Solving the Euler equations in 2D using a finite-volume method

Shraman Maiti[1]

Abstract

A cell-centered finite volume method (FVM) has been used to simulate an inviscid flow in a channel with a bump. The solver uses the Advection Upstream Splitting Method (AUSM) and a first-order Roe scheme to find the inviscid fluxes across the cell faces. Results have been compared for both the schemes. Additionally, a higher-order state reconstruction has been done at the interfaces for more accurate flux calculations. This has been done using the Monotonic Upstream-centered Scheme for Conservation Laws (MUSCL) and has been used with the AUSM scheme.

Keywords: Channel flow, AUSM, Roe scheme, MUSCL, 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 as shown by Roe.[2]

Furthermore, it is a common practice to formulate the fluxes at the interface as some function of the neighboring cell states, using a flux vector-splitting method as used by Liou and Steffen[3] to formulate the Advection Upstream Splitting Method (AUSM). It is an extension to the splitting scheme given by van Leer.[4]

Here, 2D Euler equations have been solved by FVM for a flow within a channel and over a bump. Three flow Mach numbers have been considered, subsonic, transonic and supersonic. For the flux calculations first order schemes such as AUSM has been used and an approximate Riemann problem solution based on the Roe scheme have been used. The flux calculation has also been extended to higher order state reconstruction at the faces using the Monotonic Upstream-centered Scheme for Conservation Laws (MUSCL).

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. P.L. Roe, "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," *Journal of Computational Physics*, October 1981, https://doi.org/10.1016/0021-9991(81)90128-5.

3. Meng-Sing Liou and Christopher J. Steffen Jr., "A New Flux Splitting Scheme," *Journal of Computational Physics*, July 1993, https://doi.org/10.1006/jcph.1993.1122.

4. Eighth International Conference on Numerical Methods in Fluid Dynamics, *Flux-Vector Splitting for the Euler Equation* (Springer, 1982), https://doi.org/10.1007/3-540-11948-5_66.

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

2 Computational methodology

2.1 Problem Solved

The 2D Euler equations in conservative form are given as follows:

$$\boldsymbol{q_t} + \boldsymbol{f_x} + \boldsymbol{f_y} = \mathbf{0} \tag{1}$$

where,

$$\boldsymbol{q} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho e_T \end{bmatrix}, \qquad \boldsymbol{f} = \begin{bmatrix} \rho u_n \\ \rho u^2 + p \\ \rho v^2 + p \\ \rho u h_0 \end{bmatrix},$$

$x_y = \frac{\partial x}{\partial y}$, e_T = specific total energy and h_0 = specific total enthalpy

2.2 Discretization

The discretized form for solving using FVM was as follows:

$$\frac{\bar{q}_{i,j}^{n+1} - \bar{q}_{i,j}^{n}}{\Delta t} + \frac{\sum_k \tilde{F}_k}{\Delta V} = 0 \tag{2}$$

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 calculated specifically for the scheme at use.

2.3 Flux scheme

The flux given by the AUSM is as follows:

$$\tilde{F}_{i+\frac{1}{2},j} = A_{i+\frac{1}{2},j}(\rho_i C_{LS}^{+}[1, u, v, h_0]_i^T + \rho_{i+1} C_{LS}^{-}[1, u, v, h_0]_{i+1}^T + (\tilde{D}_i p_i + \tilde{D}_{i+1} p_{i+1})[0, n_x, n_y, 0]^T) \tag{3}$$

where, C_{LS} are the Liou-Steffen polynomials, which themselves are functions of the van Leer polynomials C_{VL}. The $\tilde{D}$'s are the dissipation coefficients. More on the formulations of C_{LS} and C_{VL} can be found in the papers by Liou and Steffen[5] and van Leer,[6] respectively.

The flux formulated using the Roe scheme is based on the dissipation due to the 4 wavespeeds (coming from the 4 equations being solved, i.e., $\tilde{u}_n$, $\tilde{u}_n$, $|\tilde{u}_n + \tilde{a}|$ and $|\tilde{u}_n - \tilde{a}|$), where $\tilde{u}_n$ is the Roe-averaged normal velocity and $\tilde{a}$ is the Roe-averaged local sound speed. At supersonic speeds, all the characteristic wavespeeds are positive whereas at subsonic speeds three characteristic wavespeeds are positive and one is negative. The total flux at a face is given as the average of the neighboring states and sum of the dissipation due to the four characteristic wavespeeds. The flux is as follows:

5. Liou and Jr., "A New Flux Splitting Scheme."

6. Eighth International Conference on Numerical Methods in Fluid Dynamics, *Flux-Vector Splitting for the Euler Equation.*

$$\tilde{F}_{i+\frac{1}{2},j} = \frac{1}{2} A_{i+\frac{1}{2},j}([\rho u_n, \rho u + p, \rho v + p, \rho u_n h_0]_i^T + [\rho u_n, \rho u + p, \rho v + p, \rho u_n h_0]_{i+1}^T - \tilde{D}) \quad (4)$$

here, $\tilde{D}$ will have 4 components, D_1 and D_2 corresponding to wavespeeds $|\tilde{u}_n|$, D_3 corresponding to wavespeed $|\tilde{u}_n - \tilde{a}|$ and D_4 corresponding to wavespeed $|\tilde{u}_n + \tilde{a}|$. The D's are given as follows:

$$D_{1,2} = |\tilde{u}_n|[(\Delta\rho - \frac{\Delta p}{\tilde{a}^2})[1, \tilde{u}, \tilde{v}, \frac{\tilde{u}^2 + \tilde{v}^2}{2}]^T + [0, \Delta u - |\Delta u_n| n_x, \Delta v - |\Delta u_n| n_y, \tilde{u}\Delta u + \tilde{v}\Delta v - \tilde{u_n}\Delta u_n]^T] \quad (5)$$

$$D_3 = |\tilde{u}_n - \tilde{a}|[(\frac{\Delta p - \tilde{\rho}\tilde{a}\Delta u_n}{2\tilde{a}^2})[1, \tilde{u} - \tilde{a}n_x, \tilde{v} - \tilde{a}n_y, \tilde{h_0} - \tilde{a}\tilde{u_n}]^T] \quad (6)$$

$$D_4 = |\tilde{u}_n + \tilde{a}|[(\frac{\Delta p + \tilde{\rho}\tilde{a}\Delta u_n}{2\tilde{a}^2})[1, \tilde{u} + \tilde{a}n_x, \tilde{v} + \tilde{a}n_y, \tilde{h_0} + \tilde{a}\tilde{u_n}]^T] \quad (7)$$

More on the formulation of the Roe fluxes can be found in the book by Blazek.[7]

2.4 Mesh

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

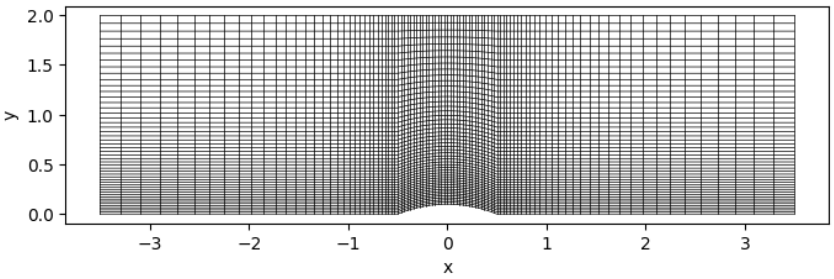

Figure 1: Mesh with 96 x 48 cells

2.5 Boundary conditions

The boundary conditions on the inlet had freestream Mach numbers of 0.4, 0.8 and 1.4 and had a static gauge pressure of 0.01 atm on the inlet. Both the top and bottom faces have been treated as slip walls.

2.6 Higher order state reconstruction

The states at the interfaces have been reconstructed from the cell-center values using the MUSCL technique, wherein the state at the left interface of a cell is the state at the interface subtracted by by a higher order term. This higher order term is a function of the state at the cell and the state difference between itself and the neighboring cells acted upon by a limiter. The limiter is used to limit the slope of the reconstructed functions in order to minimize the oscillations in the solution. In this work, a *minmod* limiter has been used. Similarly the state on the right interface is reconstructed by adding the higher order term to the cell state values.

7. Jiri Blazek, *Computational Fluid Dynamics: Principles and Applications* (Elsevier, 2015), https://doi.org/10.1016/C2013-0-19038-1.

3 Results and Discussions

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

3.1 Flow variables for linear reconstruction

The pressure plots for all the cases are shown in figures 2 through 4. We see that both the schemes give similar results in case of the subsonic and transonic flows. However, when it comes to the supersonic flows, AUSM seems to not be able to develop the trailing shock completely.

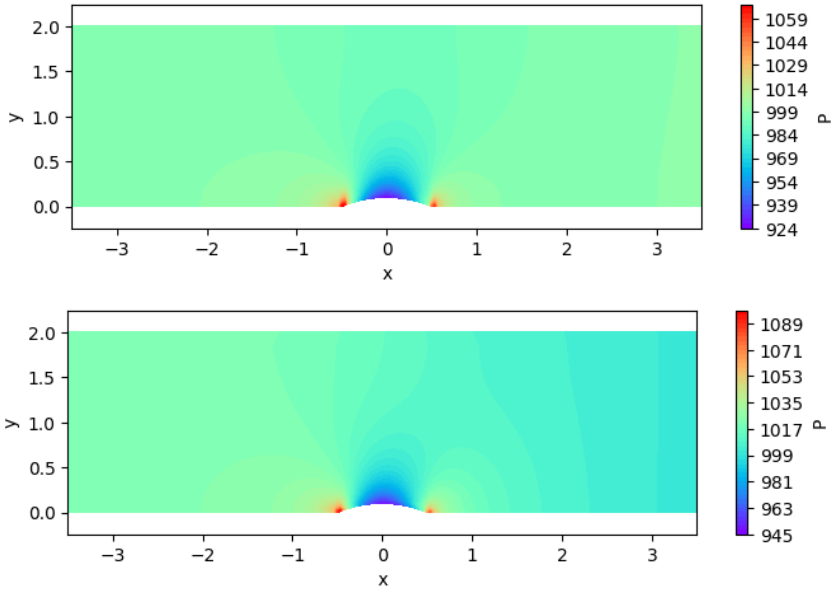

Figure 2: Pressure contours at M=0.4 computed using AUSM (top) and Roe (bottom) schemes

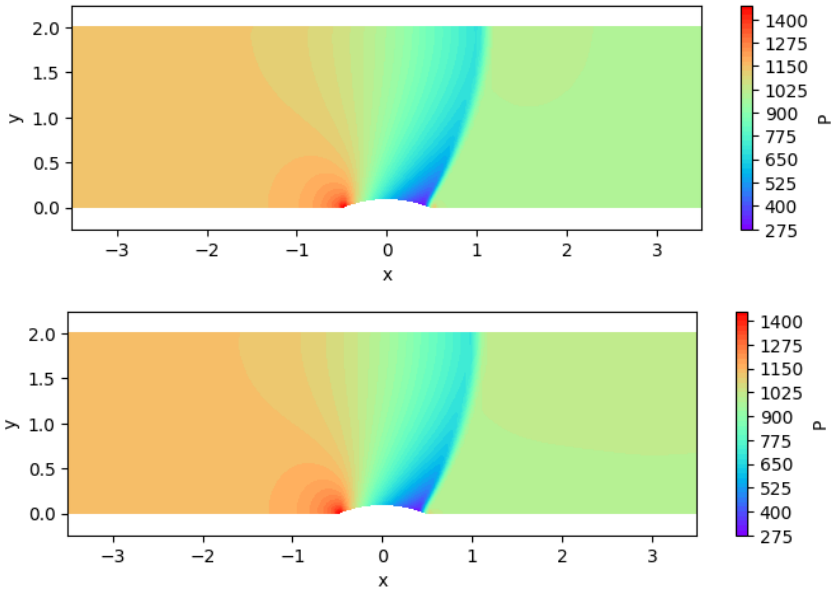

Figure 3: Pressure contours at M=0.8 computed using AUSM (top) and Roe (bottom) schemes

