

Project report 2: Bow Shock

Subject: Advanced Computational Fluid Dynamics (AAE6201-20241-A)

Date: 04/12/2024

1 Problem description

Sod shock tube and bow shock problem are all known as Riemann problem, the governing equation form Riemann problem are:

1. Conservation of mass:

$$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u)}{\partial x} = 0 \quad (1)$$

2. Conservation of momentum:

$$\frac{\partial(\rho u)}{\partial t} + \frac{\partial(\rho u^2 + p)}{\partial x} = 0 \quad (2)$$

3. Conservation of energy:

$$\frac{\partial(\rho E)}{\partial t} + \frac{\partial(\rho u E + p)}{\partial x} = 0 \quad (3)$$

where ρ is the density, u is the velocity, p is the pressure, and E is the total energy per unit volume. The original equations can be written as follow:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} = 0 \quad (4)$$

in which

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho E \end{bmatrix}, \mathbf{F} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u E + pu \end{bmatrix} \quad (5)$$

2 Requirements

1. Use second-order reconstruction
2. Use different Riemann solvers (at least include HLL and AUSM)
3. Use first-order difference formula for time discretization (monitor the change in density to check convergence)
4. Compare the numerical solution with the empirical shock shape

3 Finite Volume Method and its reconstruction

In Finite Volume Method the governing equation as estimated after space integral, use Gaussian's divergence law, one have:

$$\frac{\partial}{\partial t} \int_{\Omega} U d\Omega + \oint_S \vec{F} \cdot \vec{n} dS = 0 \quad (6)$$

define

$$\bar{U} = \frac{1}{\Omega} \int_{\Omega} U d\Omega \quad (7)$$

In FVM, we only care about the average value of U in the cell and use the assign this value to the central point of the cell, so we can drop the bar and the equation becomes:

$$\frac{\partial U}{\partial t} + \frac{1}{\Omega} \sum_{\text{faces}} F_n \Delta S = 0 \quad (8)$$

according to this integral equation, the fluxes on the interface between cells are needed, in structured mesh, fluxes on the interface are reconstructed with average value of nearby cells. However, the flow direction has not been determined. Similar to the regulation in FDM, the number of stencil points on the upwind side must be larger. Evaluating fluxes direction, U on the interface:

$$U_{I+1/2}^L = g^L(U_{I-1}, U_I, U_{I+1}), \quad U_{I+1/2}^R = g^R(U_I, U_{I+1}, U_{I+2}) \quad (9)$$

Second-order upwind scheme in FVM has no difference with its conservative form in FDM.

$$U_L = U_I + \frac{1}{2}(U_I - U_{I-1}), \quad U_R = U_{I+1} - \frac{1}{2}(U_{I+2} - U_{I+1}) \quad (10)$$

Other discretization scheme can be formed in similar way for example third order MUSL scheme

$$U_{I+1/2}^L = U_I + s_1/4 \left[(1 - s_1/3)(U_I - U_{I-1}) + (1 + s_1/3)(U_{I+1} - U_I) \right] \quad (11)$$

$$U_{I+1/2}^R = U_{I+1} - s_2/4 \left[(1 - s_2/3)(U_{I+2} - U_{I+1}) + (1 + s_2/3)(U_{I+1} - U_I) \right] \quad (12)$$

in which

$$s_1 = \frac{2(U_I - U_{I-1})(U_{I+1} - U_I) + \varepsilon}{(U_I - U_{I-1})^2 + (U_{I+1} - U_I)^2 + \varepsilon} \quad (13)$$

$$s_2 = \frac{2(U_{I+1} - U_I)(U_{I+2} - U_{I+1}) + \varepsilon}{(U_{I+1} - U_I)^2 + (U_{I+2} - U_{I+1})^2 + \varepsilon} \quad (14)$$

in this project, only second-order upwind scheme will be used.

