Explicit Incompressible Euler Solver

Computational Aerodynamics - AS5330
Group 10

Ishan Indurkar [AE18B005]

Department of Aerospace Engineering
Indian Institute of Technology Madras

Acknowledgement

We would like to express our special thanks of gratitude to Professor Shantanu Ghosh, for his guidance and advice to complete this project. Well planned course structure helped us in doing a lot of Research and involve in constant discussion.

Table of Contents

List of Figures

Abstract

The report includes the methodology followed to solve the incompressible Euler equations using explicit, finite volume techniques. A computer program is written for this purpose. The code is tested on a circular arc-bump geometry. Artificial compressibility approach is used to time-march the solution and a steady state solution is attained. It also include contour plots (colour lines) of flow variables, velocity vector plots, convergence histories (log scale), surface pressure (line plot)

1. Introduction

The system of incompressible Euler equations is generally solved using time-marching schemes. However, there is a need to introduce a pressure derivative term in the continuity equation for the time marching technique to work. Hence, the artificial compressibility approach is followed. Artificial acoustic speed is used to formulate the incompressibility in a way that the pressure derivative term vanishes for the steady state solution ensuring that the steady flow field is physical. The system, which is hyperbolic in space and time, now converts to an initial value problem which is solved by finite volume discretization of the integral form of the governing equations and using a first order explicit time integration scheme to advance the solution.

2. Methodology

2.1 Euler Equations

2-D compressible Euler equations in Cartesian coordinates in vector form can be written as

$$\begin{Bmatrix} \rho \\ \rho u \\ \rho v \\ \rho e_t \end{Bmatrix}_t + \begin{Bmatrix} \rho u \\ \rho u^2 + P \\ \rho uv \\ \rho u h_0 \end{Bmatrix}_x + \begin{Bmatrix} \rho v \\ \rho uv \\ \rho v^2 + P \\ \rho v h_0 \end{Bmatrix}_y = \begin{Bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix} \tag{1}$$

2-D incompressible Euler equations in Cartesian coordinates in vector form can be written as

$$\begin{Bmatrix} 0 \\ \rho u \\ \rho v \end{Bmatrix}_t + \begin{Bmatrix} u \\ \rho u^2 + p \\ \rho uv \end{Bmatrix}_x + \begin{Bmatrix} v \\ \rho uv \\ \rho v^2 + p \end{Bmatrix}_y = \begin{Bmatrix} 0 \\ 0 \\ 0 \end{Bmatrix} \tag{2}$$

where ρ = constant.

2.2 Artificial compressibility approach

The system of equations in quasi-linear form :
Equation of Continuity :

$$\frac{1}{\beta^2} P_t + \rho u_x + \rho v_y = 0 \tag{3}$$

where β:artifical acoustic speed
X-momentum:

$$\rho[u_t + uu_x + vu_y] + P_x = 0 \tag{4}$$

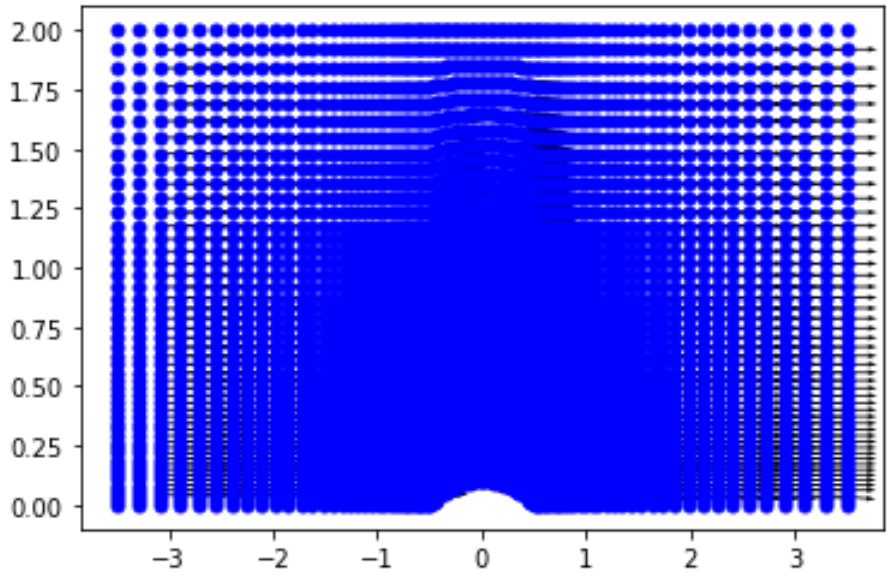
Y-momentum:

$$\rho[v_t + uv_x + vv_y] + P_y = 0 \tag{5}$$

$$\oint_{Vol_{cv}} M\frac{\partial V}{\partial t} dVol_{Cv} + \oint_{A_{CV}} (\bar{F}.\bar{n}) dA_{cv} = 0 \tag{6}$$

where M= $\begin{bmatrix} \frac{1}{\beta^2} & 0 & 0 \\ \frac{u}{\beta^2} & \rho & 0 \\ \frac{v}{\beta^2} & 0 & \rho \end{bmatrix}$, $V = \begin{bmatrix} P \\ u \\ v \end{bmatrix}, F_x = \begin{bmatrix} \rho u \\ \rho u^2 + P \\ \rho uv \end{bmatrix} F_y = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + P \end{bmatrix}$

2.3 Grid Structure



2.4 Flux Formulation

Flux is obtained with the advection based upwinding scheme at the cell interface. We define flux as F then the flux have three significance: mass flux, U-direction based momentum flux and V-Direction based momentum flux (in a 2 Dimensional orthogonal axis x,y - system)

We redefine F as $\bar{F}_{i+1/2}$ where i+1/2 correspond to the interface between i and i+1.

$$\implies \bar{F}_{(i+1/2)} = A_{i+1/2} \times \left(\rho U^+_{i+1/2} \begin{Bmatrix} 1 \\ u \\ v \end{Bmatrix}_i + \rho U^-_{i+1/2} \begin{Bmatrix} 1 \\ u \\ v \end{Bmatrix}_{i+1} + P_{i+1/2} \begin{Bmatrix} 0 \\ n_x \\ n_y \end{Bmatrix} \right)$$