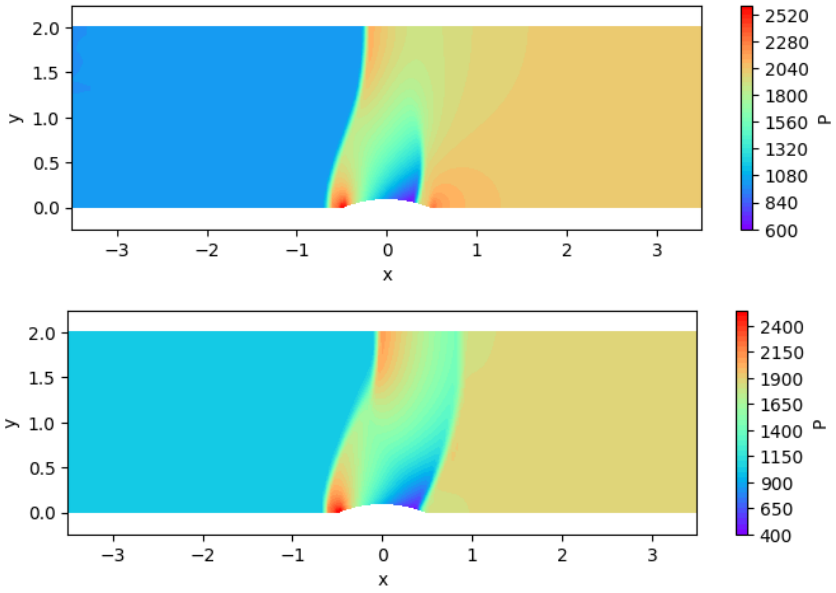

Figure 4: Pressure contours at M=1.4 computed using AUSM (top) and Roe (bottom) schemes

The Mach number contours are shown for the transonic and supersonic cases (M=0.8) in figures 5 and 6:

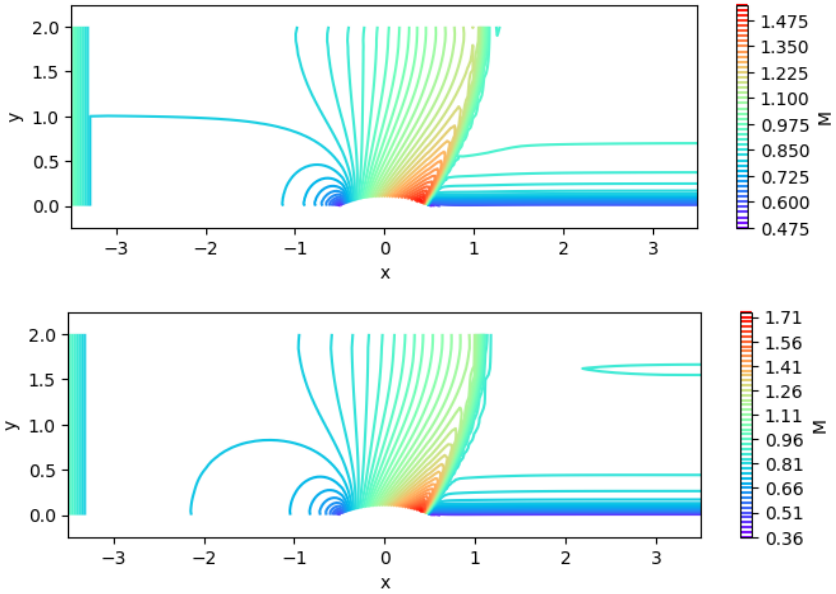

Figure 5: Mach number contours at M=0.8 computed using AUSM (top) and Roe (bottom) schemes

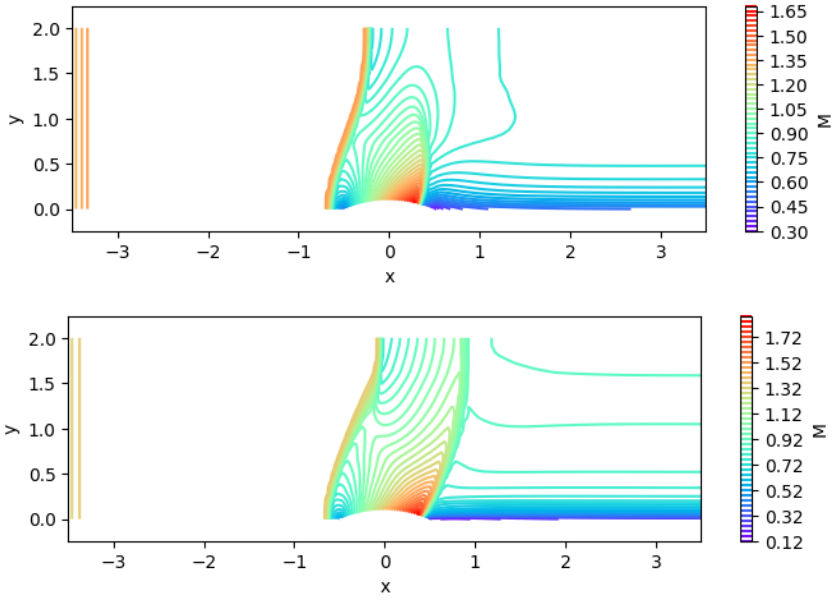

Figure 6: Mach number contours at M=1.4 computed using AUSM (top) and Roe (bottom) schemes

The mismatch in the shock shape for M=1.4 is clearly visible in the Mach contour as well. The velocity streamlines have been plotted in figures 7 and 8:

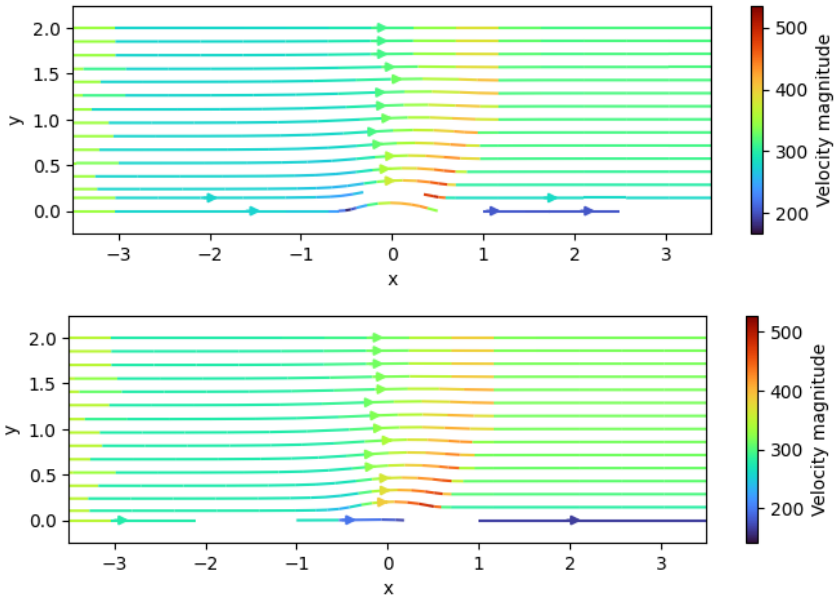

Figure 7: Velocity streamlines at M=0.8 computed using AUSM (top) and Roe (bottom) schemes

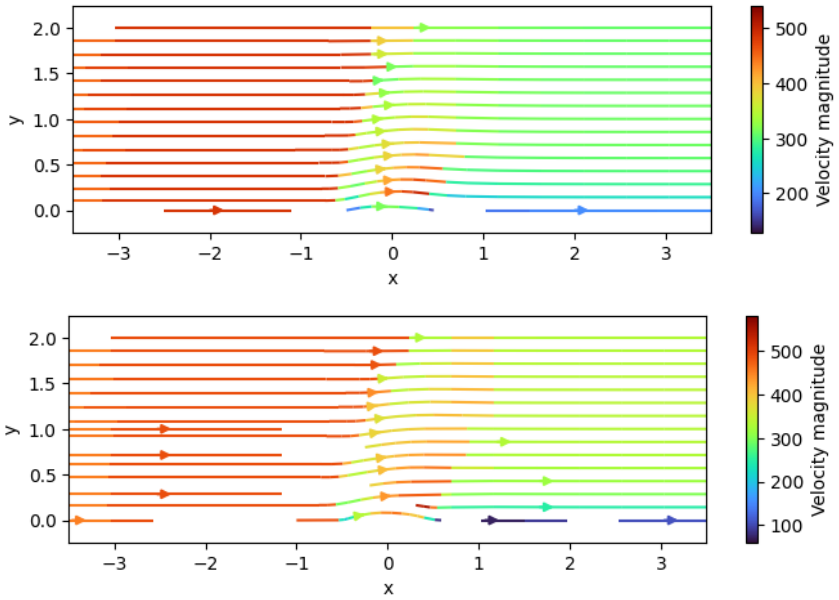

Figure 8: Velocity streamlines at M=1.4 computed using AUSM (top) and Roe (bottom) schemes

The stream plots of velocity corroborates with the physical flow features that are expected at the respective Mach numbers.

3.2 Residual convergence

The residual convergence plots for the 2 schemes have been compared at M=1.4 (supersonic) in figure 9:

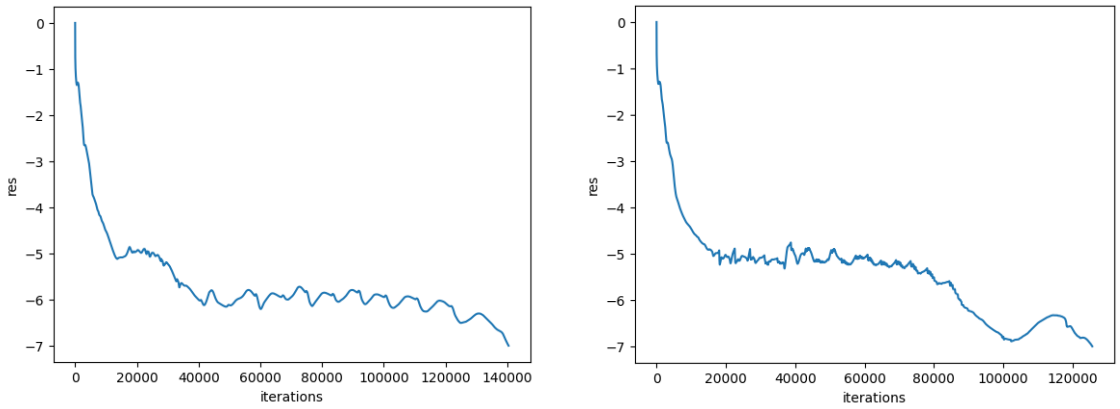

Figure 9: The residual convergence plots for the AUSM scheme (left) and the Roe scheme (right) at M=1.4

Although it is known for AUSM to be faster than the Roe scheme, but here we see that Roe is giving a slightly faster convergence although the difference is quite less.

3.3 Higher order reconstructed case

The MUSCL technique was applied to the transonic flow case and the results are shown below in figure 10. Its residual convergence plot (converged till $1e-12$ is shown in figure 11.