4 Interface flux

Consider a 2-D mesh,

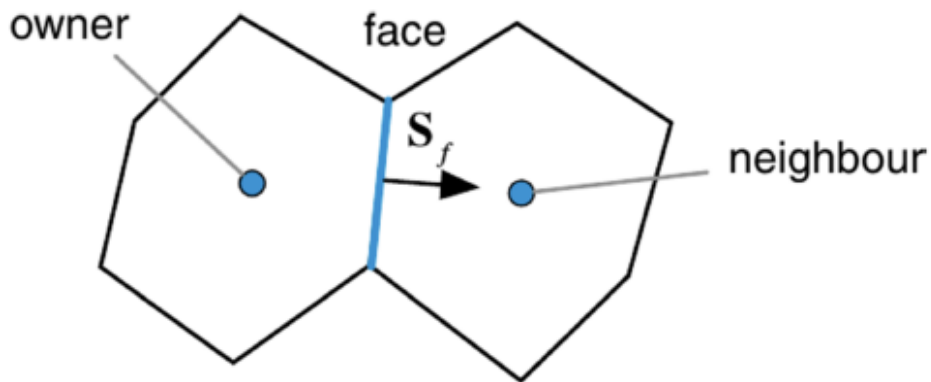


Figure 1: 2D mesh and their relationship

the cell calculated in this step is called owner, and the cell next to it was called neighbour, the face between owner and neighbour is very important in FVM, especially in 2-D blunt-body problem. Because in 2-D blunt-body problem, the flow is transonic, the equation is hybrid hyperbolic-parabolic, the number and direction characteristic lines on the face are not determined.

4.1 HLL scheme

As shown is Figure 2, the black lines represent the faces of the cell, and the dots in the middle of the black lines represent the face centers, where Riemann problems exist. The region surrounded by blue dashed lines represents the influencing region, where the values will change as time progresses. For time steps that are relatively small, the influencing region exists only in the small peripheral area outside the faces of the cell. The other region represents the unaffected region, where the values do not change with time. Now consider the cell represented by the red dashed lines.

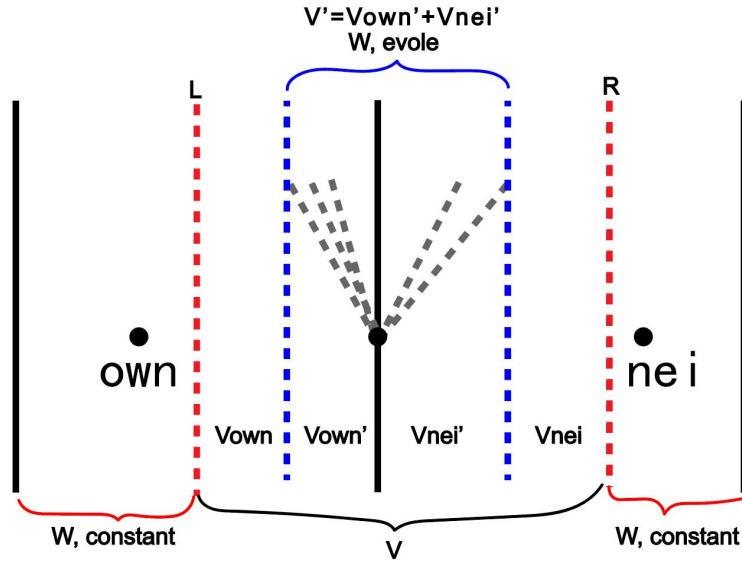


Figure 2: HLL Riemann solver

Consider an original equation:

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} = 0 \quad (15)$$

Integrated along x:

$$\frac{d}{dt} \int_{x_L}^{x_R} U(x, t) dx = F(U(x_L, t)) - F(U(x_R, t)) \quad (16)$$

Integrated along t

$$\int_{x_L}^{x_R} U(x, T) dx - \int_{x_L}^{x_R} U(x, 0) dx = \int_0^T [F(U(x_L, t)) - F(U(x_R, t))] dt \quad (17)$$

when $t = 0$, U does not change with position, and from time 0 to T , the influence or single has not propagate to L and R . The time integral equation can be written as

$$\int_{x_L}^{x_R} U(x, T) dx = (x_R - x_L)U + T(F_L - F_R) \quad (18)$$



Form the perspective of signal propagating, the region shown in Figure can be classified into three different parts.

$$\int_{x_L}^{x_R} U(x, T) dx = \int_{x_{own}} U(x, T) dx + \int_{x_{bet}} U(x, T) dx + \int_{x_{nei}} U(x, T) dx \quad (19)$$

The left-hand side becomes

$$\int_{x_L}^{x_R} U(x, T) dx = (TS_L - x_L)U_L + \int_{x_{bet}} U(x, T) dx + (x_R - TS_R)U_R \quad (20)$$

where S_L and S_R represent the propagating speed form the surface to the left and the right. Combine equation 14 and 12, we have

$$\int_{TS_L}^{TS_R} U(x, T) dx = T(S_R U_R - S_L U_L + F_L - F_R) \quad (21)$$

In HLL scheme, the average value of U between L and R can be calculated with the equation below:

$$U_{HLL} = \frac{1}{T(S_R - S_L)} \int_{TS_L}^{TS_R} U(x, T) dx = \frac{S_R U_R - S_L U_L + F_L - F_R}{S_R - S_L} \quad (22)$$

