vector only has 1 component
176.     if Identifier==1:
177.         P_CD_N=(P[i]+P[i+n])/2
178.         P_CD_S=P[i]
179.         P_CD_W=P[i]
180.         P_CD_E=(P[i]+P[i+1])/2
181.     if Identifier ==2:
182.         P_CD_N=(P[i]+P[i+n])/2
183.         P_CD_S=P[i]
184.         P_CD_W=(P[i]+P[i-1])/2
185.         P_CD_E=(P[i]+P[i+1])/2
186.     if Identifier ==3:
187.         P_CD_N=(P[i]+P[i+n])/2
188.         P_CD_S=P[i]
189.         P_CD_W=(P[i]+P[i-1])/2
190.         P_CD_E=(P[i])
191.     if Identifier ==4:

```

```

192.         P_CD_N=(P[i]+P[i+n])/2
193.         P_CD_S=(P[i]+P[i-n])/2
194.         P_CD_W=P[i]
195.         P_CD_E=(P[i]+P[i+1])/2
196.     if Identifier==5:
197.         P_CD_N=(P[i]+P[i+n])/2
198.         P_CD_S=(P[i]+P[i-n])/2
199.         P_CD_W=(P[i]+P[i-1])/2
200.         P_CD_E=(P[i]+P[i+1])/2
201.     if Identifier==6:
202.         P_CD_N=(P[i]+P[i+n])/2
203.         P_CD_S=(P[i]+P[i-n])/2
204.         P_CD_W=(P[i]+P[i-1])/2
205.         P_CD_E=(P[i])
206.     if Identifier==7:
207.         P_CD_N=(P[i])
208.         P_CD_S=(P[i]+P[i-n])/2
209.         P_CD_W=(P[i])
210.         P_CD_E=(P[i]+P[i+1])/2
211.     if Identifier==8:
212.         P_CD_N=(P[i])
213.         P_CD_S=(P[i]+P[i-n])/2
214.         P_CD_W=(P[i]+P[i-1])/2
215.         P_CD_E=(P[i]+P[i+1])/2
216.     if Identifier==9:
217.         P_CD_N=(P[i])
218.         P_CD_S=(P[i]+P[i-n])/2
219.         P_CD_W=(P[i]+P[i-1])/2
220.         P_CD_E=(P[i])
221.     P_CD_NdotC= P_CD_N*(C[i,0])
222.     P_CD_EdotC= P_CD_E*(C[i,1])
223.     P_CD_SdotC= P_CD_S*(C[i,2])
224.     P_CD_WdotC= P_CD_W*(C[i,3])
225.
226.     return P_CD_NdotC,P_CD_EdotC,P_CD_SdotC,P_CD_WdotC

```

Appendix E: Graphs.py

```
1. #Graphs.py
2. #Description: Plot graphs for Question 1 and 2 of Assignment 3B
3. #Author: Neil Mackenzie
4. import Plotter as pl
5. import pylab as py
6. import matplotlib as mpl
7. from matplotlib.rcsetup import cycler
8.
9. def Question1Plot(Velo_Magnitude,Velo,Pressure,n,m,IterationList,ResidualList):
10.     #Send information to Plotter.py as required by Bevan Jones' code
11.     Graphs=pl.Plotter("MEC4045F Assignment 3B\nPlots for a "+str(n)+" by "+str(m)+"
12.         Node Mesh",2,3,5)
13.     Graphs.Add1DPlot(1,"Convergence","Iteration","Residual","log",['-','-','-
14.         ',],True)
15.     Graphs.Add2DPlot(2,"Colour Plot of Velocity Magnitude","x (m)","y (m)", m,n,Tru
16.         e)
17.     Graphs.Add2DPlot(3,"Colour Plot of Pressure","x (m)","y (m)",m,n,True)
18.     Graphs.Add2DPlot(4,"Colour Plot of X-Velocity","x (m)","y (m)",m,n,True)
19.     Graphs.Add2DPlot(5,"Colour Plot of Y-Velocity","x (m)","y (m)",m,n,True)
20.     Legend_List = ["X-Velocity","Y-Velocity","Pressure"]
21.
22.     Graphs.Update1DPlotData(1,IterationList,ResidualList,Legend_List)
23.     Graphs.Update2DPlotData(2,Velo_Magnitude,"Velocity (m/s)")
24.     Graphs.Update2DPlotData(3,Pressure,"Pressure (Pa)")
25.     Graphs.Update2DPlotData(4,Velo[:,0],"X-Velocity (m/s)")
26.     Graphs.Update2DPlotData(5,Velo[:,1],"Y-Velocity (m/s)")
27.
28. def Question2Plot(U_X,U_Y,Y_Coord,n,m):
29.     #Set colour cycle
30.     mpl.rcParams['axes.prop_cycle']=cycler('color',['r','b','g'])
31.     #Plot X-velocity vs y-height
32.     py.figure(2)
33.     Label = str(m) + "X" + str(n) + " Node Mesh"
34.     py.plot(U_X,Y_Coord,label= Label)
35.     py.title("X-Velocity at x = 0.5")
36.     py.xlabel("Y-Coordinate (m)")
37.     py.ylabel("X-Velocity (m/s)")
38.     py.legend()
39.
40.     #Plot Y-velocity vs y-height
41.     py.figure(3)
42.     Label = str(m) + "X" + str(n) + " Node Mesh"
43.     py.plot(U_Y,Y_Coord, label = Label)
44.     py.title("Y-Velocity at x = 0.5")
45.     py.xlabel("Y-Coordniate (m)")
46.     py.ylabel("Y-Velocity (m/s)")
47.     py.legend()
```