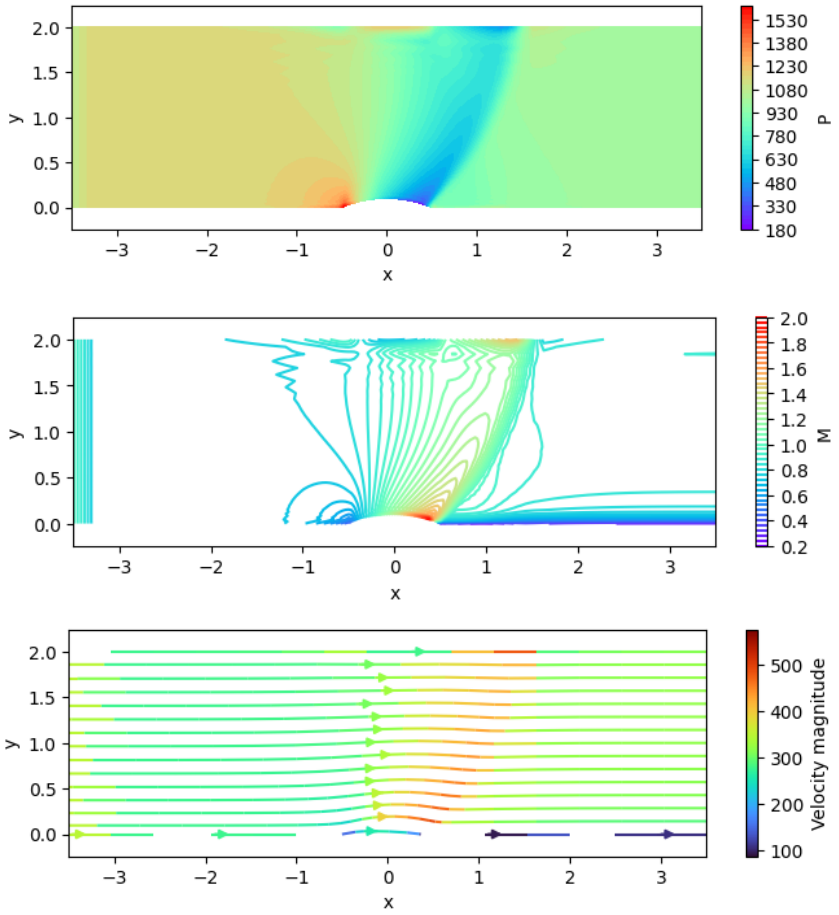

Figure 10: Pressure contours (top), Mach contours (middle) and velocity streamlines (bottom) at M=0.8 computed using AUSM with MUSCL reconstruction

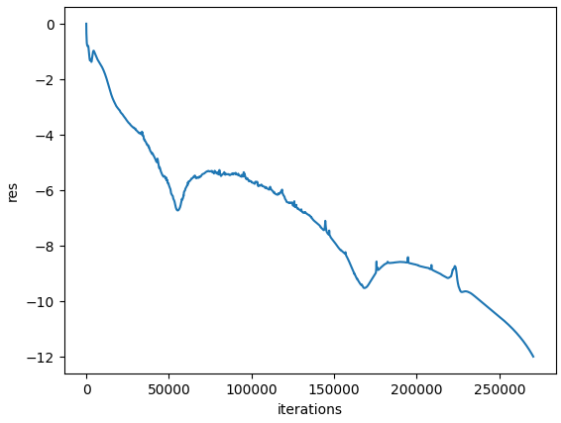

Figure 11: Residual convergence for M=0.8 with MUSCL reconstruction

The higher order reconstruction solutions are giving an oscillatory result, in spite of using a limiter.

4 Conclusion

For the linear reconstruction of the states, Roe seems to be giving more accurate results than AUSM, which is quite evident in the M=1.4 (supersonic) case. But due to their highly dissipative nature, both the schemes converge to non-oscillatory stable results. But the flow features like shocks are quite smeared out. In the case of higher-order reconstructed states, as the discontinuities at the cell interfaces are lower, the dissipative property of the scheme is lower as well and therefore the results are oscillatory. But the convergence obtained was much faster as it reached $1e-7$ at around 150,000 iterations against 270,000 for regular AUSM.

Appendix

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

Listing 1: Initialization

```
import numpy as np
import matplotlib.pyplot as plt
file1 = 'meshfile.txt'

T_inf = 300
R_air = 287
M_inf = 1.4
p_stat = 1000                                # Freestream static pressure is
    given to be 0.01 atm ~ 1e3 Pa
gamma = 1.4
rho_inf = p_stat/(R_air*T_inf)
a_inf = np.sqrt(gamma*p_stat/rho_inf)
u_inf = M_inf*a_inf
v_inf = 0
cfl = 0.5
tol = 1e-7
if M_inf < 1:
    mode = 'sub'
else:
    mode = 'sup'
restart = False
if restart:
    last_itr = 130000
    last_time = 0.6938204408751635
high_recon = False
```

Listing 2: Meshing

```
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]

vols_2d = np.reshape(cellvols, (96,48), order='C')
```

Listing 3: Boundary conditions

```
# Boundary Conditions

def in_bc(dim, r, mode):
    if mode == 'sub':
        rho_l = rho_inf
        u_l = u_inf
        v_l = v_inf
        p_l = r[3, 0, :]

    if mode == 'sup':
        rho_l = rho_inf
        u_l = u_inf
        v_l = v_inf
        p_l = p_stat

    return rho_l, u_l, v_l, p_l

def out_bc(dim, r, mode):
    if mode == 'sub':
        rho_r = r[0, dim[0]-2, :]
        u_r = r[1, dim[0]-2, :]
        v_r = r[2, dim[0]-2, :]
        p_r = p_stat

    if mode == 'sup':
        rho_r = r[0, dim[0]-2, :]
```

```
27          u_r = r[1, dim[0]-2, :]
28          v_r = r[2, dim[0]-2, :]
29          p_r = r[3, dim[0]-2, :]
30
31      return rho_r, u_r, v_r, p_r
32
33  def slipwall(dim, r, yfacesA, yfacesN):
34      rho_d = r[0, :, 0]
35      nx = yfacesN[:,0,0]/yfacesA[:,0]
36      ny = yfacesN[:,0,1]/yfacesA[:,0]
37      u_d = (r[1, :, 0]*(ny**2-nx**2)-2*r[2, :, 0]*nx*ny)/(nx**2+ny**2)
38      v_d = (r[2, :, 0]*(nx**2-ny**2)-2*r[1, :, 0]*nx*ny)/(nx**2+ny**2)
39      p_d = r[3, :, 0]
40      rho_u = r[0, :, dim[1]-2]
41      p_u = r[3, :, dim[1]-2]
42
43      return rho_d, p_d, rho_u, p_u, u_d, v_d
```