Equation 22 defines the case where the Riemann problem partially propagates to the left or partially to the right. If all characteristic values propagate to the left or to the right, the HLL reduces to a standard interpolation format. So the latest question left is How to estimate S_L and S_R , in most cases we cannot use the exact solution; otherwise, the computational cost is the same as the Godunov scheme. An approximation method was established.

$$S_L = \min(u_L - a_L, \tilde{u} - \tilde{a}) S_R = \min(u_R + a_R, \tilde{u} + \tilde{a}) \quad (23)$$

4.2 AUSM scheme

The AUSM scheme split fluxes based on the Mach number direction on the face of the grid [1]. AUSM splitting assumes that for faces of grids, their influencing regions depend on the upstream and downstream. In supersonic conditions, grid information comes entirely from the upstream. In subsonic conditions, grid face information comes from the grids on the left and right sides. Therefore, AUSM splitting separates the contributions of grids faces into contributions from the left and right grids (upstream and downstream) based on the magnitude of the eigenvalues. At the face interface:

$$M^+ = \begin{cases} M & \text{if } M > 1 \\ \frac{(M+1)^2}{4} & \text{if } |M| \leq 1 \\ 0 & \text{if } M < -1 \end{cases} \quad M^- = \begin{cases} 0 & \text{if } M > 1 \\ -\frac{(M-1)^2}{4} & \text{if } |M| \leq 1 \\ M & \text{if } M < -1 \end{cases} \quad (24)$$

The AUSM scheme considers that the convection and pressure in the equation variables come from different contributions. Therefore, when dealing with fluxes, it is necessary to handle the convection contribution and pressure contribution separately, of the flux F is separated as the convective and pressure terms.

$$F = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u E + pu \end{bmatrix} = \begin{bmatrix} \rho u \\ \rho u^2 \\ \rho u H \end{bmatrix} + \begin{bmatrix} 0 \\ p \\ 0 \end{bmatrix} = F^c + F^p \quad (25)$$

Similarly, we have

$$p = \frac{c^2}{\gamma_f} \rho_f = p^+ + p^- \quad (26)$$

$$p^+ = \begin{cases} 1 & \text{if } M > 1 \\ \frac{(M+1)^2}{4}(2-M) & \text{if } |M| \leq 1 \\ 0 & \text{if } M < -1 \end{cases}, p^- = \begin{cases} 0 & \text{if } M > 1 \\ \frac{(M-1)^2}{4}(2+M) & \text{if } |M| \leq 1 \\ 1 & \text{if } M < -1 \end{cases} \quad (27)$$

5 Instancing in 2D problem

Instancing the numerical scheme in 2D Riemann problem with AUSM as an example [2]. General governing equation:

$$\frac{\partial U}{\partial t} + \frac{1}{\Omega} \sum_{\text{faces}} F_n \Delta S = 0 \quad (28)$$

in which

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho E \end{bmatrix}, \mathbf{F} = \begin{bmatrix} \rho u_n \\ \rho u u_n + p n_x \\ \rho v u_n + p n_y \\ \rho E u_n + p u_n \end{bmatrix} \quad (29)$$

In a structured grid as shown in Figure 3, the area of grid and length of face can be calculated from the coordinate of grid points.