2.5 Artificial Dissipation

The elliptic-hyperbolic advective upwinding Euler system often may not ensure oscillation free solution. Hence, some dissipation is added to the continuity and momentum equations. The dissipation term is of

the form

$$\frac{c\Delta p}{\lambda}$$

where c is a constant taken to be 0.5 in one case and 1.0 in another case, and is $\max_{ip}$*aramumwavespeednormaltothef*

$$\rho U|^{+}_{i+1/2} = \left(\rho_{max}(0,U_i) + c\frac{P_i - P_{i+1}}{2}\right) A_{i+1/2}$$

$$\dot{m}|^{-}_{i+1/2} = \rho U|^{-}_{i+1/2} = \left(\rho_{min}(0,U_i) + c\frac{P_i - P_{i+1}}{2}\right) A_{i+1/2}$$

$$\dot{m} = \dot{m}|^{+} + \dot{m}|^{-}$$

2.6 Time Step Calculation

To calculate local time steps and march the solution using these values. In this case,

$$\frac{V_i j}{dt} = \frac{\sum_{k=1}^{\infty} A_k |\lambda_k|}{2c} \tag{7}$$

the resulted caused divergence in the solution therefore another dt is used

$$dt = 10^{-4}$$

2.7 Boundary Conditions

Ghost Cell: Ghost cells are used to get the flux at boundary. The convective flux at boundaries is made zero. The idea is to prevent any flow that might have been possible through the slip walls. $U_G = U_I$, $V_G = V_I$ and $P_G = P_{back} = P_b$

Initialisation The $U_{i,j}$ $V_{i,j}$ is initialised with zero, while the $P_{i,j}$ is initialised as value of back pressure.

2.8 Residue

The residual is defined as $\bar{R}_{i+1/2,j} = \bar{F}_{i+1/2,j} - \bar{F}_{i-1/2,j} + \bar{F}_{i,j+1/2} - \bar{F}_{i,j-1/2}$

2.9 Solution Convergence Check

The residual norm is calculated.

$$||\bar{R}|| = \sum_{j=2}^{j_{max}} \sum_{i=2}^{i_{max}} \left[\left(\frac{R^1_{i,j}}{S_1}\right)^2 + \left(\frac{R^2_{i,j}}{S_2}\right)^2 + \left(\frac{R^3_{i,j}}{S_3}\right)^2 \right] \tag{8}$$

Where:
$R^1_{i,j}$ = mass residual, $R^2_{i,j}$= x-momentum residual and $R^3_{i,j}$= y-momentum residual. $S_1 = \rho U_\infty$, $S_2 = S_3 = \rho U^2_\infty$

If the ratio of residual norm after nth iteration and initial residual norm is less than the tolerance value then the solution can be said to have converged.

$$\frac{R_{i,j\,at\,T\,iteration}}{R_{i,j\,at\,first\,iteration}} < 10^{-3} \implies solution\,converged \tag{9}$$

3. Conclusion

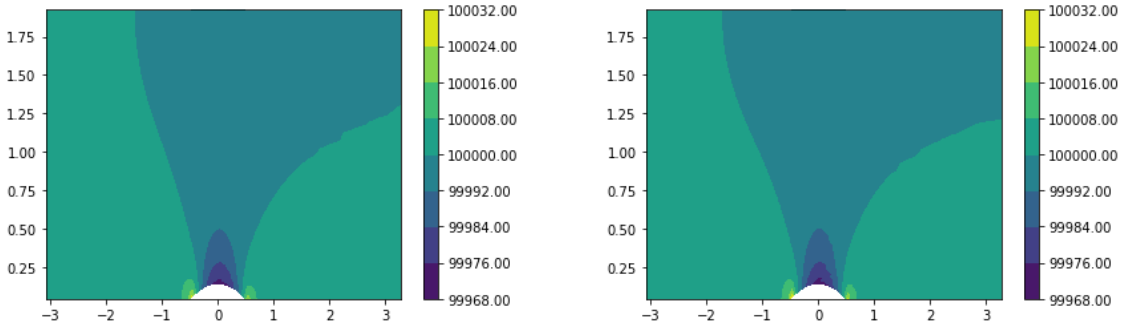
The following conclusion is drawn:

1. The ratio of residual is decreasing which implies convergence of solver and around 10,000 to 50,000 it tends to go below 10^{-3} [therefore we went with n=50,000 iterations for proper solution output]
2. The solver processed the bum grid data with 66% memory usage with process rate 38.46 iteration per sec. The whole processing took 21.667 min. and each iteration took around 0.026 sec.
3. The ratio of residual at c=1 seem to decrease at slightly higher rate as compared to c=0.5
4. Pressure , Velocity contour and lines obtained (Which is the required output from the solver).
5. Residual vs Iteration for CFL 1 and CFL 0.5 is obtained (which suggest a convergence in solution)

4. Results

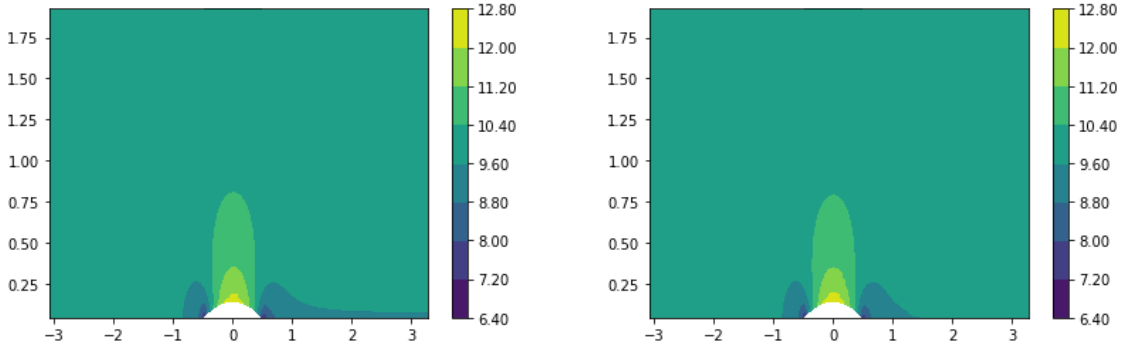
4.1 Choice of CFL

In the current case, the cfl is chosen as C=1 and C=0.5. The following plots and contours were obtained.