Listing 4: Roe flux

```
1   def fluxroe(dim, r):
2
3
4       # Fluxes initialization
5       xFluxes = np.zeros((dim[0], dim[1]-1, 4))
6       yFluxes = np.zeros((dim[0]-1, dim[1], 4))
7
8       # X-face fluxes
9       for i in range(1, dim[0]-1):
10          area = xfacesA[i,:]
11          nx = xfacesN[i,:,0]/area
12          ny = xfacesN[i,:,1]/area
13          # Left states
14          rho_l = r[0, i-1, :]
15          u_l = r[1, i-1, :]
16          v_l = r[2, i-1, :]
17          p_l = r[3, i-1, :]
18          h0_l = (gamma/(gamma - 1))*(p_l/rho_l) + 0.5 * (u_l**2 + v_l
                **2)
19          u_nl = u_l*nx + v_l*ny
20          vel_l = np.sqrt(u_l**2 + v_l**2)
21          # Right states
22          rho_r = r[0, i, :]
23          u_r = r[1, i, :]
24          v_r = r[2, i, :]
25          p_r = r[3, i, :]
26          h0_r = (gamma/(gamma - 1))*(p_r/rho_r) + 0.5 * (u_r**2 + v_r
                **2)
27          u_nr = u_r*nx + v_r*ny
28          vel_r = np.sqrt(u_r**2 + v_r**2)
29
30          # Roe states
31          r_factor = np.sqrt(rho_r/rho_l)
32          rho_roe = np.sqrt(rho_r*rho_l)
33          u_roe = (u_l + u_r*r_factor)/(1+r_factor)
34          v_roe = (v_l + v_r*r_factor)/(1+r_factor)
35          vel_roe = np.sqrt(u_roe**2 + v_roe**2)
36          h0_roe = (h0_l + h0_r*r_factor)/(1+r_factor)
```

```
37          a_roe = np.sqrt((gamma-1.0)*(h0_roe-0.5*(vel_roe**2)))
38          un_roe = u_roe*nx + v_roe*ny
39
40          # Dissipation using Roe states
41
42          alpha0 = area*np.abs(un_roe)*((rho_r-rho_l)-(p_r-p_l)/a_roe**2)
43          alpha1 = (area/(2*a_roe**2))*np.abs(un_roe+a_roe)*((p_r-p_l)+(
                rho_roe*a_roe*(u_nr-u_nl)))
44          alpha2 = (area/(2*a_roe**2))*np.abs(un_roe-a_roe)*((p_r-p_l)-(
                rho_roe*a_roe*(u_nr-u_nl)))
45          alpha3 = alpha0 + alpha1 + alpha2
46          alpha4 = a_roe*(alpha1-alpha2)
47          alpha5 = area*np.abs(un_roe)*((rho_roe*(u_r-u_l))-(nx*rho_roe*(
                u_nr-u_nl)))
48          alpha6 = area*np.abs(un_roe)*((rho_roe*(v_r-v_l))-(ny*rho_roe*(
                u_nr-u_nl)))
49
50          diss0 = alpha3
51          diss1 = u_roe*alpha3 + nx*alpha4 + alpha5
52          diss2 = v_roe*alpha3 + ny*alpha4 + alpha6
53          diss3 = h0_roe*(alpha3-alpha0) + un_roe*alpha4 + u_roe*alpha5 +
                 v_roe*alpha6 + 0.5*(u_roe**2+v_roe**2)*alpha0
54
55          fm_pl = 0.5*area*rho_l*u_nl
56          fm_mi = 0.5*area*rho_r*u_nr
57
58          xFluxes[i,:,0] = (fm_pl + fm_mi) - 0.5*diss0
59          xFluxes[i,:,1] = (fm_pl*u_l + fm_mi*u_r) + 0.5*(p_l+p_r)*nx*
                area - 0.5*diss1
60          xFluxes[i,:,2] = (fm_pl*v_l + fm_mi*v_r) + 0.5*(p_l+p_r)*ny*
                area - 0.5*diss2
61          xFluxes[i,:,3] = (fm_pl*h0_l + fm_mi*h0_r) - 0.5*diss3
62
63      # Left boundary flux
64      area = xfacesA[0,:]
65      nx = xfacesN[0,:,0]/area
66      ny = xfacesN[0,:,1]/area
67      # Left states
68      bc_l = in_bc(dim, r, mode)
69      rho_l = bc_l[0]
70      u_l = bc_l[1]
71      v_l = bc_l[2]
72      p_l = bc_l[3]
73      h0_l = (gamma/(gamma - 1))*(p_l/rho_l) + 0.5 * (u_l**2 + v_l**2)
74      u_nl = u_l*nx + v_l*ny
75      vel_l = np.sqrt(u_l**2 + v_l**2)
76      # Right states
77      rho_r = r[0, 0, :]
78      u_r = r[1, 0, :]
79      v_r = r[2, 0, :]
80      p_r = r[3, 0, :]
81      h0_r = (gamma/(gamma - 1))*(p_r/rho_r) + 0.5 * (u_r**2 + v_r**2)
82      u_nr = u_r*nx + v_r*ny
83      vel_r = np.sqrt(u_r**2 + v_r**2)
84
85      # Roe states
86      r_factor = np.sqrt(rho_r/rho_l)
87      rho_roe = np.sqrt(rho_r*rho_l)
```

```
88      u_roe = (u_l + u_r*r_factor)/(1+r_factor)
89      v_roe = (v_l + v_r*r_factor)/(1+r_factor)
90      vel_roe = np.sqrt(u_roe**2 + v_roe**2)
91      h0_roe = (h0_l + h0_r*r_factor)/(1+r_factor)
92      a_roe = np.sqrt((gamma-1.0)*(h0_roe-0.5*(vel_roe**2)))
93      un_roe = u_roe*nx + v_roe*ny
94
95      # Dissipation using Roe states
96
97      alpha0 = area*np.abs(un_roe)*((rho_r-rho_l)-(p_r-p_l)/a_roe**2)
98      alpha1 = (area/(2*a_roe**2))*np.abs(un_roe+a_roe)*((p_r-p_l)+(
            rho_roe*a_roe*(u_nr-u_nl)))
99      alpha2 = (area/(2*a_roe**2))*np.abs(un_roe-a_roe)*((p_r-p_l)-(
            rho_roe*a_roe*(u_nr-u_nl)))
100     alpha3 = alpha0 + alpha1 + alpha2
101     alpha4 = a_roe*(alpha1-alpha2)
102     alpha5 = area*np.abs(un_roe)*((rho_roe*(u_r-u_l))-(nx*rho_roe*(u_nr
            -u_nl)))
103     alpha6 = area*np.abs(un_roe)*((rho_roe*(v_r-v_l))-(ny*rho_roe*(u_nr
            -u_nl)))
104
105     diss0 = alpha3
106     diss1 = u_roe*alpha3 + nx*alpha4 + alpha5
107     diss2 = v_roe*alpha3 + ny*alpha4 + alpha6
108     diss3 = h0_roe*(alpha3-alpha0) + un_roe*alpha4 + u_roe*alpha5 +
            v_roe*alpha6 + 0.5*(u_roe**2+v_roe**2)*alpha0
109
110     fm_pl = 0.5*area*rho_l*u_nl
111     fm_mi = 0.5*area*rho_r*u_nr
112
113     xFluxes[0,:,0] = (fm_pl + fm_mi) - 0.5*diss0
114     xFluxes[0,:,1] = (fm_pl*u_l + fm_mi*u_r) + 0.5*(p_l+p_r)*nx*area -
            0.5*diss1
115     xFluxes[0,:,2] = (fm_pl*v_l + fm_mi*v_r) + 0.5*(p_l+p_r)*ny*area -
            0.5*diss2
116     xFluxes[0,:,3] = (fm_pl*h0_l + fm_mi*h0_r) - 0.5*diss3
117
118     # Right boundary flux
119     area = xfacesA[dim[0]-1,:]
120     nx = xfacesN[dim[0]-1,:,0]/area
121     ny = xfacesN[dim[0]-1,:,1]/area
122     # Left states
123     rho_l = r[0, dim[0]-2, :]
124     u_l = r[1, dim[0]-2, :]
125     v_l = r[2, dim[0]-2, :]
126     p_l = r[3, dim[0]-2, :]
127     h0_l = (gamma/(gamma - 1))*(p_l/rho_l) + 0.5 * (u_l**2 + v_l**2)
128     u_nl = u_l*nx + v_l*ny
129     vel_l = np.sqrt(u_l**2 + v_l**2)
130     # Right states
131     bc_r = out_bc(dim, r, mode)
132     rho_r = bc_r[0]
133     u_r = bc_r[1]
134     v_r = bc_r[2]
135     p_r = bc_r[3]
136     h0_r = (gamma/(gamma - 1))*(p_r/rho_r) + 0.5 * (u_r**2 + v_r**2)
137     u_nr = u_r*nx + v_r*ny
138     vel_r = np.sqrt(u_r**2 + v_r**2)
```

```
139
140         # Roe states
141         r_factor = np.sqrt(rho_r/rho_l)
142         rho_roe = np.sqrt(rho_r*rho_l)
143         u_roe = (u_l + u_r*r_factor)/(1+r_factor)
144         v_roe = (v_l + v_r*r_factor)/(1+r_factor)
145         vel_roe = np.sqrt(u_roe**2 + v_roe**2)
146         h0_roe = (h0_l + h0_r*r_factor)/(1+r_factor)
147         a_roe = np.sqrt((gamma-1.0)*(h0_roe-0.5*(vel_roe**2)))
148         un_roe = u_roe*nx + v_roe*ny
149
150         # Dissipation using Roe states
151
152         alpha0 = area*np.abs(un_roe)*((rho_r-rho_l)-(p_r-p_l)/a_roe**2)
153         alpha1 = (area/(2*a_roe**2))*np.abs(un_roe+a_roe)*((p_r-p_l)+(
                rho_roe*a_roe*(u_nr-u_nl)))
154         alpha2 = (area/(2*a_roe**2))*np.abs(un_roe-a_roe)*((p_r-p_l)-(
                rho_roe*a_roe*(u_nr-u_nl)))
155         alpha3 = alpha0 + alpha1 + alpha2
156         alpha4 = a_roe*(alpha1-alpha2)
157         alpha5 = area*np.abs(un_roe)*((rho_roe*(u_r-u_l))-(nx*rho_roe*(u_nr
                -u_nl)))
158         alpha6 = area*np.abs(un_roe)*((rho_roe*(v_r-v_l))-(ny*rho_roe*(u_nr
                -u_nl)))
159
160         diss0 = alpha3
161         diss1 = u_roe*alpha3 + nx*alpha4 + alpha5
162         diss2 = v_roe*alpha3 + ny*alpha4 + alpha6
163         diss3 = h0_roe*(alpha3-alpha0) + un_roe*alpha4 + u_roe*alpha5 +
                v_roe*alpha6 + 0.5*(u_roe**2+v_roe**2)*alpha0
164
165         fm_pl = 0.5*area*rho_l*u_nl
166         fm_mi = 0.5*area*rho_r*u_nr
167
168         xFluxes[dim[0]-1,:,0] = (fm_pl + fm_mi) - 0.5*diss0
169         xFluxes[dim[0]-1,:,1] = (fm_pl*u_l + fm_mi*u_r) + 0.5*(p_l+p_r)*nx*
                area - 0.5*diss1
170         xFluxes[dim[0]-1,:,2] = (fm_pl*v_l + fm_mi*v_r) + 0.5*(p_l+p_r)*ny*
                area - 0.5*diss2
171         xFluxes[dim[0]-1,:,3] = (fm_pl*h0_l + fm_mi*h0_r) - 0.5*diss3
172
173         # Y-face fluxes
174         for j in range(1, dim[1]-1):
175             area = yfacesA[:,j]
176             nx = yfacesN[:,j,0]/area
177             ny = yfacesN[:,j,1]/area
178             # Lower states
179             rho_d = r[0, :, j-1]
180             u_d = r[1, :, j-1]
181             v_d = r[2, :, j-1]
182             p_d = r[3, :, j-1]
183             h0_d = (gamma/(gamma - 1))*(p_d/rho_d) + 0.5 * (u_d**2 + v_d
                    **2)
184             u_nd = u_d*nx + v_d*ny
185             vel_d = np.sqrt(u_d**2 + v_d**2)
186             # Upper states
187             rho_u = r[0, :, j]
188             u_u = r[1, :, j]
```

```
            v_u = r[2, :, j]
            p_u = r[3, :, j]
            h0_u = (gamma/(gamma - 1))*(p_u/rho_u) + 0.5 * (u_u**2 + v_u
                **2)
            u_nu = u_u*nx + v_u*ny
            vel_u = np.sqrt(u_u**2 + v_u**2)

            # Roe states
            r_factor = np.sqrt(rho_u/rho_d)
            rho_roe = np.sqrt(rho_u*rho_d)
            u_roe = (u_d + u_u*r_factor)/(1+r_factor)
            v_roe = (v_d + v_u*r_factor)/(1+r_factor)
            vel_roe = np.sqrt(u_roe**2 + v_roe**2)
            h0_roe = (h0_d + h0_u*r_factor)/(1+r_factor)
            a_roe = np.sqrt((gamma-1.0)*(h0_roe-0.5*(vel_roe**2)))
            un_roe = u_roe*nx + v_roe*ny

            # Dissipation using Roe states

            alpha0 = area*np.abs(un_roe)*((rho_u-rho_d)-(p_u-p_d)/a_roe**2)
            alpha1 = (area/(2*a_roe**2))*np.abs(un_roe+a_roe)*((p_u-p_d)+(
                rho_roe*a_roe*(u_nu-u_nd)))
            alpha2 = (area/(2*a_roe**2))*np.abs(un_roe-a_roe)*((p_u-p_d)-(
                rho_roe*a_roe*(u_nu-u_nd)))
            alpha3 = alpha0 + alpha1 + alpha2
            alpha4 = a_roe*(alpha1-alpha2)
            alpha5 = area*np.abs(un_roe)*((rho_roe*(u_u-u_d))-(nx*rho_roe*(
                u_nu-u_nd)))
            alpha6 = area*np.abs(un_roe)*((rho_roe*(v_u-v_d))-(ny*rho_roe*(
                u_nu-u_nd)))

            diss0 = alpha3
            diss1 = u_roe*alpha3 + nx*alpha4 + alpha5
            diss2 = v_roe*alpha3 + ny*alpha4 + alpha6
            diss3 = h0_roe*(alpha3-alpha0) + un_roe*alpha4 + u_roe*alpha5 +
                 v_roe*alpha6 + 0.5*(u_roe**2+v_roe**2)*alpha0

            fm_pl = 0.5*area*rho_d*u_nd
            fm_mi = 0.5*area*rho_u*u_nu

            yFluxes[:,j,0] = (fm_pl + fm_mi) - 0.5*diss0
            yFluxes[:,j,1] = (fm_pl*u_d + fm_mi*u_u) + 0.5*(p_d+p_u)*nx*
                area - 0.5*diss1
            yFluxes[:,j,2] = (fm_pl*v_d + fm_mi*v_u) + 0.5*(p_d+p_u)*ny*
                area - 0.5*diss2
            yFluxes[:,j,3] = (fm_pl*h0_d + fm_mi*h0_u) - 0.5*diss3

            bc_w = slipwall(dim,r,yfacesA,yfacesN)

        # Lower boundary flux
        area = yfacesA[:,0]
        nx = yfacesN[:,0,0]/area
        ny = yfacesN[:,0,1]/area
        # Lower states
        rho_d = bc_w[0]
        u_d = r[1, :, 0]
        v_d = -r[2, :, 0]
        p_d = bc_w[1]
```

```
239     h0_d = (gamma/(gamma - 1))*(p_d/rho_d) + 0.5 * (u_d**2 + v_d**2)
240     u_nd = u_d*nx + v_d*ny
241     vel_d = np.sqrt(u_d**2 + v_d**2)
242     # Upper states
243     rho_u = r[0, :, 0]
244     u_u = r[1, :, 0]
245     v_u = r[2, :, 0]
246     p_u = r[3, :, 0]
247     h0_u = (gamma/(gamma - 1))*(p_u/rho_u) + 0.5 * (u_u**2 + v_u**2)
248     u_nu = u_u*nx + v_u*ny
249     vel_u = np.sqrt(u_u**2 + v_u**2)
250
251     # Roe states not needed at wall
252
253     fm_pl = 0
254     fm_mi = 0
255
256     yFluxes[:,0,0] = (fm_pl + fm_mi)
257     yFluxes[:,0,1] = (fm_pl*u_d + fm_mi*u_u) + 0.5*(p_d+p_u)*nx*area
258     yFluxes[:,0,2] = (fm_pl*v_d + fm_mi*v_u) + 0.5*(p_d+p_u)*ny*area
259     yFluxes[:,0,3] = (fm_pl*h0_d + fm_mi*h0_u)
260
261     # Upper boundary flux
262     area = yfacesA[:,dim[1]-1]
263     nx = yfacesN[:,dim[1]-1,0]/area
264     ny = yfacesN[:,dim[1]-1,1]/area
265     # Lower states
266     rho_d = r[0, :, dim[1]-2]
267     u_d = r[1, :, dim[1]-2]
268     v_d = r[2, :, dim[1]-2]
269     p_d = r[3, :, dim[1]-2]
270     h0_d = (gamma/(gamma - 1))*(p_d/rho_d) + 0.5 * (u_d**2 + v_d**2)
271     u_nd = u_d*nx + v_d*ny
272     vel_d = np.sqrt(u_d**2 + v_d**2)
273     # Upper states
274     rho_u = bc_w[2]
275     u_u = r[1, :, dim[1]-2]
276     v_u = -r[2, :, dim[1]-2]
277     p_u = bc_w[3]
278     h0_u = (gamma/(gamma - 1))*(p_u/rho_u) + 0.5 * (u_u**2 + v_u**2)
279     u_nu = u_u*nx + v_u*ny
280     vel_u = np.sqrt(u_u**2 + v_u**2)
281
282     # Roe states not needed at wall
283
284     fm_pl = 0
285     fm_mi = 0
286
287     yFluxes[:,dim[1]-1,0] = (fm_pl + fm_mi)
288     yFluxes[:,dim[1]-1,1] = (fm_pl*u_d + fm_mi*u_u) + 0.5*(p_d+p_u)*nx*
            area
289     yFluxes[:,dim[1]-1,2] = (fm_pl*v_d + fm_mi*v_u) + 0.5*(p_d+p_u)*ny*
            area
290     yFluxes[:,dim[1]-1,3] = (fm_pl*h0_d + fm_mi*h0_u)
291
292     res = ((xFluxes[1:, :, :] - xFluxes[:-1, :, :]) + (yFluxes[:, 1:,
            :] - yFluxes[:, :-1, :]))
293
```

```
294     return res
```

