Solving the incompressible Euler equations in 2D using a finite-volume method with the artificial compressibility approach

Shraman Maiti[1]

Abstract

A cell-centered finite volume method (FVM) has been used with the artificial compressibility approach to simulate an inviscid, incompressible flow in a channel with a bump. The artificial compressibility method, used here, simplifies the calculation of fluxes, leading to better convergence of results. For the flux calculations, an upwinding scheme has been used, added with extra dissipation based on the pressure difference among the neighboring cells. Additionally the performance of the dissipation has been tested by varying the associated co-efficient.

Keywords: Channel flow, Artificial compressibility, Finite-volume method.

1 Introduction

The cell-centered finite volume method (FVM) has been a widely used and well-validated method for computationally solving flow equations over various domains (both, internal and external). One of the main keypoints that differentiate FVMs from finite-difference methods (FDM) is the requirement to solve the equations in their conservative form, rather than a non-conservative differential form. Therefore, traditionally FVMs have had a better capability in handling a wide range of flows as singularities in the equations can be handled better if they are in their conservative form. But major additional steps that come up in FVMs are the handling of numerical fluxes as cell-centered FVMs solve average values within every cell and leads to numerical discontinuities at every interface. The information from one cell to the next call has to travel in the form of fluxes through their included interface. Their have been several approaches to formulating these fluxes. For doing so, the practice of solving a Riemann problem is widely done at each interface. The Godunov method gives the exact solution to the Riemann problem but runs into various problems while handling non-linear problems such as the Euler equations. For such problems, approximate solutions to the Riemann problem are used. Furthermore, it is a common practice to formulate the fluxes at the interface as some function of the neighboring cell states.

Additionally, using an artificial compressibility[2] approach makes the the process of solving the equations in a compact form much simpler. Here, compact form means handling all the variables being solved for as a single solution vector as in incompressible flows, we technically solve the pressure difference and not the exact pressure values. This technique helps us to solve for all the flow variables simultaneously, speeding up the entire process. But such an approach is only possible for solving steady flows, or taking a time-dependent problem to its steady solution.

Here, 2D Euler equations have been solved by FVM using the artificial compressibility approach for a flow within a channel and over a bump. For the flux calculations an advection based upwinding scheme (based on pressure dissipation) has been used which gives the mass flux as the function of the neighboring cell states. The performance of the dissipation has also been tested by varying the coefficient of this dissipation.

The following section, Computational Methodology, discusses the set-up of the problem, followed by the discretization method and the flux scheme used. Following this section, the Results section compares the solutions plots for the different dissipation values and gives an insight to the

1. MS Scholar, Department of Aerospace Engineering, IIT Madras, India, ae25s006@smail.ac.in
2. M. M. Rahman and T. Siikonen, "An Artificial Compressibility Method for Incompressible Flows," *Numerical Heat Transfer*, November 30, 2010, https://doi.org/10.1080/104077901753243188.

numerical scheme used. Finally, in the Conclusion section, we understand the effectiveness of using the scheme and the effect of dissipation.

2 Computational Methodology

2.1 Problem Solved

The 2D incompressible Euler equations are given as follows:

$$\boldsymbol{M}\boldsymbol{q_t} + \boldsymbol{A}\boldsymbol{q_x} + \boldsymbol{B}\boldsymbol{q_y} = \boldsymbol{0} \tag{1}$$

where,

$$\boldsymbol{q} = \begin{bmatrix} p \\ u \\ v \end{bmatrix}, \quad \boldsymbol{M} = \begin{bmatrix} 1/\beta^2 & 0 & 0 \\ 0 & \rho & 0 \\ 0 & 0 & \rho \end{bmatrix}, \quad \boldsymbol{A} = \begin{bmatrix} 0 & \rho & 0 \\ 1 & \rho u & 0 \\ 0 & 0 & \rho u \end{bmatrix}, \quad \boldsymbol{B} = \begin{bmatrix} 0 & 0 & \rho \\ 0 & \rho v & 0 \\ 1 & 0 & \rho v \end{bmatrix}$$

and $x_y = \frac{\partial x}{\partial y}$

The conservative form is given by:

$$\boldsymbol{M}\boldsymbol{q_t} + \nabla.\boldsymbol{f} = 0 \tag{2}$$

where,

$$\boldsymbol{f} = \begin{bmatrix} \rho \bar{v} \\ \rho u \bar{v} + p\delta_{i1} \\ \rho v \bar{v} + p\delta_{i2} \end{bmatrix}$$

Taking a volume integral of the equation and converting the volume integral of $\nabla \boldsymbol{f}$ to a surface integral using the Gauss-divergence theorem, we are ready to discretize the equations.