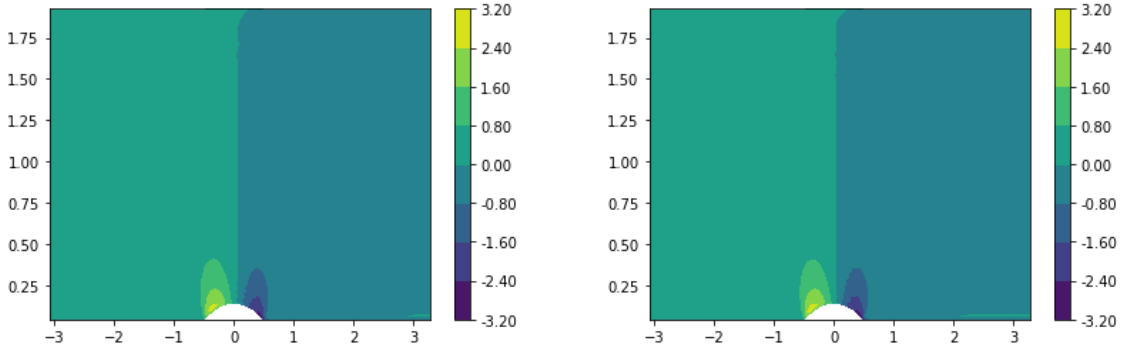
(a) Pressure Contour at c=1 **(b)** Pressure Contour at c=0.5

Fig. 4.1. side by side comparison of Pressure contours for c=1,0.5



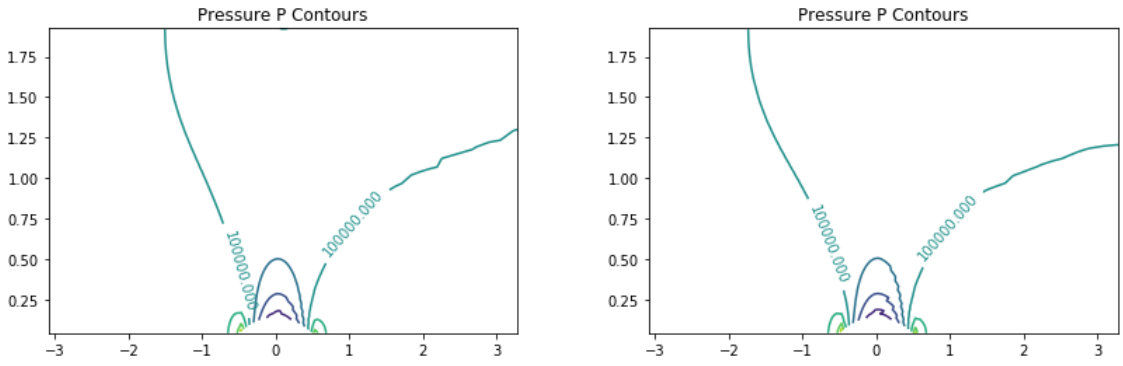
(a) U velocity Contour at c=1 **(b)** U velocity Contour at c=0.5

Fig. 4.2. side by side comparison of U component of velocity contours for c=1,0.5



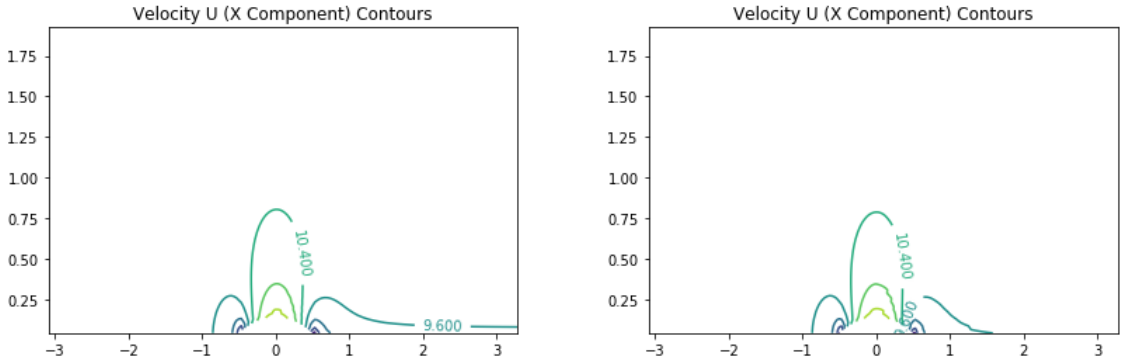
(a) V velocity Contour at c=1 **(b)** V velocity Contour at c=0.5

Fig. 4.3. side by side comparison of V component of velocity contours for c=1,0.5



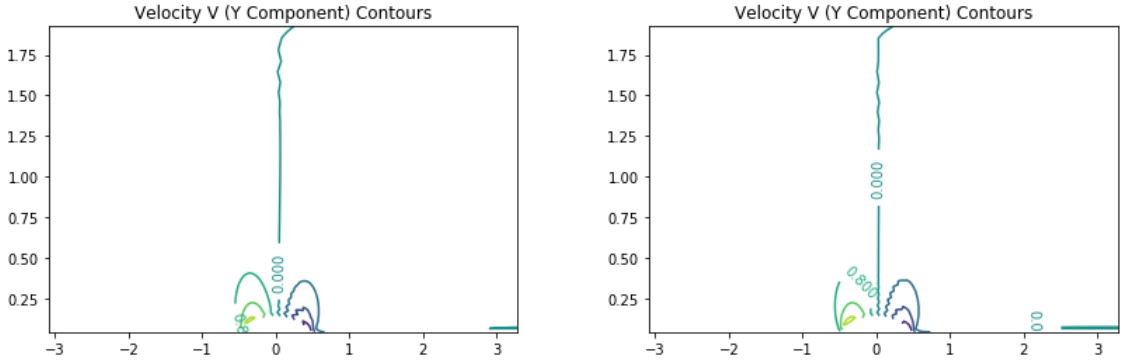
(a) Pressure line at c=1 **(b)** Pressure line at c=0.5