Listing 5: AUSM flux

```
1  def fluxls(dim, r):
2
3      # Fluxes initialization
4      xFluxes = np.zeros((dim[0], dim[1]-1, 4))
5      yFluxes = np.zeros((dim[0]-1, dim[1], 4))
6
7      # X-face fluxes
8      for i in range(1, dim[0]-1):
9          area = xfacesA[i,:]
10         nx = xfacesN[i,:,0]/area
11         ny = xfacesN[i,:,1]/area
12         # Left states
13         rho_l = r[0, i-1, :]
14         u_l = r[1, i-1, :]
15         v_l = r[2, i-1, :]
16         p_l = r[3, i-1, :]
17         h0_l = (gamma/(gamma - 1))*(p_l/rho_l) + 0.5 * (u_l**2 + v_l
               **2)
18         a_l = np.sqrt(gamma*(p_l/rho_l))
19         u_nl = u_l*nx + v_l*ny
20         # Right states
21         rho_r = r[0, i, :]
22         u_r = r[1, i, :]
23         v_r = r[2, i, :]
24         p_r = r[3, i, :]
25         h0_r = (gamma/(gamma - 1))*(p_r/rho_r) + 0.5 * (u_r**2 + v_r
               **2)
26         a_r = np.sqrt(gamma*(p_r/rho_r))
27         u_nr = u_r*nx + v_r*ny
28
29         a_avg = 0.5*(a_l + a_r)
30         xma_l = u_nl/a_avg
31         xma_r = u_nr/a_avg
32
33         al_l = 0.5*(1.0 + np.sign((xma_l)))
34         al_r = 0.5*(1.0 - np.sign((xma_r)))
35
36         be_l = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_l)))
37         be_r = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_r)))
38
39         # Defining the van Leer polynomials
40         d_pl = 0.25*(((xma_l+1)**2)*(2-xma_l))
41         d_mi = 0.25*(((xma_r-1)**2)*(2+xma_r))
42
43         d_l = al_l*(1+be_l) - be_l*d_pl
44         d_r = al_r*(1+be_r) - be_r*d_mi
45
46         cvl_pl = al_l * (1+be_l) * xma_l - be_l * 0.25 * (1+xma_l)**2
47         cvl_mi = al_r * (1+be_r) * xma_r + be_r * 0.25 * (1-xma_r)**2
48
49         # Using the van Leer polynoials to define the Liou-Stefan
               polynomials
50         cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
51         cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
```

```
52
53          # Defining the mass-flux using the LS formulation
54          fm_pl = area * a_avg * cls_pl * rho_l
55          fm_mi = area * a_avg * cls_mi * rho_r
56          xFluxes[i,:, 0] = fm_pl + fm_mi
57          xFluxes[i,:, 1] = fm_pl * u_l + fm_mi * u_r + (d_l*p_l + d_r*
                p_r) * nx*area
58          xFluxes[i,:, 2] = fm_pl * v_l + fm_mi * v_r + (d_l*p_l + d_r*
                p_r) * ny*area
59          xFluxes[i,:, 3] = fm_pl * h0_l + fm_mi * h0_r
60
61      # Left boundary flux
62      area = xfacesA[0,:]
63      nx = xfacesN[0,:,0]/area
64      ny = xfacesN[0,:,1]/area
65      # Left states
66      bc_l = in_bc(dim, r, mode)
67      rho_l = bc_l[0]
68      u_l = bc_l[1]
69      v_l = bc_l[2]
70      p_l = bc_l[3]
71      h0_l = (gamma/(gamma - 1))*(p_l/rho_l) + 0.5 * (u_l**2 + v_l**2)
72      a_l = np.sqrt(gamma*(p_l/rho_l))
73      u_nl = u_l*nx + v_l*ny
74      # Right states
75      rho_r = r[0, 0, :]
76      u_r = r[1, 0, :]
77      v_r = r[2, 0, :]
78      p_r = r[3, 0, :]
79      h0_r = (gamma/(gamma - 1))*(p_r/rho_r) + 0.5 * (u_r**2 + v_r**2)
80      a_r = np.sqrt(gamma*(p_r/rho_r))
81      u_nr = u_r*nx + v_r*ny
82
83      a_avg = 0.5*(a_l + a_r)
84      xma_l = u_nl/a_avg
85      xma_r = u_nr/a_avg
86
87      al_l = 0.5*(1.0 + np.sign((xma_l)))
88      al_r = 0.5*(1.0 - np.sign((xma_r)))
89
90      be_l = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_l)))
91      be_r = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_r)))
92
93      # Defining the van Leer polynomials
94      d_pl = 0.25*(((xma_l+1)**2)*(2-xma_l))
95      d_mi = 0.25*(((xma_r-1)**2)*(2+xma_r))
96
97      d_l = al_l*(1+be_l) - be_l*d_pl
98      d_r = al_r*(1+be_r) - be_r*d_mi
99
100     cvl_pl = al_l * (1+be_l) * xma_l - be_l * 0.25 * (1+xma_l)**2
101     cvl_mi = al_r * (1+be_r) * xma_r + be_r * 0.25 * (1-xma_r)**2
102
103     # Using the van Leer polynoials to define the Liou-Stefan
            polynomials
104     cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
105     cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
106
```

```
107       # Defining the mass-flux using the LS formulation
108       fm_pl = area * a_avg * cls_pl * rho_l
109       fm_mi = area * a_avg * cls_mi * rho_r
110       xFluxes[0,:, 0] = fm_pl + fm_mi
111       xFluxes[0,:, 1] = fm_pl * u_l + fm_mi * u_r + (d_l*p_l + d_r*p_r) *
              nx*area
112       xFluxes[0,:, 2] = fm_pl * v_l + fm_mi * v_r + (d_l*p_l + d_r*p_r) *
              ny*area
113       xFluxes[0,:, 3] = fm_pl * h0_l + fm_mi * h0_r
114
115       # Right boundary flux
116       area = xfacesA[dim[0]-1,:]
117       nx = xfacesN[dim[0]-1,:,0]/area
118       ny = xfacesN[dim[0]-1,:,1]/area
119       # Left states
120       rho_l = r[0, dim[0]-2, :]
121       u_l = r[1, dim[0]-2, :]
122       v_l = r[2, dim[0]-2, :]
123       p_l = r[3, dim[0]-2, :]
124       h0_l = (gamma/(gamma - 1))*(p_l/rho_l) + 0.5 * (u_l**2 + v_l**2)
125       a_l = np.sqrt(gamma*(p_l/rho_l))
126       u_nl = u_l*nx + v_l*ny
127       # Right states
128       bc_r = out_bc(dim, r, mode)
129       rho_r = bc_r[0]
130       u_r = bc_r[1]
131       v_r = bc_r[2]
132       p_r = bc_r[3]
133       h0_r = (gamma/(gamma - 1))*(p_r/rho_r) + 0.5 * (u_r**2 + v_r**2)
134       a_r = np.sqrt(gamma*(p_r/rho_r))
135       u_nr = u_r*nx + v_r*ny
136
137       a_avg = 0.5*(a_l + a_r)
138       xma_l = u_nl/a_avg
139       xma_r = u_nr/a_avg
140
141       al_l = 0.5*(1.0 + np.sign((xma_l)))
142       al_r = 0.5*(1.0 - np.sign((xma_r)))
143
144       be_l = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_l)))
145       be_r = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_r)))
146
147       # Defining the van Leer polynomials
148       d_pl = 0.25*(((xma_l+1)**2)*(2-xma_l))
149       d_mi = 0.25*(((xma_r-1)**2)*(2+xma_r))
150
151       d_l = al_l*(1+be_l) - be_l*d_pl
152       d_r = al_r*(1+be_r) - be_r*d_mi
153
154       cvl_pl = al_l * (1+be_l) * xma_l - be_l * 0.25 * (1+xma_l)**2
155       cvl_mi = al_r * (1+be_r) * xma_r + be_r * 0.25 * (1-xma_r)**2
156
157       # Using the van Leer polynoials to define the Liou-Stefan
              polynomials
158       cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
159       cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
160
161       # Defining the mass-flux using the LS formulation
```

```
        fm_pl = area * a_avg * cls_pl * rho_l
        fm_mi = area * a_avg * cls_mi * rho_r
        xFluxes[dim[0]-1,:, 0] = fm_pl + fm_mi
        xFluxes[dim[0]-1,:, 1] = fm_pl * u_l + fm_mi * u_r + (d_l*p_l + d_r
            *p_r) * nx*area
        xFluxes[dim[0]-1,:, 2] = fm_pl * v_l + fm_mi * v_r + (d_l*p_l + d_r
            *p_r) * ny*area
        xFluxes[dim[0]-1,:, 3] = fm_pl * h0_l + fm_mi * h0_r

        # Y-face fluxes
        for j in range(1, dim[1]-1):
            area = yfacesA[:,j]
            nx = yfacesN[:,j,0]/area
            ny = yfacesN[:,j,1]/area
            # Lower states
            rho_d = r[0, :, j-1]
            u_d = r[1, :, j-1]
            v_d = r[2, :, j-1]
            p_d = r[3, :, j-1]
            h0_d = (gamma/(gamma - 1))*(p_d/rho_d) + 0.5 * (u_d**2 + v_d
                **2)
            a_d = np.sqrt(gamma*(p_d/rho_d))
            u_nd = u_d*nx + v_d*ny
            # Upper states
            rho_u = r[0, :, j]
            u_u = r[1, :, j]
            v_u = r[2, :, j]
            p_u = r[3, :, j]
            h0_u = (gamma/(gamma - 1))*(p_u/rho_u) + 0.5 * (u_u**2 + v_u
                **2)
            a_u = np.sqrt(gamma*(p_u/rho_u))
            u_nu = u_u*nx + v_u*ny

            a_avg = 0.5*(a_d + a_u)
            xma_d = u_nd/a_avg
            xma_u = u_nu/a_avg

            al_d = 0.5*(1.0 + np.sign((xma_d)))
            al_u = 0.5*(1.0 - np.sign((xma_u)))

            be_d = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_d)))
            be_u = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_u)))

            # Defining the van Leer polynomials
            d_pl = 0.25*(((xma_d+1)**2)*(2-xma_d))
            d_mi = 0.25*(((xma_u-1)**2)*(2+xma_u))

            d_d = al_d*(1+be_d) - be_d*d_pl
            d_u = al_u*(1+be_u) - be_u*d_mi

            cvl_pl = al_d * (1+be_d) * xma_d - be_d * 0.25 * (1+xma_d)**2
            cvl_mi = al_u * (1+be_u) * xma_u + be_u * 0.25 * (1-xma_u)**2

            # Using the van Leer polynoials to define the Liou-Stefan
                polynomials
            cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
            cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)