2.2 Discretization

The discretized form for solving using FVM was as follows:

$$\boldsymbol{M}\frac{\bar{q}_{i,j}^{n+1} - \bar{q}_{i,j}^{n}}{\Delta t} + \frac{\sum_k \tilde{F}_k}{\Delta V} = 0 \tag{3}$$

where, $\tilde{F}_k = A_k \bar{F}_k$ in which, $\bar{F}_k$ is the flux vector of each face, qualitatively similar to the $\bar{f}$ defined above and A_k is the corresponding face area.

The flux at an interface, say $i + \frac{1}{2}, j$ that lies between the cells i, j and $i + 1, j$ is given by: (second index dropped for ease of writing)

$$\tilde{F}_{i+\frac{1}{2},j} = A_{i+\frac{1}{2},j}(\rho u^{+}_{i+\frac{1}{2}}[1, u, v]^T_i + \rho u^{-}_{i+\frac{1}{2}}[1, u, v]^T_{i+1} + p[0, n_x, n_y]^T) \tag{4}$$

2.3 Flux Scheme

The first two terms inside the bracket on the RHS of equation 4 gives the convective flux whereas the third one gives the pressure flux. The convective flux taken for our case is as follows:

$$\dot{m}^{+}_{i+\frac{1}{2}} = \left(\rho U|^{+}_{i} + \alpha \frac{p_i - p_{i+1}}{2\lambda_{\max}}\right) A_{i+\frac{1}{2}}, \tag{5}$$

$$\dot{m}^{-}_{i+\frac{1}{2}} = \left(\rho U|^{-}_{i+1} + \alpha \frac{p_i - p_{i+1}}{2\lambda_{\max}}\right) A_{i+\frac{1}{2}}, \tag{6}$$

$$\dot{m}_{i+\frac{1}{2}} = \dot{m}^{+}_{i+\frac{1}{2}} + \dot{m}^{-}_{i+\frac{1}{2}}. \tag{7}$$

where,

$$U|_i = \vec{v}_i \cdot \hat{n}_{i+\frac{1}{2}}, \tag{8}$$

$$U|_{i+1} = \vec{v}_{i+1} \cdot \hat{n}_{i+\frac{1}{2}}. \tag{9}$$

where, α is the dissipation coefficient. For our cases $\alpha = 0.5$ and 1 have been taken.

2.4 Mesh

A structured grid as shown in figure 1 has been used for the computation.

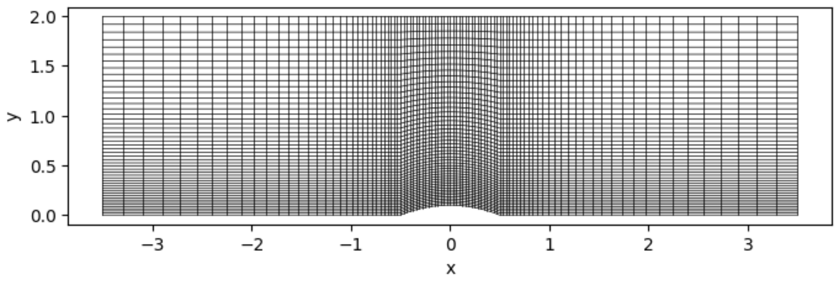

Figure 1: Mesh with 96 x 48 cells

2.5 Boundary conditions

The boundary conditions on the inlet had freestream x-velocity of 20 m/s and y-velocity of 0 m/s and had a static guage pressure of 0 on the inlet. Both the top and bottom faces have been treated as slip walls.

3 Results

The various plots and graphs have been compared and inferred from in this section.