Fig. 4.4. side by side comparison of Pressure contours for c=1,0.5



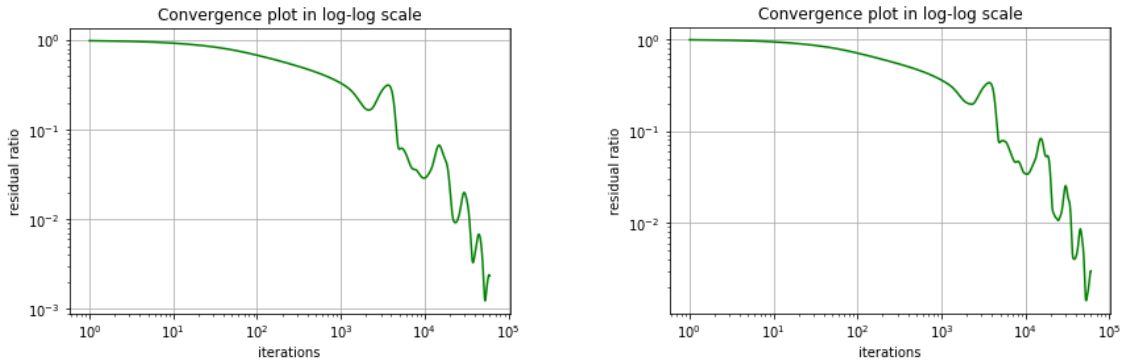
(a) U velocity line at c=1 **(b)** U velocity line at c=0.5

Fig. 4.5. side by side comparison of U component of velocity contours for c=1,0.5



(a) V velocity line at c=1 **(b)** V velocity line at c=0.5

Fig. 4.6. side by side comparison of V component of velocity contours for c=1,0.5



(a) Residual vs Iteration at c=1

(b) Residual vs Iteration at c=0.5

Fig. 4.7. side by side comparison of Residual vs Iteration for c=1,0.5

Appendix A: Code For Solver

A.1 Code importing library

```
# -*- coding: utf-8 -*-
"""

Last Update on Wed Oct 13 18:35:11 2021

"""
#Pre-Processing
import time, os, sys, io  #Estimate time from each function and address of file

#Processing
import numpy as np #Creating array and matrix:

#Post-Processing
import matplotlib.pyplot as plt #plot graphs:
from numpy import exp,arange #properly creating ploting axis lines
from pylab import meshgrid,cm,imshow,contour,clabel,colorbar,axis,title,show
```

A.2 Defined Functions (modules of solver): Pre-Processing

```
def read_mesh(string):
    if not os.path.exists(string):
        print('The file %s does not exist. Program will exit...' %(string))
        sys.exit()
    else:
        startTime=time.time()
        print('Reading Mesh data from %s...'%(string))
    # Read data from file
    P = np.loadtxt(string)
```

```
    imax=int(P[0,0])
    jmax=int(P[0,1])
    # Initialise arrays to store X(i,j) and Y(i,j)
    X=np.zeros((imax+1,jmax+1))
    Y=np.zeros((imax+1,jmax+1))

    i=1 # Starting value of i=1
    j=1 # Starting value of j=1
    #Store data into X and Y
    for k in range(1,len(P)): # Loop to scan through all data points
        X[i,j]=P[k,0]
        Y[i,j]=P[k,1]
        if j<jmax+1: # Check if right end is reached or not
            if i<imax: # if right end is not reached, increment i, keep j unchanged
                i+=1;
            else: # if right end is reached, increment j , set i=1
                j+=1
                i=1
    totalTime = round(time.time() - startTime, 2)
    print('Completed in : %s seconds' % (totalTime))
    return X,Y,imax,jmax # return X(i,j) and Y(i,j)
```