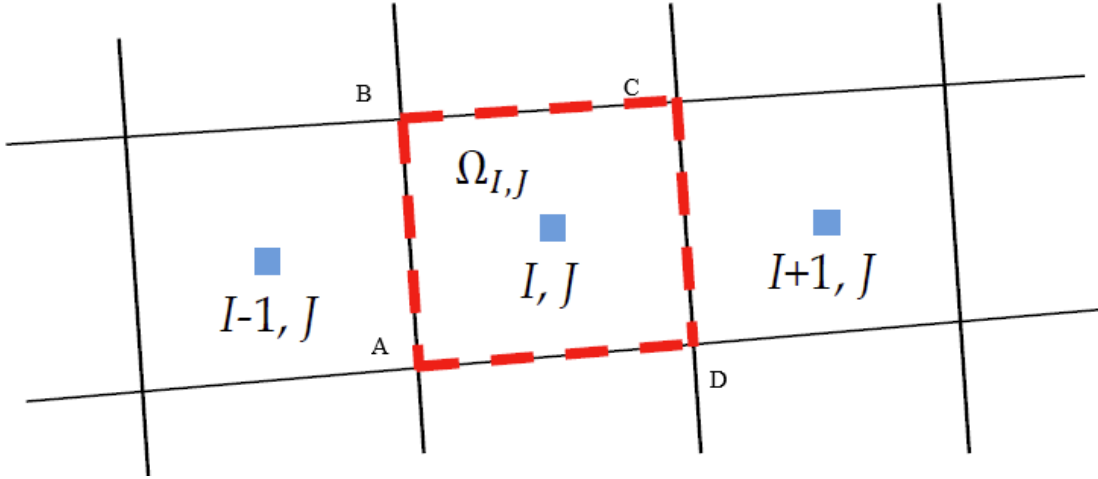


Figure 3: Regions in sod shock tube with different space discretization schemes

$$\begin{aligned} \Omega_{IJ} &= \frac{1}{2} |\vec{r}_{AC} \times \vec{r}_{BD}| \\ &= \frac{1}{2} [(x_C - x_A)(y_D - y_B) - (x_D - x_B)(y_C - y_A)] \\ &= \frac{1}{2} (\Delta x_{AC} \Delta y_{BD} - \Delta x_{BD} \Delta y_{AC}) \end{aligned} \quad (30)$$

and length of edge



$$\Delta S_{AB} = \text{mag}(\vec{r}_{AB}) = \sqrt{\Delta x_{AB}^2 + \Delta y_{AB}^2} \quad (31)$$

Consider the flux vector \mathbf{F} on the face CD, separate it as the convective and pressure terms.

$$F_{CD} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho v u \\ \rho E u + p u \end{bmatrix} = \begin{bmatrix} \rho u \\ \rho u^2 \\ \rho v u \\ \rho E u + p u \end{bmatrix} + \begin{bmatrix} 0 \\ p_{CD} \\ 0 \\ 0 \end{bmatrix} = F_{CD}^c + F_{CD}^p \quad (32)$$

the specific calculation method of F_{CD}^c is dependent on the flow direction of u .

$$F_{CD}^c = \begin{bmatrix} \rho u \\ \rho u^2 \\ \rho v u \\ \rho H u \end{bmatrix} = u \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho H \end{bmatrix} = u_{CD} \Psi_{CD} \quad (33)$$

it should be noted that, the u_{CD} on surface CD is unknown, it has to be calculated utilize nearby cell information. Take second order upwind scheme as an example.

$$U_{CD} = \begin{cases} U_L = U_I + \frac{1}{2}(U_I - U_{I-1}) & \text{if } u_{CD} \geq 0 \\ U_R = U_{I+1} - \frac{1}{2}(U_{I+2} - U_{I+1}) & \text{if } u_{CD} < 0 \end{cases} \quad (34)$$

Further,

$$u_{CD} = M_{CD} a_{CD} = (M^+ + M^-) \frac{a_L + a_R}{2} \quad (35)$$

$$M^+ = \begin{cases} M_L & \text{if } M_L > 1 \\ \frac{(M_L+1)^2}{4} & \text{if } |M_L| \leq 1 \\ 0 & \text{if } M_L < -1 \end{cases} \quad M^- = \begin{cases} 0 & \text{if } M_R > 1 \\ -\frac{(M_R-1)^2}{4} & \text{if } |M_R| \leq 1 \\ M_R & \text{if } M_R < -1 \end{cases} \quad (36)$$

where

$$M_L = \frac{u_L}{a_L} \quad M_R = \frac{u_R}{a_R} \quad (37)$$

Similarly, we have

$$p_{CD} = p^+ p_L + p^- p_R \quad (38)$$

$$p^+ = \begin{cases} 1 & \text{if } M_L > 1 \\ \frac{(M_L+1)^2}{4} (2 - M_L) & \text{if } |M_L| \leq 1 \\ 0 & \text{if } M_L < -1 \end{cases}, \quad p^- = \begin{cases} 0 & \text{if } M_R > 1 \\ \frac{(M_R-1)^2}{4} (2 + M_R) & \text{if } |M_R| \leq 1 \\ 1 & \text{if } M_R < -1 \end{cases} \quad (39)$$

Similarly, we have

$$G_{CD} = \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 + p \\ \rho E v + p v \end{bmatrix} = \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 \\ \rho E v + p v \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ p_{CD} \\ 0 \end{bmatrix} = G_{CD}^c + G_{CD}^p \quad (40)$$

the specific calculation method of G_{CD}^c is dependent on the flow direction of v .

$$G_{CD}^c = \begin{bmatrix} \rho v \\ \rho u v \\ \rho v^2 \\ \rho H v \end{bmatrix} = v \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho H \end{bmatrix} = v_{CD} \Psi_{CD} \quad (41)$$

it should be noticed that, the method to calculate Mach number changed

$$M_L = \frac{v_L}{a_L} \quad M_R = \frac{v_R}{a_R} \quad (42)$$

substitute all equations back to general equation we have:

$$U_{IJ}^{n+1} = U_{IJ}^n + \frac{\Delta t}{\Omega_{IJ}} \sum F_n \Delta S \quad (43)$$

where

$$F_n = F \times n_x + G \times n_y \quad (44)$$

5.1 Boundary conditions

1. Upper boundary: Supersonic inflow

For supersonic in flow, all eigenvalues have the same sign. All values are determined by upstream values, in this case, the value in ghost cells next to upper boundary should be set to free stream value in every time step.