3.1 Flow variables

The Mach number and pressure plots for the dissipation coefficient of 1 are shown in figures 2 and 3:

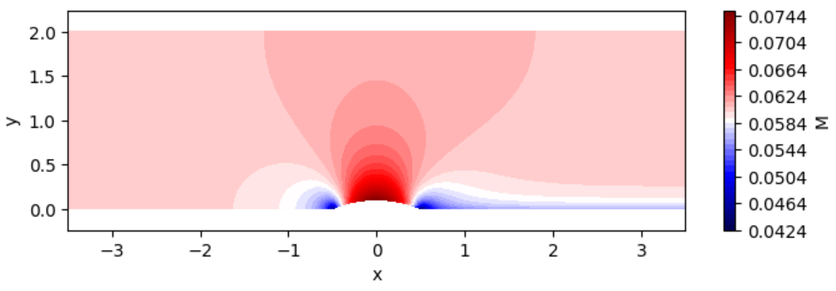

Figure 2: Mach number contour at a dissipation coefficient of 1

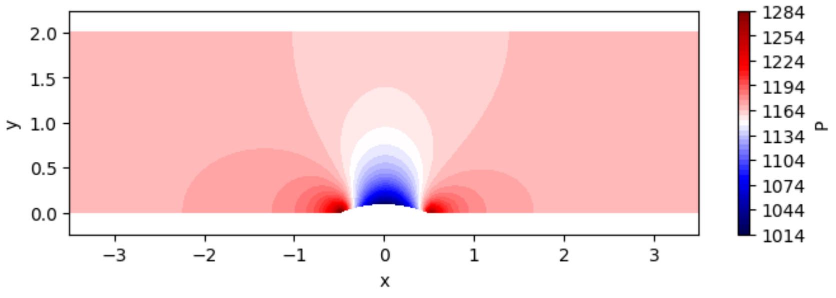

Figure 3: Pressure contour at a dissipation coefficient of 1

The Mach number and pressure plots for the dissipation coefficient of 0.5 are shown in figures 4 and 5:

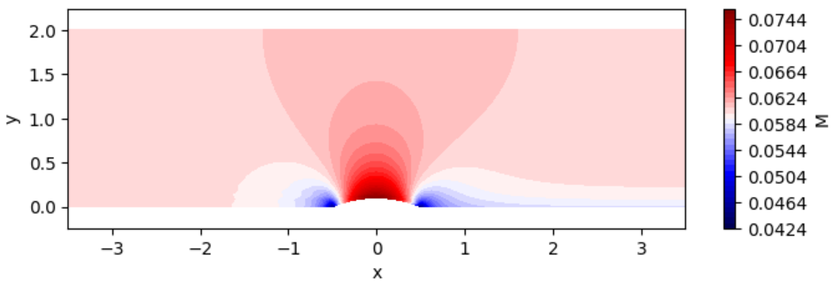

Figure 4: Mach number contour at a dissipation coefficient of 0.5

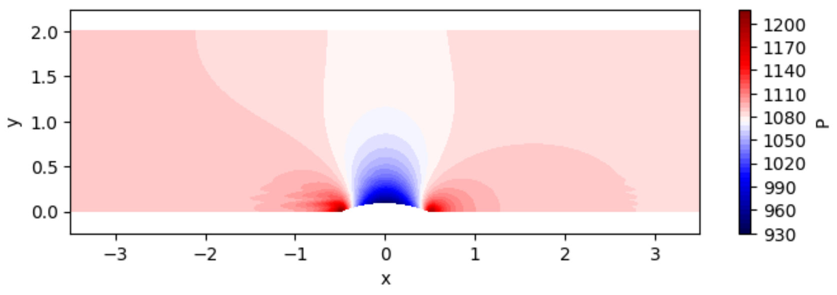

Figure 5: Pressure contour at a dissipation coefficient of 0.5

It can be clearly seen in the plots that for $\alpha = 0.5$ the solution has significant errors whereas for $\alpha = 1$ the thee features are distinctively visible, But in the following subsection we will see that the residuals were not able to converge further for $\alpha = 0.5$ implying that a higher dissipation is more suitable to get a better converged solution.

Other features like the stagnation zones and the increase and decrease of velocity and pressure respectively over the bump are clearly visible.