A.3 Defined Functions (modules of solver): Processing

```
######################## CELL AREA Calculator

def calc_area_triangle(x1,y1,x2,y2,x3,y3):

    d1=np.sqrt((x2-x1)**2+(y2-y1)**2) #length of first line
    d2=np.sqrt((x3-x2)**2+(y3-y2)**2) #length of second line
    r1x=(x2-x1) # X component of first line vector
    r1y=(y2-y1) # Y component of first line vector
    r2x=(x3-x2) # X component of second line vector
    r2y=(y3-y2) # Y component of second line vector
    t1=r1x*r2x+r1y*r2y # t1 stores the dot product value of the two line vectors
    theta=np.arccos(t1/(d1*d2))
    # theta stores the value of the acute angle between theline vectors.
    return np.abs(0.5*d1*d2*np.sin(theta))

######################## CELL AREA
def calc_cell_areas(X,Y,imax,jmax):



    print("Calculating Cell Areas..")
    startTime=time.time()
```

```
    area=np.zeros((imax+1,jmax+1))
    for j in range(2,jmax+1):
        for i in range(2,imax+1):
        # Calculate area by adding areas of the two triangles formin the cell
            area1 = calc_area_triangle(X[i-1,j-1],Y[i-1,j-1],X[i,j-1],
            Y[i,j-1],X[i,j],Y[i,j])
            area2 = calc_area_triangle(X[i,j],Y[i,j],X[i-1,j],Y[i-1,j],
            X[i-1,j-1],Y[i-1,j-1])
            area[i,j]=area1+area2
    totalTime = round(time.time() - startTime, 2)
    print('Completed in : %s seconds' % (totalTime))
    return area
######################## EDGE LENGTH calculator

def calc_edge_lengths(X,Y,imax,jmax):


    print("Calculating Cell Face Lengths...")
    startTime=time.time()
    length=np.zeros((imax+1,jmax+1,5))

    length[2:imax+1,2:jmax+1,1]=np.sqrt((X[1:imax,2:jmax+1]-X[1:imax,1:jmax])
    **2+(Y[1:imax,2:jmax    +1]-Y[1:imax,1:jmax])**2)
    length[2:imax+1,2:jmax+1,2]=np.sqrt((X[2:imax+1,1:jmax]-X[1:imax,1:jmax])
    **2+(Y[2:imax    +1,1:jmax]-Y[1:imax,1:jmax])**2)
    length[2:imax+1,2:jmax+1,3]=np.sqrt((X[2:imax+1,1:jmax]-
    X[2:imax+1,2:jmax+1])**2+(Y[2:imax    +1,1:jmax]-Y[2:imax+1,2:jmax+1])**2)
    length[2:imax+1,2:jmax+1,4]=np.sqrt((X[1:imax,2:jmax+1]-
    X[2:imax+1,2:jmax+1])**2+(Y    [1:imax,2:jmax+1]-Y[2:imax+1,2:jmax+1])**2)
    totalTime = round(time.time() - startTime, 2)
    print('Completed in : %s seconds' % (totalTime))
    return length



############## Normal Calculation
def calc_normals(X,Y,imax,jmax):



    print("Calculating Cell Face Normals")
    startTime=time.time()
    xnormals=np.zeros((imax+1,jmax+1,3))
    ynormals=np.zeros((imax+1,jmax+1,3))
    unitxnormals=np.zeros((imax+1,jmax+1,3))
    unitynormals=np.zeros((imax+1,jmax+1,3))
    for j in range(2,jmax+1):
```

```
        for i in range(2,imax+1):
            xnormals[i,j,1]=(Y[i-1,j-1]-Y[i-1,j])
            ynormals[i,j,1]=-1*(X[i-1,j-1]-X[i-1,j])
            unitxnormals[i,j,1]=xnormals[i,j,1]/(np.sqrt(xnormals[i,j,1]
            **2+ynormals[i,j,1]**2))
            unitynormals[i,j,1]=ynormals[i,j,1]/(np.sqrt(xnormals[i,j,1]
            **2+ynormals[i,j,1]**2))
            xnormals[i,j,2]=(Y[i,j-1]-Y[i-1,j-1])
            ynormals[i,j,2]=-1*(X[i,j-1]-X[i-1,j-1])
            unitxnormals[i,j,2]=xnormals[i,j,2]/(np.sqrt(xnormals[i,j,2]
            **2+ynormals[i,j,2]**2))
            unitynormals[i,j,2]=ynormals[i,j,2]/(np.sqrt(xnormals[i,j,2]

            **2+ynormals[i,j,2]**2))
    totalTime = round(time.time() - startTime, 2)
    print('Completed in : %s seconds' % (totalTime))
    return unitxnormals, unitynormals


##################

def calc_cell_normals(i,j,k,xnorm,ynorm,imax,jmax):


    if i<imax and j<jmax:
        if k==1:
            return xnorm[i,j,1],ynorm[i,j,1]
        if k==2:
            return xnorm[i,j,2],ynorm[i,j,2]
        if k==3:
            return -1*xnorm[i+1,j,1], -1*ynorm[i+1,j,1]
        if k==4:
            return -1*xnorm[i,j+1,2], -1*ynorm[i,j+1,2]
    if i==imax:
        if k==2:
            return xnorm[i,j,2],ynorm[i,j,2]
        if k==1:
            return xnorm[i,j,1],ynorm[i,j,1]