```

```
            # Defining the mass-flux using the LS formulation
            fm_pl = area * a_avg * cls_pl * rho_d
            fm_mi = area * a_avg * cls_mi * rho_u
            yFluxes[:,j, 0] = fm_pl + fm_mi
            yFluxes[:,j, 1] = fm_pl * u_d + fm_mi * u_u + (d_d*p_d + d_u*
                p_u) * nx*area
            yFluxes[:,j, 2] = fm_pl * v_d + fm_mi * v_u + (d_d*p_d + d_u*
                p_u) * ny*area
            yFluxes[:,j, 3] = fm_pl * h0_d + fm_mi * h0_u

        bc_w = slipwall(dim,r,yfacesA,yfacesN)
        # Lower boundary flux
        area = yfacesA[:,0]
        nx = yfacesN[:,0,0]/area
        ny = yfacesN[:,0,1]/area
        # Upper states
        rho_u = r[0, :, 0]
        u_u = r[1, :, 0]
        v_u = r[2, :, 0]
        p_u = r[3, :, 0]
        h0_u = (gamma/(gamma - 1))*(p_u/rho_u) + 0.5 * (u_u**2 + v_u**2)
        a_u = np.sqrt(gamma*(p_u/rho_u))
        u_nu = u_u*nx + v_u*ny
        # Lower state
        rho_d = bc_w[0]
        p_d = bc_w[1]
        a_d = np.sqrt(gamma*(p_d/rho_d))
        u_nd = -u_nu

        a_avg = 0.5*(a_d + a_u)
        xma_d = u_nd/a_avg
        xma_u = u_nu/a_avg

        al_d = 0.5*(1.0 + np.sign((xma_d)))
        al_u = 0.5*(1.0 - np.sign((xma_u)))

        be_d = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_d)))
        be_u = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_u)))

        # Defining the van Leer polynomials
        d_pl = 0.25*(((xma_d+1)**2)*(2-xma_d))
        d_mi = 0.25*(((xma_u-1)**2)*(2+xma_u))

        d_d = al_d*(1+be_d) - be_d*d_pl
        d_u = al_u*(1+be_u) - be_u*d_mi

        cvl_pl = al_d * (1+be_d) * xma_d - be_d * 0.25 * (1+xma_d)**2
        cvl_mi = al_u * (1+be_u) * xma_u + be_u * 0.25 * (1-xma_u)**2

        # Using the van Leer polynoials to define the Liou-Stefan
            polynomials
        cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
        cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)

        # Defining the mass-flux using the LS formulation
        fm_pl = 0
        fm_mi = 0
        yFluxes[:,0, 0] = fm_pl + fm_mi
```

```
270     yFluxes[:,0, 1] = fm_pl * u_d + fm_mi * u_u + (d_d*p_d + d_u*p_u) *
            nx*area
271     yFluxes[:,0, 2] = fm_pl * v_d + fm_mi * v_u + (d_d*p_d + d_u*p_u) *
            ny*area
272     yFluxes[:,0, 3] = fm_pl * h0_d + fm_mi * h0_u
273
274     # Upper boundary flux
275     area = yfacesA[:,dim[1]-1]
276     nx = yfacesN[:,dim[1]-1,0]/area
277     ny = yfacesN[:,dim[1]-1,1]/area
278     # Lower states
279     rho_d = r[0, :, dim[1]-2]
280     u_d = r[1, :, dim[1]-2]
281     v_d = r[2, :, dim[1]-2]
282     p_d = r[3, :, dim[1]-2]
283     h0_d = (gamma/(gamma - 1))*(p_d/rho_d) + 0.5 * (u_d**2 + v_d**2)
284     a_d = np.sqrt(gamma*(p_d/rho_d))
285     u_nd = u_d*nx + v_d*ny
286     # Upper states
287     rho_u = bc_w[2]
288     p_u = bc_w[3]
289     a_u = np.sqrt(gamma*(p_u/rho_u))
290     u_nu = -u_nd
291
292     a_avg = 0.5*(a_d + a_u)
293     xma_d = u_nd/a_avg
294     xma_u = u_nu/a_avg
295
296     al_d = 0.5*(1.0 + np.sign((xma_d)))
297     al_u = 0.5*(1.0 - np.sign((xma_u)))
298
299     be_d = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_d)))
300     be_u = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_u)))
301
302     # Defining the van Leer polynomials
303     d_pl = 0.25*(((xma_d+1)**2)*(2-xma_d))
304     d_mi = 0.25*(((xma_u-1)**2)*(2+xma_u))
305
306     d_d = al_d*(1+be_d) - be_d*d_pl
307     d_u = al_u*(1+be_u) - be_u*d_mi
308
309     cvl_pl = al_d * (1+be_d) * xma_d - be_d * 0.25 * (1+xma_d)**2
310     cvl_mi = al_u * (1+be_u) * xma_u + be_u * 0.25 * (1-xma_u)**2
311
312     # Using the van Leer polynoials to define the Liou-Stefan
            polynomials
313     cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
314     cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
315
316     # Defining the mass-flux using the LS formulation
317     fm_pl = 0
318     fm_mi = 0
319     yFluxes[:,dim[1]-1, 0] = fm_pl + fm_mi
320     yFluxes[:,dim[1]-1, 1] = fm_pl * u_d + fm_mi * u_u + (d_d*p_d + d_u
            *p_u) * nx*area
321     yFluxes[:,dim[1]-1, 2] = fm_pl * v_d + fm_mi * v_u + (d_d*p_d + d_u
            *p_u) * ny*area
322     yFluxes[:,dim[1]-1, 3] = fm_pl * h0_d + fm_mi * h0_u
```

```
323
324     res = ((xFluxes[1:, :, :] - xFluxes[:-1, :, :]) + (yFluxes[:, 1:,
            :] - yFluxes[:, :-1, :]))
325
326     return res
```

Listing 6: Minmod limiter

```
1  def minmod(a,b):
2      s = np.sign(a) + np.sign(b)
3      return 0.5 * s * np.minimum(np.abs(a), np.abs(b))
```

Listing 7: MUSCL reconstruction

```
1  # State reconstruction at faces
2  def muscl_reco(dim, r):
3
4      # Face state arrays
5      rl = np.zeros_like(r)            # Left face states
6      rr = np.zeros_like(r)            # Right face states
7      rd = np.zeros_like(r)            # Lower face states
8      ru = np.zeros_like(r)            # Upper face states
9
10     for i in range(1, dim[0]-2):
11         drpl0 = r[0,i+1,:] - r[0,i,:]
12         drmi0 = r[0,i,:] - r[0,i-1,:]
13         drpl1 = r[1,i+1,:] - r[1,i,:]
14         drmi1 = r[1,i,:] - r[1,i-1,:]
15         drpl2 = r[2,i+1,:] - r[2,i,:]
16         drmi2 = r[2,i,:] - r[2,i-1,:]
17         drpl3 = r[3,i+1,:] - r[3,i,:]
18         drmi3 = r[3,i,:] - r[3,i-1,:]
19
20         rl[0,i,:] = r[0,i,:] + 0.5*minmod(drpl0,drmi0)
21         rr[0,i,:] = r[0,i,:] - 0.5*minmod(drpl0,drmi0)
22         rl[1,i,:] = r[1,i,:] + 0.5*minmod(drpl1,drmi1)
23         rr[1,i,:] = r[1,i,:] - 0.5*minmod(drpl1,drmi1)
24         rl[2,i,:] = r[2,i,:] + 0.5*minmod(drpl2,drmi2)
25         rr[2,i,:] = r[2,i,:] - 0.5*minmod(drpl2,drmi2)
26         rl[3,i,:] = r[3,i,:] + 0.5*minmod(drpl3,drmi3)
27         rr[3,i,:] = r[3,i,:] - 0.5*minmod(drpl3,drmi3)
28
29     # Left cells
30     drpl0 = r[0,1,:] - r[0,0,:]
31     drmi0 = r[0,0,:] - in_bc(dim,r,mode)[0]
32     drpl1 = r[1,1,:] - r[1,0,:]
33     drmi1 = r[1,0,:] - in_bc(dim,r,mode)[1]
34     drpl2 = r[2,1,:] - r[2,0,:]
35     drmi2 = r[2,0,:] - in_bc(dim,r,mode)[2]
36     drpl3 = r[3,1,:] - r[3,0,:]
37     drmi3 = r[3,0,:] - in_bc(dim,r,mode)[3]
38
39     rl[0,0,:] = r[0,0,:] + 0.5*minmod(drpl0,drmi0)
40     rr[0,0,:] = r[0,0,:] - 0.5*minmod(drpl0,drmi0)
41     rl[1,0,:] = r[1,0,:] + 0.5*minmod(drpl1,drmi1)
42     rr[1,0,:] = r[1,0,:] - 0.5*minmod(drpl1,drmi1)
43     rl[2,0,:] = r[2,0,:] + 0.5*minmod(drpl2,drmi2)
44     rr[2,0,:] = r[2,0,:] - 0.5*minmod(drpl2,drmi2)
```

```
45      rl[3,0,:] = r[3,0,:] + 0.5*minmod(drpl3,drmi3)
46      rr[3,0,:] = r[3,0,:] - 0.5*minmod(drpl3,drmi3)
47
48      # Right cells
49      drpl0 = out_bc(dim,r,mode)[0] - r[0,dim[0]-2,:]
50      drmi0 = r[0,dim[0]-2,:] - r[0,dim[0]-3,:]
51      drpl1 = out_bc(dim,r,mode)[1] - r[1,dim[0]-2,:]
52      drmi1 = r[1,dim[0]-2,:] - r[1,dim[0]-3,:]
53      drpl2 = out_bc(dim,r,mode)[2] - r[2,dim[0]-2,:]
54      drmi2 = r[2,dim[0]-2,:] - r[2,dim[0]-3,:]
55      drpl3 = out_bc(dim,r,mode)[3] - r[3,dim[0]-2,:]
56      drmi3 = r[3,dim[0]-2,:] - r[3,dim[0]-3,:]
57
58      rl[0,dim[0]-2,:] = r[0,dim[0]-2,:] + 0.5*minmod(drpl0,drmi0)
59      rr[0,dim[0]-2,:] = r[0,dim[0]-2,:] - 0.5*minmod(drpl0,drmi0)
60      rl[1,dim[0]-2,:] = r[1,dim[0]-2,:] + 0.5*minmod(drpl1,drmi1)
61      rr[1,dim[0]-2,:] = r[1,dim[0]-2,:] - 0.5*minmod(drpl1,drmi1)
62      rl[2,dim[0]-2,:] = r[2,dim[0]-2,:] + 0.5*minmod(drpl2,drmi2)
63      rr[2,dim[0]-2,:] = r[2,dim[0]-2,:] - 0.5*minmod(drpl2,drmi2)
64      rl[3,dim[0]-2,:] = r[3,dim[0]-2,:] + 0.5*minmod(drpl3,drmi3)
65      rr[3,dim[0]-2,:] = r[3,dim[0]-2,:] - 0.5*minmod(drpl3,drmi3)
66
67      for j in range(1, dim[1]-2):
68          drpl0 = r[0,:,j+1] - r[0,:,j]
69          drmi0 = r[0,:,j] - r[0,:,j-1]
70          drpl1 = r[1,:,j+1] - r[1,:,j]
71          drmi1 = r[1,:,j] - r[1,:,j-1]
72          drpl2 = r[2,:,j+1] - r[2,:,j]
73          drmi2 = r[2,:,j] - r[2,:,j-1]
74          drpl3 = r[3,:,j+1] - r[3,:,j]
75          drmi3 = r[3,:,j] - r[3,:,j-1]
76
77          rd[0,:,j] = r[0,:,j] + 0.5*minmod(drpl0,drmi0)
78          ru[0,:,j] = r[0,:,j] - 0.5*minmod(drpl0,drmi0)
79          rd[1,:,j] = r[1,:,j] + 0.5*minmod(drpl1,drmi1)
80          ru[1,:,j] = r[1,:,j] - 0.5*minmod(drpl1,drmi1)
81          rd[2,:,j] = r[2,:,j] + 0.5*minmod(drpl2,drmi2)
82          ru[2,:,j] = r[2,:,j] - 0.5*minmod(drpl2,drmi2)
83          rd[3,:,j] = r[3,:,j] + 0.5*minmod(drpl3,drmi3)
84          ru[3,:,j] = r[3,:,j] - 0.5*minmod(drpl3,drmi3)
85      # Lower cells
86      drpl0 = r[0,:,1] - r[0,:,0]
87      drmi0 = r[0,:,0] - slipwall(dim,r,yfacesA,yfacesN)[0]
88      drpl1 = r[1,:,1] - r[1,:,0]
89      drmi1 = r[1,:,0] - slipwall(dim,r,yfacesA,yfacesN)[4]
90      drpl2 = r[2,:,1] - r[2,:,0]
91      drmi2 = r[2,:,0] - slipwall(dim,r,yfacesA,yfacesN)[5]
92      drpl3 = r[3,:,1] - r[3,:,0]
93      drmi3 = r[3,:,0] - slipwall(dim,r,yfacesA,yfacesN)[1]
94
95      rd[0,:,0] = r[0,:,0] + 0.5*minmod(drpl0,drmi0)
96      ru[0,:,0] = r[0,:,0] - 0.5*minmod(drpl0,drmi0)
97      rd[1,:,0] = r[1,:,0] + 0.5*minmod(drpl1,drmi1)
98      ru[1,:,0] = r[1,:,0] - 0.5*minmod(drpl1,drmi1)
99      rd[2,:,0] = r[2,:,0] + 0.5*minmod(drpl2,drmi2)
100     ru[2,:,0] = r[2,:,0] - 0.5*minmod(drpl2,drmi2)
101     rd[3,:,0] = r[3,:,0] + 0.5*minmod(drpl3,drmi3)
102     ru[3,:,0] = r[3,:,0] - 0.5*minmod(drpl3,drmi3)
```

```

    # Upper cells
    drpl0 = slipwall(dim,r,yfacesA,yfacesN)[2] - r[0,:,dim[1]-2]
    drmi0 = r[0,:,dim[1]-2] - r[0,:,dim[1]-3]
    drpl1 = 0
    drmi1 = r[1,:,dim[1]-2] - r[1,:,dim[1]-3]
    drpl2 = -2*(r[2,:,dim[1]-2])
    drmi2 = r[2,:,dim[1]-2] - r[2,:,dim[1]-3]
    drpl3 = slipwall(dim,r,yfacesA,yfacesN)[3] - r[3,:,dim[1]-2]
    drmi3 = r[3,:,dim[1]-2] - r[3,:,dim[1]-3]

    rd[0,:,dim[1]-2] = r[0,:,dim[1]-2] + 0.5*minmod(drpl0,drmi0)
    ru[0,:,dim[1]-2] = r[0,:,dim[1]-2] - 0.5*minmod(drpl0,drmi0)
    rd[1,:,dim[1]-2] = r[1,:,dim[1]-2] + 0.5*minmod(drpl1,drmi1)
    ru[1,:,dim[1]-2] = r[1,:,dim[1]-2] - 0.5*minmod(drpl1,drmi1)
    rd[2,:,dim[1]-2] = r[2,:,dim[1]-2] + 0.5*minmod(drpl2,drmi2)
    ru[2,:,dim[1]-2] = r[2,:,dim[1]-2] - 0.5*minmod(drpl2,drmi2)
    rd[3,:,dim[1]-2] = r[3,:,dim[1]-2] + 0.5*minmod(drpl3,drmi3)
    ru[3,:,dim[1]-2] = r[3,:,dim[1]-2] - 0.5*minmod(drpl3,drmi3)

    return rl, rr, rd, ru
```