$$\mathbf{U}_{in} = \begin{bmatrix} \rho_\infty \\ \rho_\infty u_\infty \\ \rho_\infty v_\infty \\ \rho_\infty E_\infty \end{bmatrix} \quad (45)$$

2. Left and right boundaries: Supersonic outflow

Similarly, at the outlet boundary, all values are determined by upstream values. In this case, the value in ghost cells should be set to next boundary cell value in every time step. What should be concerned is that the flow after shock wave is subsonic, as for subsonic flow, one characteristic line point into the computational domain so external pressure has to be specified. The primitive variables at the farfield boundary are obtained from

$$\begin{aligned} p_{boundary} &= p_{farfield} \\ \rho_{boundary} &= \rho_{inside} + (p_{boundary} - p_{inside})/a_{ref}^2 \\ u_{boundary} &= u_{inside} + n_x(p_{boundary} - p_{inside})/\rho_{ref} \times a_{ref} \\ v_{boundary} &= v_{inside} + n_y(p_{boundary} - p_{inside})/\rho_{ref} \times a_{ref} \end{aligned} \quad (46)$$

where p_{ref} and a_{ref} represent a reference state. The reference state is normally set equal to the state at the interior point. The values in point a are determined from the freestream state. Physical properties in the ghost cells can be obtained by linear extrapolation from the states.

3. Lower boundary: Inviscid wall

Inviscid wall boundary is equivalent to a symmetry boundary. In symmetry boundary, the normal velocity is zero and the gradient of the tangential velocity normal to the boundary is zero. In this case, every time step the value in ghost cell are set to boundary cell and velocity normal to boundary are set to 0.

6 Code establishing

6.1 Julia language

Julia [3] was designed for high performance. Julia programs automatically compile to efficient native code via LLVM, and support multiple platforms. With Julia, the computational efficiency can be improved significantly.

6.2 Computational grid

The computational grid has 160 (circumferential) \times 80 (wall normal) cells in PLOT3D format. Data format for plot3D comes from this 1990s manual pdf-Plot3D is a simple way to construct a structured grid using 4 points to define a cell in 2D and 8 points for 3D. The ASCII format Plot3D mesh can be simply read via python library plot3d distributed by NASA.

```
1 @pyimport plot3d as p3d
2
3 # read mesh
4 block = p3d.read_plot3D("Cylinder.dat", binary = false)
5
6 IMAX, JMAX, KMAX = size(block[1].X)
7 X_coordinate = block[1].X
8 Y_coordinate = block[1].Y
```

6.3 Trans-sonic outflow

For trans-sonic outflow boundary, a switch defined according to local Mach number should be established [4].

```
1  for j in 3:JMAX+1
2      rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[IMAX+1,j,:])
3      a_out=sqrt(gamma * p_out/rho_out)
4      M_out=sqrt(u_out^2+v_out^2)/a_out
5      n=normal_vector(IMAX+2,j,IMAX+1,j)
6      if M_out >= 1
7          U_bar[end, j, :] = U_bar[IMAX+1, j, :]
8          U_bar[end-1, j, :] = U_bar[IMAX+1, j, :]
9      else
10         p=p_b
11         rho=rho_out+(p_b-p_out)/a_out^2
12         u=u_out+(p_b-p_out)*n[1]/(rho_out*a_out)
13         v=v_out+(p_b-p_out)*n[2]/(rho_out*a_out)
14         P_U_bar=[rho,rho*u,rho*v, 0.5* rho * (u^2 + v^2) + p / (gamma - 1)]
15         U_bar[end-1, j, :]=2 .*P_U_bar.- U_bar[IMAX+1, j, :]
16         U_bar[end-1, j, :]=4 .*P_U_bar.- 3 .*U_bar[IMAX+1, j, :]
17     end
18     rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[3, j, :])
19     a_out=sqrt(gamma * p_out/rho_out)
20     M_out=sqrt(u_out^2+v_out^2)/a_out
21     n=normal_vector(3, j, 4, j)
22     if M_out >= 1
23         U_bar[1, j, :] = U_bar[3, j, :]
24         U_bar[2, j, :] = U_bar[3, j, :]
25     else
26         p=p_b
27         rho=rho_out+(p_b-p_out)/a_out^2
28         u=u_out+(p_b-p_out)*n[1]/(rho_out*a_out)
```




```

29     v=v_out+(p_b-p_out)*n[2]/(rho_out*a_out)
30     P_U_bar=[rho,rho*u,rho*v, 0.5* rho * (u^2 + v^2) + p / (gamma - 1)]
31     U_bar[2, j,:]=2 .* P_U_bar.- U_bar[3, j,: ]
32     U_bar[1, j,:]=4 .* P_U_bar.- 3 .*U_bar[3, j,: ]
33 end
34 end