3.2 Residual convergence

The residual convergence plots for the 2 cases have been compared in figure 6:

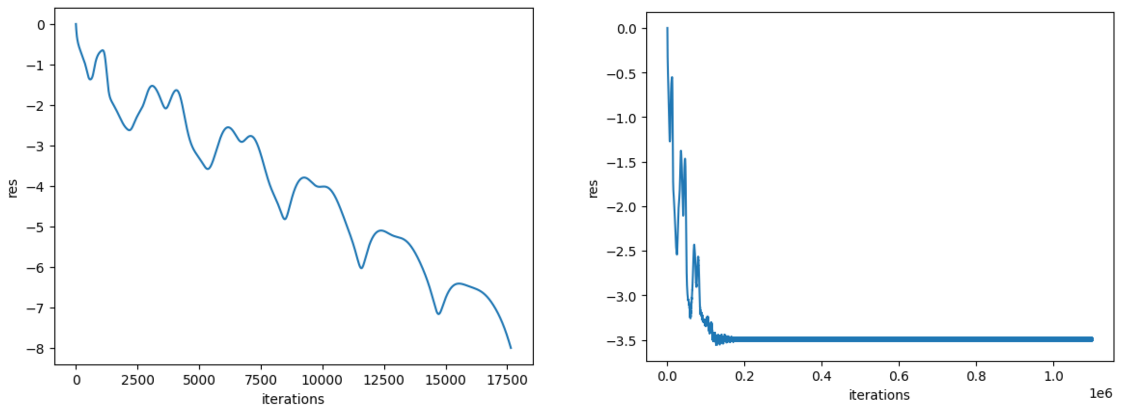

Figure 6: The residual convergence plots for $\alpha = 1$ (left) and $\alpha = 0.5$ (right)

As mentioned earlier, the residuals flatten at a larger value for $\alpha = 0.5$ whereas for $\alpha = 1$ the convergence is excellent. A higher dissipation clearly helps in faster and better convergence.

4 Conclusion

The finite volume method with artificial compressibility gives conclusive results to prove its capability is solving such problems. The contours show expected trends in Mach number and pressure values. Additionally, adding a pressure based dissipation greatly improves the stability of the scheme. With increasing value of the dissipation, it is able to converge to the correct solution faster.

Appendix

The code snippets have been added here for the readers' reference.

Listing 1: Initialization

```python
import numpy as np
import matplotlib.pyplot as plt

resfile = 'solfiles/solution_iter_100000.npz'
file1 = 'meshfile.txt'

rho = 1.293
alpha = 1
beta = 20
u_inf = 20
v_inf = 0
p_stat = 0                          # Since guage pressure is 0
cfl = 0.8
tol = 1e-8
p_atm = 101326
rho = 1.293
gamma = 1.4
a_sound = np.sqrt(gamma*p_atm/rho)
restart = False
if restart:
    last_itr = 100000
    last_res = 0.004307725768801071
```