    if j==jmax:
        if k==1:
            return xnorm[i,j,1],ynorm[i,j,1]
        if k==2:
            return xnorm[i,j,2],ynorm[i,j,2]

######### INITILISE VECTORES
def initialize(imax,jmax):
```

```
    print("Intializing solution vector arrays...")
    startTime=time.time()
    p=np.zeros((imax+1,jmax+1))
    u=np.zeros((imax+1,jmax+1))
    v=np.zeros((imax+1,jmax+1))
    totalTime = round(time.time() - startTime, 2)
    print('Completed in : %s seconds' % (totalTime))
    return p,u,v
####### SET BOUNDARY CONDITIONS
def set_inlet_exit_boundary_conditions(p,u,v,imax,jmax,uinlet,vinlet,pexit):
    T
    u[2,2:jmax+1]=uinlet
    v[2,2:jmax+1]=vinlet
    p[imax,2:jmax+1]=pexit
    return p,u,v

######### INITIAL GUESS
def set_initial_guess(p,u,v,imax,jmax,uguess,vguess,pguess):

    u[3:imax,3:jmax]=uguess
    v[3:imax,3:jmax]=vguess
    p[3:imax,3:jmax]=pguess
    return p,u,v

# SET GHOST CELL values
def set_ghost_cell_values(p,u,v,xnormal,ynormal,imax,jmax):
    #
    p[2,3:jmax]=p[3,3:jmax]
    u[imax,3:jmax]=u[imax-1,3:jmax]
    v[imax,3:jmax]=v[imax-1,3:jmax]
    #for i in range(3,imax):
    p[3:imax,2]=p[3:imax,3]
    p[3:imax,jmax]=p[3:imax,jmax-1]
    return p,u,v

############## FLUX CALCULATION

def calc_fluxes(rho,p,u,v,beta,imax,jmax,length,xnormal,ynormal):
    #choice of CFL value
    C = 1.
    #C = 0.5
    U1 = np.zeros((imax+1,jmax+1))
    U2 = np.zeros((imax+1,jmax+1))
    c = np.zeros((imax+1,jmax+1))
    U_face = np.zeros((imax+1,jmax+1))
    UPOS = np.zeros((imax+1,jmax+1))
```

```
UNEG = np.zeros((imax+1,jmax+1))
temp1 = np.zeros((imax+1,jmax+1))
temp2 = np.zeros((imax+1,jmax+1))
temp3 = np.zeros((imax+1,jmax+1))
flux1 = np.zeros((imax+1,jmax+1,3))
flux2 = np.zeros((imax+1,jmax+1,3))
flux3 = np.zeros((imax+1,jmax+1,3))

# notation used in class.
xnormal2=xnormal
ynormal2=ynormal
xnormal3 = (xnormal2)
ynormal3 = (ynormal2)



U1[3:imax,3:jmax] = u[2:imax-1,3:jmax]*-1*xnormal3[3:imax,3:jmax,1]
+ v[2:imax-1,3:jmax]*-1*ynormal3[3:imax,3:jmax,1] #U(i)
U2[3:imax,3:jmax] = u[3:imax,3:jmax] *-1*xnormal3[3:imax,3:jmax,1]
+ v[3:imax,3:jmax]*-1*ynormal3[3:imax,3:jmax,1] #U(i+1)

U_face[3:imax,3:jmax] = 0.5*(U1[3:imax,3:jmax]+U2[3:imax,3:jmax])
c[3:imax,3:jmax] = 2*C/(abs(U_face[3:imax,3:jmax])+
pow((U_face[3:imax,3:jmax]**2+4*beta**2),0.5))

UPOS[3:imax,3:jmax] = np.vectorize(lambda x: x if x>0 else 0.)(U1[3:imax,3:jmax])
UNEG[3:imax,3:jmax] = np.vectorize(lambda x: x if x<0 else 0.)(U2[3:imax,3:jmax])

temp1[3:imax,3:jmax] = rho * UPOS[3:imax,3:jmax] + 0.5*c[3:imax,3:jmax]*
(p[2:imax-1,3:jmax]-p[3:imax,3:jmax])
temp2[3:imax,3:jmax] = rho * UNEG[3:imax,3:jmax] + 0.5*c[3:imax,3:jmax]*
(p[2:imax-1,3:jmax]-p[3:imax,3:jmax])
temp3[3:imax,3:jmax] = 0.5*(p[2:imax-1,3:jmax]+p[3:imax,3:jmax])
flux1[3:imax,3:jmax,1] = length[3:imax,3:jmax,1] * (temp1[3:imax,3:jmax]*
1+temp2[3:imax,3:jmax]*1)
flux2[3:imax,3:jmax,1] = length[3:imax,3:jmax,1] * (temp1[3:imax,3:jmax]*
u[2:imax-1,3:jmax]+temp2[3:imax,3:jmax]
*u[3:imax,3:jmax]+temp3[3:imax,3:jmax]*-1*xnormal3[3:imax,3:jmax,1])
flux3[3:imax,3:jmax,1] = length[3:imax,3:jmax,1] * (temp1[3:imax,3:jmax]*
v[2:imax-1,3:jmax]+temp2[3:imax,3:jmax]
*v[3:imax,3:jmax]+temp3[3:imax,3:jmax]*-1*ynormal3[3:imax,3:jmax,1])


# k=2 is the G FLux as mentioned in class.
k=2
U1[3:imax,3:jmax] = u[3:imax,2:jmax-1]*-1*xnormal3[3:imax,3:jmax,2]
```

```
+
v[3:imax,2:jmax-1]*-1*ynormal3[3:imax,3:jmax,2]
U2[3:imax,3:jmax] = u[3:imax,3:jmax]*-1*xnormal3[3:imax,3:jmax,2] +
v[3:imax,3:jmax]*-1*ynormal3[3:imax,3:jmax,2]

U_face[3:imax,3:jmax] = 0.5*(U1[3:imax,3:jmax]+U2[3:imax,3:jmax])
c[3:imax,3:jmax] = 2*C/(abs(U_face[3:imax,3:jmax])+

pow((U_face[3:imax,3:jmax]**2+4*beta**2),0.5))

#UPOS[3:imax,3:jmax] = np.vectorize(lambda x: x if x>0 else
0.)(U1[3:imax,3:jmax])
UPOS[3:imax,3:jmax] = np.vectorize(lambda x: x if x>0 else
0.)(U1[3:imax,3:jmax])
UNEG[3:imax,3:jmax] = np.vectorize(lambda x: x if x<0 else
0.)(U2[3:imax,3:jmax])