```

6.4 Parallelization

This code supports multi-thread parallelization based on OpenMP. In Julia, *Base* library contains a module called *Threads*. This is a macro to execute a for loop in parallel. The iteration space is distributed to coarse-grained tasks. This policy can be specified by the schedule argument. The execution of the loop waits for the evaluation of all iterations. So, OpenMP multi-threads can be called using following code.

```

1 using Base.Threads: @threads, nthreads
2 for t in 0:dt:t_end
3     println ( "t=$t")
4     for i in 3:IMAX+1
5         @threads for j in 3:JMAX+1

```

In this way, the code can be run in parallel through commend line:

```

1 $ julia -t 16 code.jl

```

As shown in Figure 4, OpenMP based program has a very high efficiency as for Computational Fluid Dynamics calculation load, the usage of CPU threads went up to 95%.



Figure 4: CPU usage

7 Simulation details

7.1 Initialization

In order to accelerate the simulation process, the density field was initialized using precalculated density after shock wave of around 0.11 kg/m^3 . So as to pressure field, but velocity field was initialized as zero. With several attempts, overall time step was set to 1×10^{-6} , simulation lasted 0.1s.

7.2 Simulation results

Figure 5, shows the simulation results of density, pressure and velocity field 0.1 second initial condition. For convenience, the two outflow are was set to supersonic outflow. The shape of shock wave is very clear, especially in density field, the density of free stream is really small, in shock wave, the fluid was strongly compressed so there is a rapid increase of density in shock wave. After shock, the fluid entered a low pressure region, velocity went up, pressure went down and density decreased.