Listing 2: Meshing

```python
with open(file1, 'r') as meshfile:
    sortlines = []
    lines = meshfile.readlines()
    for line in lines:
        line = line.strip()
        strnum = line.split()
        fnum = [float(x) for x in strnum]
        sortlines.append(fnum)
dims = sortlines[0]
dim = [int(x) for x in dims]
dim = np.array(dim)

nodes = sortlines[1:-1]

cellsdef = []
for i in range(dim[0]*(dim[1]-1)):
    if (i + 1) % 97 == 0:
        continue
    cellsdef.append([i+1, i+1+dim[0], i+dim[0], i])

cells = np.zeros((dim[0]-1,dim[1]-1), dtype=int)
cellind = 0
for j in range(dim[1]-1):
    for i in range(dim[0]-1):
        cells[i,j] = int(cellind)
        cellind += 1
```

```

facenormals = []
for cell in cellsdef:
    nE = np.array(((nodes[cell[1]][1]-nodes[cell[0]][1]),-(nodes[cell
        [1]][0]-nodes[cell[0]][0])))
    nN = np.array(((nodes[cell[2]][1]-nodes[cell[1]][1]),-(nodes[cell
        [2]][0]-nodes[cell[1]][0])))
    nW = np.array(((nodes[cell[3]][1]-nodes[cell[2]][1]),-(nodes[cell
        [3]][0]-nodes[cell[2]][0])))
    nS = np.array(((nodes[cell[0]][1]-nodes[cell[3]][1]),-(nodes[cell
        [0]][0]-nodes[cell[3]][0])))
    facenormals.append([nE, nN, nW, nS])

faceareas = []
for cell in cellsdef:
    p0 = np.array(nodes[cell[0]])
    p1 = np.array(nodes[cell[1]])
    p2 = np.array(nodes[cell[2]])
    p3 = np.array(nodes[cell[3]])
    aE = np.linalg.norm(p1-p0)
    aN = np.linalg.norm(p2-p1)
    aW = np.linalg.norm(p3-p2)
    aS = np.linalg.norm(p0-p3)
    faceareas.append([aE, aN, aW, aS])

cellvols = []
for cell in cellsdef:
    p0 = np.array(nodes[cell[0]])
    p1 = np.array(nodes[cell[1]])
    p2 = np.array(nodes[cell[2]])
    p3 = np.array(nodes[cell[3]])
    a1 = 0.5 * np.abs(p0[0]*(p1[1] - p2[1]) + p1[0]*(p2[1] - p0[1]) +
       p2[0]*(p0[1] - p1[1]))
    a2 = 0.5 * np.abs(p0[0]*(p2[1] - p3[1]) + p2[0]*(p3[1] - p0[1]) +
       p3[0]*(p0[1] - p2[1]))
    v = a1+a2
    cellvols.append(v)

xfacesN = np.zeros((dim[0],dim[1]-1, 2))
for j in range(dim[1]-1):
    for i in range(dim[0]):
        if i == 0:
            xfacesN[i,j] = -facenormals[cells[i,j]][2]
            continue
        xfacesN[i,j] = facenormals[cells[i-1,j]][0]

yfacesN = np.zeros((dim[0]-1,dim[1], 2))
for i in range(dim[0]-1):
    for j in range(dim[1]):
        if j == 0:
            yfacesN[i,j] = -facenormals[cells[i,j]][3]
            continue
        yfacesN[i,j] = facenormals[cells[i,j-1]][1]

xfacesA = np.zeros((dim[0],dim[1]-1))
for j in range(dim[1]-1):
    for i in range(dim[0]):
        if i == 0:
            xfacesA[i,j] = faceareas[cells[i,j]][2]
            continue
        xfacesA[i,j] = faceareas[cells[i-1,j]][0]

yfacesA = np.zeros((dim[0]-1,dim[1]))
for i in range(dim[0]-1):
    for j in range(dim[1]):
        if j == 0:
            yfacesA[i,j] = faceareas[cells[i,j]][3]
            continue
        yfacesA[i,j] = faceareas[cells[i,j-1]][1]
```