Listing 8: AUSM (using MUSCL reconstruction)

```
def fluxls_muscl(dim, r, rl, rr, rd, ru):

    # Fluxes initialization
    xFluxes = np.zeros((dim[0], dim[1]-1, 4))
    yFluxes = np.zeros((dim[0]-1, dim[1], 4))

    # X-face fluxes
    for i in range(1, dim[0]-1):
        area = xfacesA[i,:]
        nx = xfacesN[i,:,0]/area
        ny = xfacesN[i,:,1]/area
        # Left states
        rho_l = rr[0, i-1, :]
        u_l = rr[1, i-1, :]
        v_l = rr[2, i-1, :]
        p_l = rr[3, i-1, :]
        h0_l = (gamma/(gamma - 1))*(p_l/rho_l) + 0.5 * (u_l**2 + v_l
            **2)
        a_l = np.sqrt(gamma*(p_l/rho_l))
        u_nl = u_l*nx + v_l*ny
        # Right states
        rho_r = rl[0, i, :]
        u_r = rl[1, i, :]
        v_r = rl[2, i, :]
        p_r = rl[3, i, :]
        h0_r = (gamma/(gamma - 1))*(p_r/rho_r) + 0.5 * (u_r**2 + v_r
            **2)
        a_r = np.sqrt(gamma*(p_r/rho_r))
        u_nr = u_r*nx + v_r*ny

        a_avg = 0.5*(a_l + a_r)
        xma_l = u_nl/a_avg
        xma_r = u_nr/a_avg

```

```
33          al_l = 0.5*(1.0 + np.sign((xma_l)))
34          al_r = 0.5*(1.0 - np.sign((xma_r)))
35
36          be_l = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_l)))
37          be_r = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_r)))
38
39          # Defining the van Leer polynomials
40          d_pl = 0.25*(((xma_l+1)**2)*(2-xma_l))
41          d_mi = 0.25*(((xma_r-1)**2)*(2+xma_r))
42
43          d_l = al_l*(1+be_l) - be_l*d_pl
44          d_r = al_r*(1+be_r) - be_r*d_mi
45
46          cvl_pl = al_l * (1+be_l) * xma_l - be_l * 0.25 * (1+xma_l)**2
47          cvl_mi = al_r * (1+be_r) * xma_r + be_r * 0.25 * (1-xma_r)**2
48
49          # Using the van Leer polynoials to define the Liou-Stefan
                polynomials
50          cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
51          cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
52
53          # Defining the mass-flux using the LS formulation
54          fm_pl = area * a_avg * cls_pl * rho_l
55          fm_mi = area * a_avg * cls_mi * rho_r
56          xFluxes[i,:, 0] = fm_pl + fm_mi
57          xFluxes[i,:, 1] = fm_pl * u_l + fm_mi * u_r + (d_l*p_l + d_r*
                p_r) * nx*area
58          xFluxes[i,:, 2] = fm_pl * v_l + fm_mi * v_r + (d_l*p_l + d_r*
                p_r) * ny*area
59          xFluxes[i,:, 3] = fm_pl * h0_l + fm_mi * h0_r
60
61      # Left boundary flux
62      area = xfacesA[0,:]
63      nx = xfacesN[0,:,0]/area
64      ny = xfacesN[0,:,1]/area
65      # Left states
66      bc_l = in_bc(dim, r, mode)
67      rho_l = bc_l[0]
68      u_l = bc_l[1]
69      v_l = bc_l[2]
70      p_l = bc_l[3]
71      h0_l = (gamma/(gamma - 1))*(p_l/rho_l) + 0.5 * (u_l**2 + v_l**2)
72      a_l = np.sqrt(gamma*(p_l/rho_l))
73      u_nl = u_l*nx + v_l*ny
74      # Right states
75      rho_r = rl[0, 0, :]
76      u_r = rl[1, 0, :]
77      v_r = rl[2, 0, :]
78      p_r = rl[3, 0, :]
79      h0_r = (gamma/(gamma - 1))*(p_r/rho_r) + 0.5 * (u_r**2 + v_r**2)
80      a_r = np.sqrt(gamma*(p_r/rho_r))
81      u_nr = u_r*nx + v_r*ny
82
83      a_avg = 0.5*(a_l + a_r)
84      xma_l = u_nl/a_avg
85      xma_r = u_nr/a_avg
86
87      al_l = 0.5*(1.0 + np.sign((xma_l)))
```

```
88      al_r = 0.5*(1.0 - np.sign((xma_r)))
89
90      be_l = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_l)))
91      be_r = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_r)))
92
93      # Defining the van Leer polynomials
94      d_pl = 0.25*(((xma_l+1)**2)*(2-xma_l))
95      d_mi = 0.25*(((xma_r-1)**2)*(2+xma_r))
96
97      d_l = al_l*(1+be_l) - be_l*d_pl
98      d_r = al_r*(1+be_r) - be_r*d_mi
99
100     cvl_pl = al_l * (1+be_l) * xma_l - be_l * 0.25 * (1+xma_l)**2
101     cvl_mi = al_r * (1+be_r) * xma_r + be_r * 0.25 * (1-xma_r)**2
102
103     # Using the van Leer polynoials to define the Liou-Stefan
            polynomials
104     cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
105     cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
106
107     # Defining the mass-flux using the LS formulation
108     fm_pl = area * a_avg * cls_pl * rho_l
109     fm_mi = area * a_avg * cls_mi * rho_r
110     xFluxes[0,:, 0] = fm_pl + fm_mi
111     xFluxes[0,:, 1] = fm_pl * u_l + fm_mi * u_r + (d_l*p_l + d_r*p_r) *
            nx*area
112     xFluxes[0,:, 2] = fm_pl * v_l + fm_mi * v_r + (d_l*p_l + d_r*p_r) *
            ny*area
113     xFluxes[0,:, 3] = fm_pl * h0_l + fm_mi * h0_r
114
115
116     # Right boundary flux
117     area = xfacesA[dim[0]-1,:]
118     nx = xfacesN[dim[0]-1,:,0]/area
119     ny = xfacesN[dim[0]-1,:,1]/area
120     # Left states
121     rho_l = rr[0, dim[0]-2, :]
122     u_l = rr[1, dim[0]-2, :]
123     v_l = rr[2, dim[0]-2, :]
124     p_l = rr[3, dim[0]-2, :]
125     h0_l = (gamma/(gamma - 1))*(p_l/rho_l) + 0.5 * (u_l**2 + v_l**2)
126     a_l = np.sqrt(gamma*(p_l/rho_l))
127     u_nl = u_l*nx + v_l*ny
128     # Right states
129     bc_r = out_bc(dim, r, mode)
130     rho_r = bc_r[0]
131     u_r = bc_r[1]
132     v_r = bc_r[2]
133     p_r = bc_r[3]
134     h0_r = (gamma/(gamma - 1))*(p_r/rho_r) + 0.5 * (u_r**2 + v_r**2)
135     a_r = np.sqrt(gamma*(p_r/rho_r))
136     u_nr = u_r*nx + v_r*ny
137
138     a_avg = 0.5*(a_l + a_r)
139     xma_l = u_nl/a_avg
140     xma_r = u_nr/a_avg
141
142     al_l = 0.5*(1.0 + np.sign((xma_l)))
```

```
143         al_r = 0.5*(1.0 - np.sign((xma_r)))
144
145         be_l = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_l)))
146         be_r = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_r)))
147
148         # Defining the van Leer polynomials
149         d_pl = 0.25*(((xma_l+1)**2)*(2-xma_l))
150         d_mi = 0.25*(((xma_r-1)**2)*(2+xma_r))
151
152         d_l = al_l*(1+be_l) - be_l*d_pl
153         d_r = al_r*(1+be_r) - be_r*d_mi
154
155         cvl_pl = al_l * (1+be_l) * xma_l - be_l * 0.25 * (1+xma_l)**2
156         cvl_mi = al_r * (1+be_r) * xma_r + be_r * 0.25 * (1-xma_r)**2
157
158         # Using the van Leer polynoials to define the Liou-Stefan
              polynomials
159         cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
160         cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
161
162         # Defining the mass-flux using the LS formulation
163         fm_pl = area * a_avg * cls_pl * rho_l
164         fm_mi = area * a_avg * cls_mi * rho_r
165         xFluxes[dim[0]-1,:, 0] = fm_pl + fm_mi
166         xFluxes[dim[0]-1,:, 1] = fm_pl * u_l + fm_mi * u_r + (d_l*p_l + d_r
              *p_r) * nx*area
167         xFluxes[dim[0]-1,:, 2] = fm_pl * v_l + fm_mi * v_r + (d_l*p_l + d_r
              *p_r) * ny*area
168         xFluxes[dim[0]-1,:, 3] = fm_pl * h0_l + fm_mi * h0_r
169
170
171         # Y-face fluxes
172         for j in range(1, dim[1]-1):
173             area = yfacesA[:,j]
174             nx = yfacesN[:,j,0]/area
175             ny = yfacesN[:,j,1]/area
176             # Lower states
177             rho_d = ru[0, :, j-1]
178             u_d = ru[1, :, j-1]
179             v_d = ru[2, :, j-1]
180             p_d = ru[3, :, j-1]
181             h0_d = (gamma/(gamma - 1))*(p_d/rho_d) + 0.5 * (u_d**2 + v_d
                  **2)
182             a_d = np.sqrt(gamma*(p_d/rho_d))
183             u_nd = u_d*nx + v_d*ny
184             # Upper states
185             rho_u = rd[0, :, j]
186             u_u = rd[1, :, j]
187             v_u = rd[2, :, j]
188             p_u = rd[3, :, j]
189             h0_u = (gamma/(gamma - 1))*(p_u/rho_u) + 0.5 * (u_u**2 + v_u
                  **2)
190             a_u = np.sqrt(gamma*(p_u/rho_u))
191             u_nu = u_u*nx + v_u*ny
192
193             a_avg = 0.5*(a_d + a_u)
194             xma_d = u_nd/a_avg
195             xma_u = u_nu/a_avg
```

```
196
197             al_d = 0.5*(1.0 + np.sign((xma_d)))
198             al_u = 0.5*(1.0 - np.sign((xma_u)))
199
200             be_d = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_d)))
201             be_u = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_u)))
202
203             # Defining the van Leer polynomials
204             d_pl = 0.25*(((xma_d+1)**2)*(2-xma_d))
205             d_mi = 0.25*(((xma_u-1)**2)*(2+xma_u))
206
207             d_d = al_d*(1+be_d) - be_d*d_pl
208             d_u = al_u*(1+be_u) - be_u*d_mi
209
210             cvl_pl = al_d * (1+be_d) * xma_d - be_d * 0.25 * (1+xma_d)**2
211             cvl_mi = al_u * (1+be_u) * xma_u + be_u * 0.25 * (1-xma_u)**2
212
213             # Using the van Leer polynoials to define the Liou-Stefan
                    polynomials
214             cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
215             cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
216
217             # Defining the mass-flux using the LS formulation
218             fm_pl = area * a_avg * cls_pl * rho_d
219             fm_mi = area * a_avg * cls_mi * rho_u
220             yFluxes[:,j, 0] = fm_pl + fm_mi
221             yFluxes[:,j, 1] = fm_pl * u_d + fm_mi * u_u + (d_d*p_d + d_u*
                    p_u) * nx*area
222             yFluxes[:,j, 2] = fm_pl * v_d + fm_mi * v_u + (d_d*p_d + d_u*
                    p_u) * ny*area
223             yFluxes[:,j, 3] = fm_pl * h0_d + fm_mi * h0_u
224
225         bc_w = slipwall(dim,r,yfacesA,yfacesN)
226
227         # Lower boundary flux
228         area = yfacesA[:,0]
229         nx = yfacesN[:,0,0]/area
230         ny = yfacesN[:,0,1]/area
231         # Upper states
232         rho_u = rd[0, :, 0]
233         u_u = rd[1, :, 0]
234         v_u = rd[2, :, 0]
235         p_u = rd[3, :, 0]
236         h0_u = (gamma/(gamma - 1))*(p_u/rho_u) + 0.5 * (u_u**2 + v_u**2)
237         a_u = np.sqrt(gamma*(p_u/rho_u))
238         u_nu = u_u*nx + v_u*ny
239
240         # Lower states
241         u_nd = -u_nu
242
243
244         a_avg = 0.5*(a_d + a_u)
245         xma_d = u_nd/a_avg
246         xma_u = u_nu/a_avg
247
248         al_d = 0.5*(1.0 + np.sign((xma_d)))
249         al_u = 0.5*(1.0 - np.sign((xma_u)))
250
```

```
251       be_d = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_d)))
252       be_u = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_u)))
253
254       # Defining the van Leer polynomials
255       d_pl = 0.25*(((xma_d+1)**2)*(2-xma_d))
256       d_mi = 0.25*(((xma_u-1)**2)*(2+xma_u))
257
258       d_d = al_d*(1+be_d) - be_d*d_pl
259       d_u = al_u*(1+be_u) - be_u*d_mi
260
261       cvl_pl = al_d * (1+be_d) * xma_d - be_d * 0.25 * (1+xma_d)**2
262       cvl_mi = al_u * (1+be_u) * xma_u + be_u * 0.25 * (1-xma_u)**2
263
264       # Using the van Leer polynoials to define the Liou-Stefan
              polynomials
265       cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
266       cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
267
268       # Defining the mass-flux using the LS formulation
269       fm_pl = 0
270       fm_mi = 0
271       yFluxes[:,0, 0] = fm_pl + fm_mi
272       yFluxes[:,0, 1] = fm_pl * u_d + fm_mi * u_u + (d_d*p_d + d_u*p_u) *
              nx*area
273       yFluxes[:,0, 2] = fm_pl * v_d + fm_mi * v_u + (d_d*p_d + d_u*p_u) *
              ny*area
274       yFluxes[:,0, 3] = fm_pl * h0_d + fm_mi * h0_u
275
276
277       # Upper boundary flux
278       area = yfacesA[:,dim[1]-1]
279       nx = yfacesN[:,dim[1]-1,0]/area
280       ny = yfacesN[:,dim[1]-1,1]/area
281       # Lower states
282       rho_d = ru[0, :, dim[1]-2]
283       u_d = ru[1, :, dim[1]-2]
284       v_d = ru[2, :, dim[1]-2]
285       p_d = ru[3, :, dim[1]-2]
286       h0_d = (gamma/(gamma - 1))*(p_d/rho_d) + 0.5 * (u_d**2 + v_d**2)
287       a_d = np.sqrt(gamma*(p_d/rho_d))
288       u_nd = u_d*nx + v_d*ny
289       # Upper states
290       u_nu = -u_nd
291
292       a_avg = 0.5*(a_d + a_u)
293       xma_d = u_nd/a_avg
294       xma_u = u_nu/a_avg
295
296       al_d = 0.5*(1.0 + np.sign((xma_d)))
297       al_u = 0.5*(1.0 - np.sign((xma_u)))
298
299       be_d = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_d)))
300       be_u = -np.maximum(0.0, 1.0-np.floor(np.abs(xma_u)))
301
302       # Defining the van Leer polynomials
303       d_pl = 0.25*(((xma_d+1)**2)*(2-xma_d))
304       d_mi = 0.25*(((xma_u-1)**2)*(2+xma_u))
305
```

```
306     d_d = al_d*(1+be_d) - be_d*d_pl
307     d_u = al_u*(1+be_u) - be_u*d_mi
308 
309     cvl_pl = al_d * (1+be_d) * xma_d - be_d * 0.25 * (1+xma_d)**2
310     cvl_mi = al_u * (1+be_u) * xma_u + be_u * 0.25 * (1-xma_u)**2
311 
312     # Using the van Leer polynoials to define the Liou-Stefan
            polynomials
313     cls_pl = np.maximum(0.0, cvl_pl + cvl_mi)
314     cls_mi = np.minimum(0.0, cvl_pl + cvl_mi)
315 
316     # Defining the mass-flux using the LS formulation
317     fm_pl = 0
318     fm_mi = 0
319     yFluxes[:,dim[1]-1, 0] = fm_pl + fm_mi
320     yFluxes[:,dim[1]-1, 1] = fm_pl * u_d + fm_mi * u_u + (d_d*p_d + d_u
            *p_u) * nx*area
321     yFluxes[:,dim[1]-1, 2] = fm_pl * v_d + fm_mi * v_u + (d_d*p_d + d_u
            *p_u) * ny*area
322     yFluxes[:,dim[1]-1, 3] = fm_pl * h0_d + fm_mi * h0_u
323 
324     res = ((xFluxes[1:, :, :] - xFluxes[:-1, :, :]) + (yFluxes[:, 1:,
            :] - yFluxes[:, :-1, :]))
325 
326     return res
```