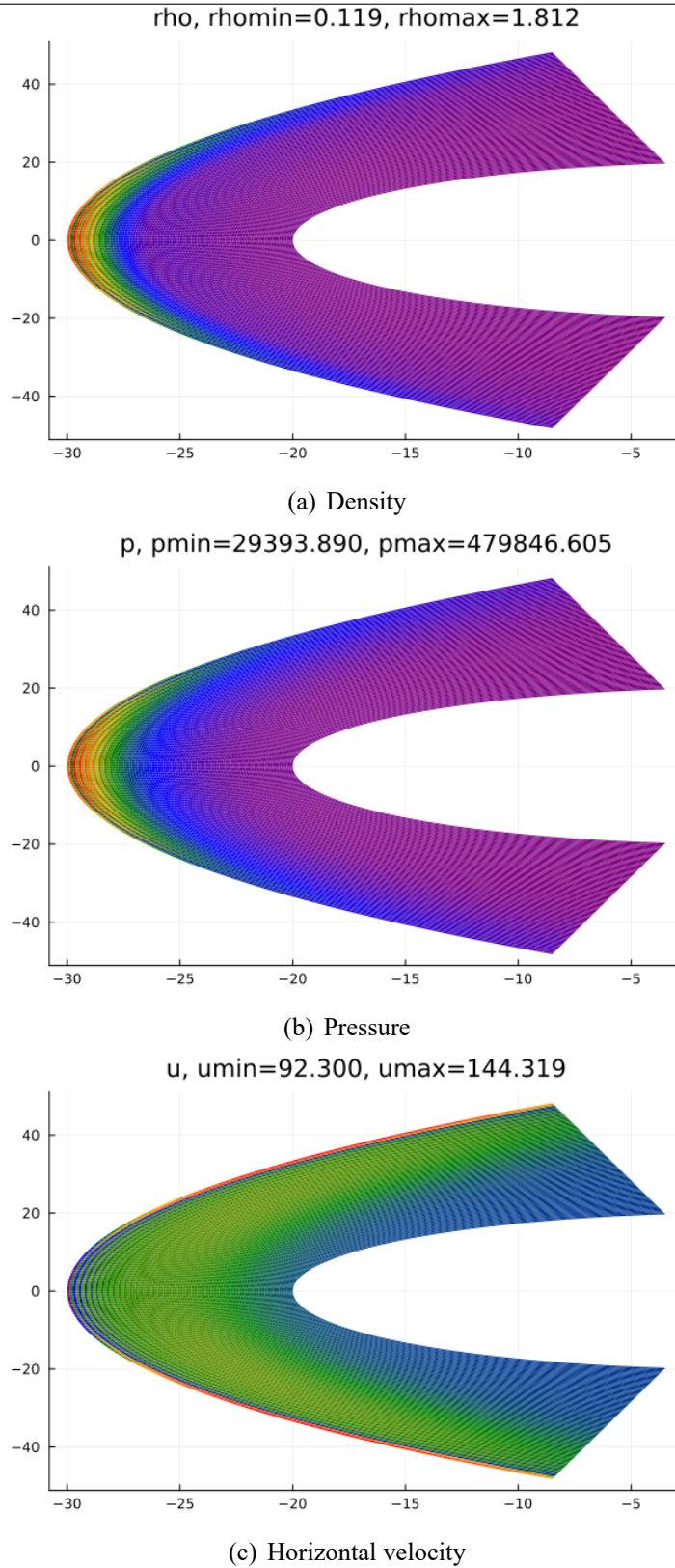


Figure 5: Density, pressure and velocity field based on AUSM scheme

As shown in Figure 6, in compare with the empirical shock wave shape, our numerical scheme and calculational program shew a pretty good agreement with the empirical shock wave shape

at Ma 8. Since the empirical shock wave is steady-state, but our simulation is not long enough to reach steady-state condition, the error is reasonable.

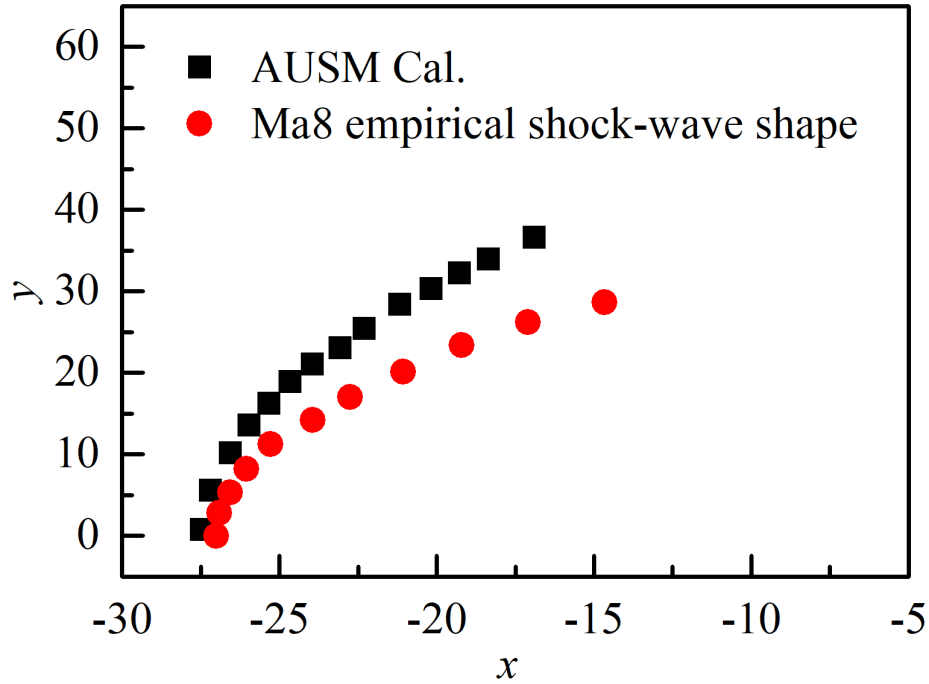


Figure 6: Regions in sod shock tube with different space discretization schemes

Further reach shows that, the stability and robustness of HLL scheme, is not as good as that of AUSM schemes. During the simulation, negative pressure always occurs and its frequency depend on initial condition. Finally, a shock wave shape at 0.005s was obtained, as can be seen in Figure 7. However, it still shows a great vary of pressure and velocity at the front edge.