Listing 3: Flux calculation function, boundary conditions have been imposed here itself

```
# Expressing the fluxes as functions of cell values
def fluxcalc(dim, p, u, v):

    # Fluxes initialization
    xFluxes = np.zeros((dim[0], dim[1]-1, 3))        # for SM: Same
        shape as xfacesA + a dimension with value 3 for the 3 eq's
    yFluxes = np.zeros((dim[0]-1, dim[1], 3))        # for SM: Same
        shape as yfacesA + a dimension with value 3 for the 3 eq's

    # X-face fluxes
    for j in range(dim[1]-1):
        for i in range(1, dim[0]-1):
            area = xfacesA[i,j]
            nx = xfacesN[i,j][0]/area
            ny = xfacesN[i,j][1]/area
            n = np.array([nx, ny])
            qL = np.array([1, u[i-1,j], v[i-1,j]])
            qR = np.array([1, u[i,j], v[i,j]])
            delP = p[i-1,j] - p[i,j]
            p_avg = (p[i-1,j] + p[i,j])/2
            u_avg = 0.5*(u[i-1,j] + u[i,j])
            v_avg = 0.5*(v[i-1,j] + v[i,j])
            v_vec = np.array([u_avg, v_avg])
            v_n = np.dot(v_vec, n)
            lambda_max = 0.5*(np.abs(v_n) + np.sqrt(v_n**2 + 4*beta**2)
                )
            vL = np.array([u[i-1,j], v[i-1,j]])
            vR = np.array([u[i,j], v[i,j]])
            vnL = np.dot(vL, n)
            vnR = np.dot(vR,n)

            press_dissipation = (alpha / (2 * lambda_max)) * delP
            c_flux = (rho * (max(0, vnL)) + press_dissipation) * qL + (
                rho*(min(0, vnR) + press_dissipation) * qR)
            p_flux = p_avg * np.array([0, nx, ny])

            xFluxes[i,j] = (c_flux + p_flux) * area

        # Left face
        area = xfacesA[0,j]
        nx = xfacesN[0,j][0]/area
        ny = xfacesN[0,j][1]/area
        n = np.array([nx, ny])
        qL = np.array([1, u_inf, v_inf])
```

```python
            qR = np.array([1, u[0,j], v[0,j]])
            delP = 0          # Inlet guage pressure is 0
            p_avg = p[0,j]
            u_avg = 0.5*(u_inf + u[0,j])
            v_avg = 0.5*(v_inf + v[0,j])
            v_vec = np.array([u_avg, v_avg])
            v_n = np.dot(v_vec, n)
            lambda_max = 0.5*(np.abs(v_n) + np.sqrt(v_n**2 + 4*beta**2))
            vL = np.array([u_inf, v_inf])
            vR = np.array([u[0,j], v[0,j]])
            vnL = np.dot(vL, n)
            vnR = np.dot(vR,n)

            press_dissipation = (alpha / (2 * lambda_max)) * delP
            c_flux = (rho * (max(0, vnL)) + press_dissipation) * qL + (rho
                *(min(0, vnR) + press_dissipation) * qR)

            p_flux = p_avg * np.array([0, nx, ny])
            xFluxes[0,j] = (c_flux + p_flux) * area

            # Right face
            area = xfacesA[dim[0]-1,j]
            nx = xfacesN[dim[0]-1,j][0]/area
            ny = xfacesN[dim[0]-1,j][1]/area
            n = np.array([nx, ny])
            qL = np.array([1, u[dim[0]-2,j], v[dim[0]-2,j]])
            qR = np.array([1, u[dim[0]-2,j], v[dim[0]-2,j]])
            delP = 0
            p_avg = p[dim[0]-2,j]
            u_avg = u[dim[0]-2,j]
            v_avg = v[dim[0]-2,j]
            v_vec = np.array([u_avg, v_avg])
            v_n = np.dot(v_vec, n)
            lambda_max = 0.5*(np.abs(v_n) + np.sqrt(v_n**2 + 4*beta**2))
            vL = np.array([u[dim[0]-2,j], v[dim[0]-2,j]])
            vR = np.array([u[dim[0]-2,j], v[dim[0]-2,j]])
            vnL = np.dot(vL, n)
            vnR = np.dot(vR,n)

            press_dissipation = (alpha / (2 * lambda_max)) * delP
            c_flux = (rho * (max(0, vnL)) + press_dissipation) * qL + (rho
                *(min(0, vnR) + press_dissipation) * qR)

            p_flux = p_avg * np.array([0, nx, ny])
            xFluxes[dim[0]-1,j] = (c_flux + p_flux) * area


        # Y-face fluxes
        for i in range(dim[0]-1):
            for j in range(1, dim[1]-1):
                area = yfacesA[i,j]
                nx = yfacesN[i,j][0]/area
                ny = yfacesN[i,j][1]/area
                n = np.array([nx, ny])
                qD = np.array([1, u[i,j-1], v[i,j-1]])
                qU = np.array([1, u[i,j], v[i,j]])
                delP = p[i,j-1] - p[i,j]
                p_avg = (p[i,j-1] + p[i,j])/2
                u_avg = 0.5*(u[i,j-1] + u[i,j])
                v_avg = 0.5*(v[i,j-1] + v[i,j])
                v_vec = np.array([u_avg, v_avg])
                v_n = np.dot(v_vec, n)
                lambda_max = 0.5*(np.abs(v_n) + np.sqrt(v_n**2 + 4*beta**2)
                    )
                vD = np.array([u[i,j-1], v[i,j-1]])
                vU = np.array([u[i,j], v[i,j]])
                vnD = np.dot(vD, n)
                vnU = np.dot(vU, n)

                press_dissipation = (alpha / (2 * lambda_max)) * delP
                c_flux = (rho * (max(0, vnD)) + press_dissipation) * qD + (
                    rho*(min(0, vnU) + press_dissipation) * qU)

                p_flux = p_avg * np.array([0, nx, ny])
                yFluxes[i,j] = (c_flux + p_flux) * area

            # Bottom face
            area = yfacesA[i,0]
            nx = yfacesN[i,0][0]/area
            ny = yfacesN[i,0][1]/area
            n = np.array([nx, ny])
            p_face = p[i,0]
            yFluxes[i,0] = p_face * np.array([0, nx, ny]) * area

            # Top face
            area = yfacesA[i,dim[1]-1]
            nx = yfacesN[i,dim[1]-1][0]/area
            ny = yfacesN[i,dim[1]-1][1]/area
            n = np.array([nx, ny])
            p_face = p[i,dim[1]-2]
            yFluxes[i,dim[1]-1] = p_face * np.array([0, nx, ny]) * area


    res = np.zeros((dim[0]-1,dim[1]-1, 3))
    for j in range(dim[1]-1):
        for i in range(dim[0]-1):
            res[i,j] = xFluxes[i+1, j] - xFluxes[i,j] + yFluxes[i,j+1]
                - yFluxes[i,j]

    return res
```