temp1[3:imax,3:jmax] = rho * UPOS[3:imax,3:jmax] +
0.5*c[3:imax,3:jmax]*(p[3:imax,2:jmax-1]-p[3:imax,3:jmax])
temp2[3:imax,3:jmax] = rho * UNEG[3:imax,3:jmax] +
0.5*c[3:imax,3:jmax]*(p[3:imax,2:jmax-1]-p[3:imax,3:jmax])
temp3[3:imax,3:jmax] = 0.5*(p[3:imax,2:jmax-1]+p[3:imax,3:jmax])
flux1[3:imax,3:jmax,2] = length[3:imax,3:jmax,2] *
(temp1[3:imax,3:jmax]*1+temp2[3:imax,3:jmax]*1)
flux2[3:imax,3:jmax,2] = length[3:imax,3:jmax,2]*
(temp1[3:imax,3:jmax]*u[3:imax,2:jmax-1]
+temp2[3:imax,3:jmax]*u[3:imax,3:jmax]+temp3[3:imax,3:jmax]
*-1
*xnormal3[3:imax,3:jmax,2])
flux3[3:imax,3:jmax,2] = length[3:imax,3:jmax,2] *
(temp1[3:imax,3:jmax]*v[3:imax,2:jmax-1]
+temp2[3:imax,3:jmax]*v[3:imax,3:jmax]+temp3[3:imax,3:jmax]
*-1
*ynormal3[3:imax,3:jmax,2])


for j in range(3,jmax):

    k=1
    #xnormal1,ynormal1 = calc_cell_normals(i-1,j,3,xnormal,ynormal
    ,imax,jmax)
    xnormal2,ynormal2 = calc_cell_normals(imax,j,1,xnormal,ynormal
    ,imax,jmax)
    xnormal3 = -1*(xnormal2)
    ynormal3 = -1*(ynormal2)
    U1 = u[imax-1,j]*xnormal3 + v[imax-1,j]*ynormal3 #U(i)
    U2 = u[imax,j] *xnormal3 + v[imax,j] *ynormal3 #U(i+1)
```

```
        U_face = 0.5*(U1+U2)
        c = 2*C/(abs(U_face)+pow((U_face**2+4*beta**2),0.5))
        UPOS = max(0,U1)
        UNEG = min(0,U2)

        temp1 = rho * UPOS + 0.5*c*(p[imax-1,j]-p[imax,j])
        temp2 = rho * UNEG + 0.5*c*(p[imax-1,j]-p[imax,j])
        temp3 = 0.5*(p[imax-1,j]+p[imax,j])
        flux1[imax,j,k] = length[imax,j,k] * (temp1*1+temp2*1)
        flux2[imax,j,k] = length[imax,j,k] *
        (temp1*u[imax-1,j]+temp2*u[imax,j]+temp3*xnormal3)
        flux3[imax,j,k] = length[imax,j,k] *
        (temp1*v[imax-1,j]+temp2*v[imax,j]+temp3*ynormal3)



    for i in range(3,imax):
        xnormal2,ynormal2 = calc_cell_normals(i,3,2,xnormal,ynormal,imax,jmax)
        xnormal3 = -1*(xnormal2)
        ynormal3 = -1*(ynormal2)
        flux1[i,3,2]=0
        flux2[i,3,2]=length[i,3,2] * ((0.5*(p[i,2]+p[i,3]))*xnormal3)
        flux3[i,3,2]=length[i,3,2] * ((0.5*(p[i,2]+p[i,3]))*ynormal3)
        xnormal2,ynormal2 = calc_cell_normals(i,jmax,2,xnormal,
        ynormal,imax,jmax)
        xnormal3 = -1*(xnormal2)
        ynormal3 = -1*(ynormal2)
        flux1[i,jmax,2]=0
        flux2[i,jmax,2]=length[i,jmax,2] * ((0.5*(p[i,jmax-1]+p[i,jmax]))*xnormal3)
        flux3[i,jmax,2]=length[i,jmax,2] * ((0.5*(p[i,jmax-1]+p[i,jmax]))*ynormal3)
    return flux1,flux2,flux3
 ######### get the flux
def get_fluxes(i,j,k,flux1,flux2,flux3,imax,jmax):
    if i<imax and j<jmax:
        if k==1:
            return flux1[i,j,1],flux2[i,j,1],flux3[i,j,1]
        if k==2:
            return flux1[i,j,2],flux2[i,j,2],flux3[i,j,2]
        if k==3:
            return flux1[i+1,j,1],flux2[i+1,j,1],flux3[i+1,j,1]
        if k==4:
            return flux1[i,j+1,2],flux2[i,j+1,2],flux3[i,j+1,2]
    if i==imax:
        if k==1:
            return flux1[i,j,1],flux2[i,j,1], flux3[i,j,1]
    if j==jmax:
        if k==2:
```

```
            return flux1[i,j,2],flux2[i,j,2], flux3[i,j,2]


######## residues
def calc_residues(flux1,flux2,flux3,imax,jmax):
    res1 = np.zeros((imax+1,jmax+1))
    res2 = np.zeros((imax+1,jmax+1))
    res3 = np.zeros((imax+1,jmax+1))
    resF1a, resF2a, resF3a = flux1[3:imax,3:jmax,1],flux2[3:imax,3:jmax,1],
    flux3[3:imax,3:jmax,1]
    resF1b, resF2b, resF3b = flux1[4:imax+1,3:jmax,1],
    flux2[4:imax+1,3:jmax,1],
    flux3[4:imax
    +1,3:jmax,1]
    resF1 = resF1b-resF1a


    resF2 = resF2b-resF2a
    resF3 = resF3b-resF3a
    resG1a, resG2a, resG3a =flux1[3:imax,3:jmax,2],flux2[3:imax,3:jmax,2],
    flux3[3:imax,3:jmax,2]
    resG1b, resG2b, resG3b =flux1[3:imax,4:jmax+1,2],flux2[3:imax,4:jmax+1,2],
    flux3[3:imax,4:jmax    +1,2]
    resG1 = resG1b-resG1a
    resG2 = resG2b-resG2a
    resG3 = resG3b-resG3a
    res1[3:imax,3:jmax]=resF1+resG1
    res2[3:imax,3:jmax]=resF2+resG2
    res3[3:imax,3:jmax]=resF3+resG3
    return res1, res2, res3
```