Listing 9: Initializing the solution variables and time step calculations

```
1  # Initializing the primitive variables' (r) and conservative variables'
       (q) arrays
2 
3  if not restart:
4      r = np.ones((4,dim[0]-1,dim[1]-1))
5      q = np.ones((4,dim[0]-1,dim[1]-1))
6 
7      r[0] = r[0] * rho_inf
8      r[1] = r[1] * u_inf
9      r[2] = r[2] * v_inf
10     r[3] = r[3] * p_stat
11 
12     q[0] = q[0] * r[0]
13     q[1] = q[1] * r[0] * r[1]
14     q[2] = q[2] * r[0] * r[2]
15     q[3] = q[3] * (r[3]/(gamma-1) + (0.5*r[0]*(r[2]**2+r[3]**2)))
16 
17 else:
18     res_data = np.load('solfiles/solution_iter_130000.npz')
19     r = res_data['r']
20     q = res_data['q']
21     res_arr = res_data['res']
22     last_res = res_arr[-1]
23 
24 # Time-step calculation
25 def tcalc(dim, r, facenormals, faceareas, cellvols, cells):
26     delta_t = np.zeros((dim[0]-1,dim[1]-1))
27     for j in range(dim[1]-1):
28         for i in range(dim[0]-1):
29             lambda_a = np.zeros(4)
```

```
            vel_p = np.array([r[1, i, j], r[2, i, j]])
            p_k = r[3, i, j]
            rho_k = r[0, i, j]
            a_k = np.sqrt(gamma*p_k/rho_k)

            for k in range(4):
                area = faceareas[cells[i,j]][k]
                n = facenormals[cells[i,j]][k]
                vn = np.dot(vel_p,n)
                lambda_a[k] = np.abs(vn) + (a_k * area)


            t_by_v = cfl/np.sum(np.abs(lambda_a))
            delta_t[i,j] = t_by_v*cellvols[cells[i,j]]

    return delta_t
```

Listing 10: Main loop

```
# Main Loop
if not restart:
    resnorm = []
else:
    resnorm = res_data['res'].tolist()

if not restart:
    itr = 0
else:
    itr = len(resnorm)

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

if not restart:
    time_itr = 0.0
else:
    time_itr = last_time

if high_recon:
    rl,rr,rd,ru = muscl_reco(dim,r)

if high_recon:
    res = fluxls_muscl(dim, r, rl, rr, rd, ru)                    #
        Initialization of residual vector
else:
    res = fluxls(dim, r)

# Scales for convergence residuals
scale1 = rho_inf * u_inf
scale2 = rho_inf * u_inf * u_inf
scale3 = rho_inf * u_inf * u_inf
scale4 = rho_inf * u_inf * (((gamma*p_stat)/(p_stat*rho_inf))+(0.5*
    u_inf*u_inf))


while np.abs(c_res) > tol and itr <= 500000:
```

```
38
39      delta_t = tcalc(dim, r, facenormals, faceareas, cellvols, cells)
40      del_t = np.min(delta_t)
41      q_new = np.zeros_like(q)
42
43      # Updating the conservative variables
44      for k in range(4):
45          q_new[k] = q[k] - (del_t / vols_2d) * res[:,:,k]
46
47      # Updating the primitive variables
48      r[0] = q_new[0]
49      r[1] = q_new[1]/q_new[0]
50      r[2] = q_new[2]/q_new[0]
51      r[3] = (gamma-1) * (q_new[3] - 0.5*((q_new[1]**2 + q_new[2]**2)/
            q_new[0]))
52
53      if high_recon:
54          rl,rr,rd,ru = muscl_reco(dim,r)
                                                    # Updating the face
                reconstructed states
55
56      q = np.copy(q_new)
57      if high_recon:
58          res = fluxls_muscl(dim, r, rl, rr, rd, ru)
                                                # Recalculating residuals
59      else:
60          res = fluxls(dim,r)
61
62      resnorm.append(0.0)
63
64      resnorm[itr] = np.sum((res[:,:,0]/scale1)**2+(res[:,:,1]/scale2)
            **2+(res[:,:,2]/scale3)**2+(res[:,:,3]/scale4)**2)
65
66
67      if itr == 0:
68          resnorm0 = resnorm[itr]
69          c_res = 1
70      else:
71          resnorm0 = resnorm[0]
72          stab = 1e-12
73          c_res = (resnorm[itr]) / (resnorm0 + stab)
74
75      time_itr += del_t
76
77      # if itr%50 == 0 or np.abs(c_res) <= tol:                    #
            Printing residuals every 50 iterations and after convergence
78      #     print(f"Residual is {np.abs(c_res)} at time {time_itr}s
            corresponding to iteration {itr}.")         # SM: Only for
            debugging. Remove later.
79      print(f"Residual is {np.abs(c_res)} at time {time_itr}s
            corresponding to iteration {itr}.")         # SM: Only for
            debugging. Remove later.
80      itr += 1
81
82      if itr%5000 == 0 or np.abs(c_res) <= tol:                  #
            Saving every 5000 iterations and after convergence
83          resnorm_arr = np.array(resnorm)
84          filename = f"solfiles/solution_iter_{itr}.npz"
```

```
85            np.savez(filename, q=q, r=r, res=resnorm_arr)
86            print(f"Saved solution to {filename}")
```