Listing 4: Initializing the solution variables and time step calculations

```
# Cell values initialization
if not restart:
    p = np.zeros((dim[0]+1,dim[1]+1))
    u = np.zeros((dim[0]+1,dim[1]+1))
    v = np.zeros((dim[0]+1,dim[1]+1))
else:
    res_data = np.load('solfiles/solution_iter_100000.npz')
    p = res_data['p']
    u = res_data['u']
    v = res_data['v']

# Time-step calaculation
def tcalc(dim, u, v, facenormals, faceareas, cellvols, cells):
```

```python
    t_step = np.zeros((dim[0]-1,dim[1]-1))
    for j in range(dim[1]-1):
        for i in range(dim[0]-1):
            uP = u[i,j]
            vP = v[i,j]
            velP = np.array([uP,vP])
            a = np.zeros(4)
            nW = np.zeros((4,2))
            vn = np.zeros(4)
            lambda_k = np.zeros(4)
            for k in range(4):
                a[k] = faceareas[cells[i,j]][k]
                nW[k] = np.array(facenormals[cells[i,j]][0])
                vn[k] = np.dot(velP, nW[k])
                lambda_k[k] = 0.5 * (np.abs(vn[k]) + np.sqrt(vn[k]**2 +
                    4*beta**2))
            t_step[i,j] = cellvols[cells[i,j]] * cfl / (np.dot(a,
                lambda_k))
    del_t = np.min(t_step)
    return np.abs(del_t)
```

Listing 5: Main loop

```python
# Main Loop
if not restart:
    itr = 0
else:
    itr = last_itr + 2

if not restart:
    c_res = 1
else:
    c_res = last_res
time_itr = 0.0

res = fluxcalc(dim, p, u, v)        # Initialization of residual vector

# Scales for convergence residuals
scale1 = rho * u_inf
scale2 = rho * u_inf * u_inf
scale3 = rho * u_inf * u_inf

if not restart:
    resnorm = []
else:
    res = res_data['res']


while np.abs(c_res) > tol and itr <= 100000:

    del_t = tcalc(dim, u, v, facenormals, faceareas, cellvols, cells)
    p_new = np.zeros_like(p)
    u_new = np.zeros_like(u)
    v_new = np.zeros_like(v)
    for j in range(dim[1]-1):
        for i in range(dim[0]-1):

            tt = del_t/cellvols[cells[i,j]]
            p_new[i,j] = p[i,j] - (beta**2) * (tt) * res[i,j,0]
            u_new[i,j] = u[i,j] - (tt) * ((res[i,j,1]/rho) - u[i,j]*res[i,j,0])
            v_new[i,j] = v[i,j] - (tt) * ((res[i,j,2]/rho) - v[i,j]*res[i,j,0])

    p = np.copy(p_new)
    u = np.copy(u_new)
    v = np.copy(v_new)

    res = fluxcalc(dim, p, u, v)                              # Recalculating residuals

    resnorm.append(0.0)

    for j in range(dim[1]-1):
        for i in range(dim[0]-1):
            resnorm[itr] += ((res[i,j,0]/scale1)**2+(res[i,j,1]/scale2)**2+(res[i,j,2]/scale3)**2)

    if itr == 0:
        resnorm0 = resnorm[itr]

    if itr == 0:
        resnorm0 = resnorm[itr]
        c_res = 1
    else:
        stab = 1e-12
        c_res = (resnorm[itr]) / (resnorm0 + stab)

    time_itr += del_t
    print(f"Residual is {np.abs(c_res)} at time {time_itr}s corresponding to iteration {itr}.")
    itr += 1

    if itr%1000 == 0 or np.abs(c_res) <= tol:                  # Saving every 1000 iterations and after convergence
        resnorm_arr = np.array(resnorm)
        filename = f"solfiles/solution_iter_{itr}.npz"
        np.savez(filename, p=p, u=u, v=v, res=resnorm_arr)
        print(f"Saved solution to {filename}")
```