A.4 Defined Functions (modules of solver): Post-Processing

```
######### CREATE PLOT FIGURE

def create_figure():
    plt.figure()
    fig, axs = plt.subplots(1,2)
    return fig,axs
### inlet redisue
def calc_residual(r1,r2,r3,rho,uinlet,imax,jmax):
    residual = 0
    for j in range(3,jmax):
        for i in range(3,imax):
            residual = residual + ((r1[i,j]/(rho*uinlet))**2 +
            (r2[i,j]/    (rho*uinlet**2))**2 + (r3[i,j]/(rho*uinlet**2))**2)
```

```
    return np.sqrt(residual)

def march_one_step(rho,p_n,u_n,v_n,beta,area,res1,res2,res3,imax
,jmax,uinlet,vinlet,pexit):
    p_new=np.zeros((imax+1,jmax+1))
    u_new=np.zeros((imax+1,jmax+1))
    v_new=np.zeros((imax+1,jmax+1))
    p_new, u_new, v_new = set_inlet_exit_boundary_conditions(p_new,u_new,v_new,imax,
    jmax,uinlet,vinlet,pexit)
    dt = 0.0001 ## nominal value, need to put in the equation
    u_new[3:imax,3:jmax] = u_n[3:imax,3:jmax] -
    dt/area[3:imax,3:jmax]*((-u_n[3:imax,3:jmax]/rho)*res1[3:imax,3:jmax]

    + res2[3:imax,3:jmax]/rho)
    v_new[3:imax,3:jmax] = v_n[3:imax,3:jmax] -
    dt/area[3:imax,3:jmax]*((-v_n[3:imax,3:jmax]/rho)*res1[3:imax,3:jmax] +

    res3[3:imax,3:jmax]/rho)
    p_new[3:imax,3:jmax] = p_n[3:imax,3:jmax] -
    dt/area[3:imax,3:jmax]*(beta**2*res1[3:imax,3:jmax])


    p,u,v = set_ghost_cell_values(p_new,u_new,v_new,xnormal,ynormal,imax,jmax)
    flux1,flux2,flux3 =
    calc_fluxes(rho,p,u,v,beta,imax,jmax,length,xnormal,ynormal)
    r1,r2,r3 = calc_residues(flux1,flux2,flux3,imax,jmax)
    residual = calc_residual(r1,r2,r3,rho, uinlet,maxi,maxj)
    print(residual)
    return p,u,v,r1,r2,r3,residual
########## save data
def save_data(p,u,v,r1,r2,r3,residual):
    np.savetxt('p.txt',p)
    np.savetxt('u.txt',u)
    np.savetxt('v.txt',v)
    np.savetxt('r1.txt',r1)
    np.savetxt('r2.txt',r2)
    np.savetxt('r3.txt',r3)
    np.savetxt('residual.txt',residual)

################## Load Data
def load_data():
    p=np.loadtxt('p.txt')
    u=np.loadtxt('u.txt')
    v=np.loadtxt('v.txt')
    r1=np.loadtxt('r1.txt')
    r2=np.loadtxt('r2.txt')
    r3=np.loadtxt('r3.txt')
```

```
    residual=np.loadtxt('residual.txt')
    return p,u,v,r1,r2,r3,residual
####### puv plot
def puvplots(imax,jmax,X1,Y1,p,u,v):
    fig, axs = plt.subplots(1,3)
    a = np.arange(3,imax)
    b = np.arange(3,jmax)
    X, Y = np.meshgrid(a,b)
    levels_u = np.linspace(5, +15, 400)
    levels_v = np.linspace(-5, +5, 400)
    levels_p = np.linspace(+99900, +100050, 400)
    udata =u[X,Y]
    vdata =v[X,Y]
    pdata =p[X,Y]
    cs = axs[0].contourf(X1[X,Y], Y1[X,Y], udata, levels=levels_u)
    fig.colorbar(cs, ax=axs[0], format="%.2f")
    cs = axs[1].contourf(X1[X,Y], Y1[X,Y], vdata, levels=levels_v)
    plt.title('Velocity U (X Component) Contours')
    fig.colorbar(cs, ax=axs[1], format="%.2f")
    cs = axs[2].contourf(X1[X,Y], Y1[X,Y], pdata, levels=levels_p)
    plt.title('Velocity V (Y Component) Contours')
    fig.colorbar(cs, ax=axs[2])
    plt.title('Pressure P Contours')
    plt.show()


def puvlineplots(imax,jmax,X1,Y1,p,u,v):
    a = np.arange(3,imax)
    b = np.arange(3,jmax)
    X, Y = np.meshgrid(a,b)
    levels_u = np.linspace(5, +15, 400)
    levels_v = np.linspace(-5, +5, 400)
    levels_p = np.linspace(+99900, +100050, 400)
    udata =u[X,Y]
    vdata =v[X,Y]
    pdata =p[X,Y]
    plt.figure()
    CS = plt.contour(X1[X,Y], Y1[X,Y], udata)
    plt.clabel(CS, inline=1, fontsize=10)
    plt.title('Velocity U (X Component) Contours')
    plt.show()
    plt.figure()
    CS = plt.contour(X1[X,Y], Y1[X,Y], vdata)
    plt.clabel(CS, inline=1, fontsize=10)
    plt.title('Velocity V (Y Component) Contours')
    plt.show()
    plt.figure()
```

```
    CS = plt.contour(X1[X,Y], Y1[X,Y], pdata)
    plt.clabel(CS, inline=1, fontsize=10)
    plt.title('Pressure P Contours')
    plt.show()

def quiverplot(X1,Y1,U,V,imax,jmax):
    plt.figure()
    a = np.arange(3,imax)
    b = np.arange(3,jmax)
    X, Y = np.meshgrid(a,b)
    udata =U[X,Y]
    vdata =V[X,Y]
    plt.plot(X1[1:,1:],Y1[1:,1:],'yo')
    plt.quiver(X1[X,Y],Y1[X,Y], 0.0003*U[X,Y], 0.0003*V[X,Y],
    scale = 0.01,units = 'inches')
    plt.show()

def check_convergence(residual0, residual,tolerance):
    if residual/residual0 < tolerance:
        return True
    else:
        return False

def time_march(rho,p,u,v,beta,area,r1,r2,r3,imax,jmax,n,X,Y,
uinlet,vinlet,pexit,residual0):
    residual = np.zeros(n+1)
    i=1
    #for i in range(1,n+1):
    while(i<n+1):
        print("Time March Number :%s"%(i))
        p, u, v , r1, r2, r3, residual[i] = march_one_step(rho,p,u,v,beta,
        area,r1,r2,r3,maxi,maxj,
        uinlet,vinlet,pexit)
        i+=1
    save_data(p,u,v,r1,r2,r3,residual)
    puvplots(imax,jmax,X,Y,p,u,v)
    puvlineplots(imax,jmax,X,Y,p,u,v)
    quiverplot(X,Y,u,v,imax,jmax)
    plot_residual_ratio(n,residual,residual0)
    return p,u,v,r1,r2,r3,residual,n
def plot_residual_ratio(n,residual,residual0):
    #makes the residual convergence plot
    residual_ratio = residual/residual0
    plt.figure()
    plt.loglog(range(1,n+1),residual_ratio[1:n+1],'yo')
    plt.xlabel('iterations')
    plt.ylabel('residual ratio')
```

```
    plt.title('Convergence plot in log-log scale')
    plt.grid(True)
    plt.show()



##############MAIN CODE
string='E:/Whatsapp data/whatsapp data/computational aerodynamics/bumpgrid.dat'

print("-----------------------------------------------------------")
startTime = time.time()
X1,Y1, maxi,maxj=read_mesh(string)
area = calc_cell_areas(X1,Y1,maxi,maxj)
length = calc_edge_lengths(X1,Y1,maxi,maxj)
xnormal,ynormal = calc_normals(X1,Y1,maxi,maxj)
p,u,v = initialize(maxi,maxj)
p,u,v = set_inlet_exit_boundary_conditions(p,u,v,maxi,maxj,10.0,0.0,100000.0)
p,u,v = set_initial_guess(p,u,v,maxi,maxj,8.0,0.0,100000.0)
p,u,v = set_ghost_cell_values(p,u,v, xnormal, ynormal, maxi, maxj)
flux1=np.zeros((maxi+1,maxj+1,3))


flux2=np.zeros((maxi+1,maxj+1,3))
flux3=np.zeros((maxi+1,maxj+1,3))
flux1,flux2,flux3 = calc_fluxes(1.0,p,u,v,10.0,maxi,maxj,length,xnormal,ynormal)
r1,r2,r3 = calc_residues(flux1,flux2,flux3,maxi,maxj)
residual0 = calc_residual(r1,r2,r3,1.00, 10.00,maxi,maxj)

p, u, v , r1, r2, r3,residual = march_one_step(1.00,p,u,v,10.0,area,r1,r2,r3,
maxi,maxj,10.0,0.0,100000.0)
n=60000

p,u,v,res1,res2,res3,residual,n = time_march(1.00,p,u,v,10.0,area,r1,r2,r3,
maxi,maxj,n,X1,Y1,10.0,
0.0,100000.0,residual0)

"""
The time taken to run the program for n=50,000:
Time march processing time: 0.026 sec (avg time roughly)

Whole Processing time: 21.667 min
Process rate: 38.46 Iterations per sec

Memory Usage: 66%
"""
```