Listing 6: Interpolation of cell values to nodes. Ghost cells have been used here

```
dim = np.array((97,49))

# Extrapolate values to nodes
loaded_data = np.load('solfiles_NG/solution_iter_17651.npz')
p_loaded = loaded_data['p']
u_loaded = loaded_data['u']
v_loaded = loaded_data['v']

# Ghost layer addition
p_final = np.zeros((dim[0]+1, dim[1]+1))
p_final[1:-1, 1:-1] = np.copy(p_loaded)
u_final = np.zeros((dim[0]+1, dim[1]+1))
u_final[1:-1, 1:-1] = np.copy(u_loaded)
v_final = np.zeros((dim[0]+1, dim[1]+1))
```

```
v_final[1:-1, 1:-1] = np.copy(v_loaded)

# Left
u_final[0,:] = u_inf
v_final[0,:] = v_inf
p_final[0,:] = p_final[1,:]

# Right
u_final[-1,:] = u_final[-2,:]
v_final[-1,:] = v_final[-2,:]
p_final[-1,:] = p_final[-2,:]

# Bottom
p_final[:,0]  =  p_final[:,1]
u_final[:,0] =  u_final[:,1]
v_final[:,0] = -v_final[:,1]

# Top
p_final[:,-1] =  p_final[:,-2]
u_final[:,-1] =  u_final[:,-2]
v_final[:,-1] = -v_final[:,-2]

p_node = np.zeros((dim[0], dim[1]))
u_node = np.zeros_like(p_node)
v_node = np.zeros_like(p_node)

for j in range(dim[1]):
    for i in range(dim[0]):
        p_node[i,j] = 0.25*(p_final[i,j] + p_final[i+1,j] + p_final[i
            +1,j+1] + p_final[i,j+1])
        u_node[i,j] = 0.25*(u_final[i,j] + u_final[i+1,j] + u_final[i
            +1,j+1] + u_final[i,j+1])
        v_node[i,j] = 0.25*(v_final[i,j] + v_final[i+1,j] + v_final[i
            +1,j+1] + v_final[i,j+1])
```

Listing 7: Plotting; please make relevant changes to view the required solution

```
nodes = np.array(nodes)
nodes_arr = np.zeros((dim[0],dim[1],2))
n_itr = 0
for j in range(dim[1]):
    for i in range(dim[0]):
        nodes_arr[i,j] = nodes[n_itr]
        n_itr += 1
x_coords = nodes_arr[:,:,0]
y_coords = nodes_arr[:,:,1]

x_len = x_coords.max() - x_coords.min()
y_len = y_coords.max() - y_coords.min()
fig_width = 8
fig_height = fig_width * (y_len / x_len)

plt.figure(figsize=(fig_width, fig_height))

plt.contourf(x_coords, y_coords, p_node, levels=50, cmap='seismic')
plt.colorbar(label='P')
plt.xlabel('x')
plt.ylabel('y')
plt.grid(False)
plt.axis('equal')
plt.show()
```