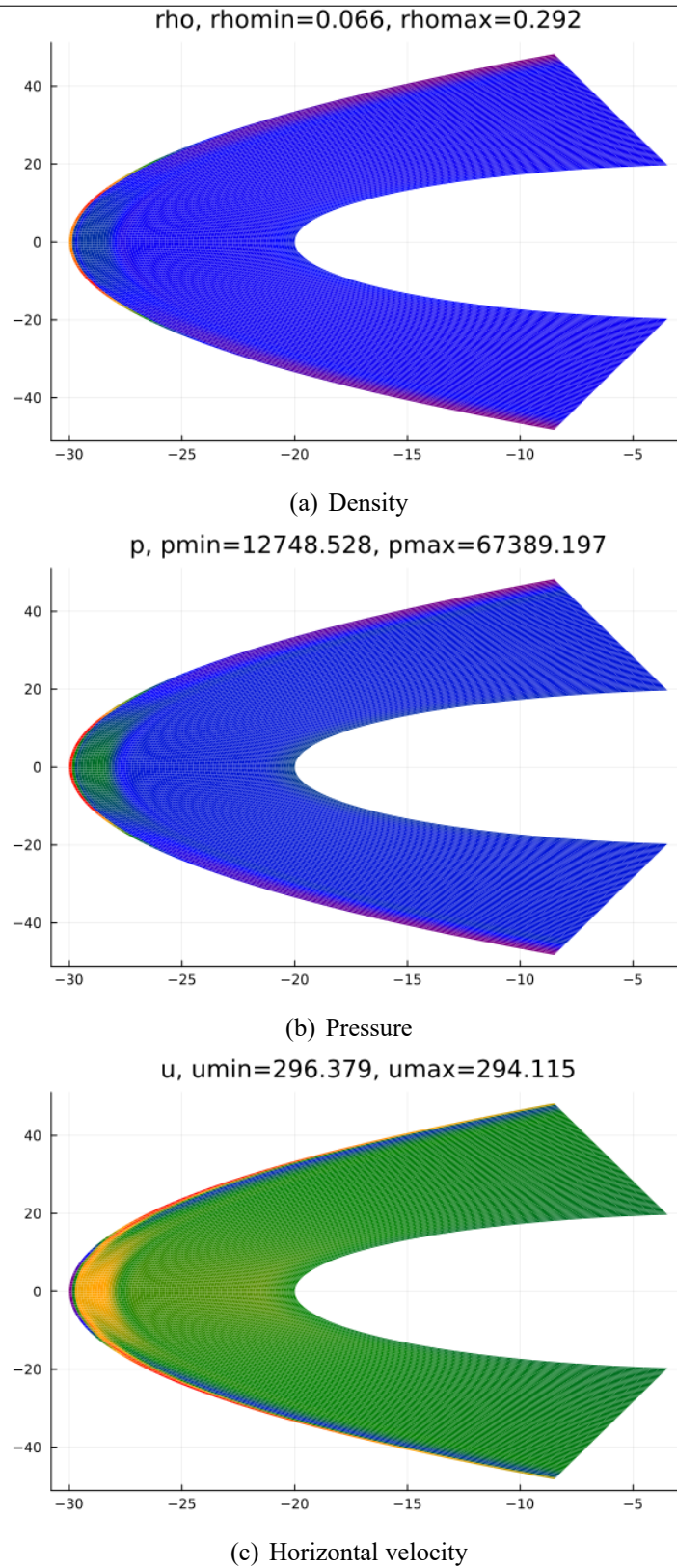


Figure 7: Density, pressure and velocity field based on HLL scheme

References

- [1] M. Liou, Ten years in the making - AUSM-family, in: 15th AIAA Computational Fluid Dynamics Conference, American Institute of Aeronautics and Astronautics, Anaheim, CA, U.S.A., 2001. doi:10.2514/6.2001-2521.
- [2] B. Xie, X. Deng, Z. Sun, F. Xiao, A hybrid pressure–density-based Mach uniform algorithm for 2D Euler equations on unstructured grids by using multi-moment finite volume method, Journal of Computational Physics 335 (2017) 637–663. doi:10.1016/j.jcp.2017.01.043.
- [3] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A fresh approach to numerical computing, SIAM Review 59 (1) (2017) 65–98. doi:10.1137/141000671.
- [4] J. E. Matsson, An introduction to ansys fluent 2023, Sdc Publications, 2023.

8 Appendix

8.1 AUSM based 2D Riemann solver

```

1  #!/usr/local/bin/julia
2  #coded based on julia v1.9
3  #copyright © ZHANG Ting, polyU, Hong Kong
4  #e-mail ting123.zhang@connect.polyu.hk
5  #updated at 05/Dec/2024
6  using CSV
7  using DataFrames
8  using Plots
9  using PyCall
10 using LinearAlgebra
11 using Colors
12 using Printf
13 @pyimport plot3d as p3d
14 using Base.Threads: @threads, nthreads
15
16 # confirm the number of threads
17 println ( "Number of threads: ", nthreads () )
18 # read mesh
19 block = p3d.read_plot3D( "Cylinder.dat", binary = false )
20
21 IMAX, JMAX, KMAX = size(block[1].X)
22 X_coordinate = block [1]. X
23 Y_coordinate = block [1]. Y
24 U_bar = zeros(Float64, IMAX+3, JMAX+3, 4)
25
26
27 gamma = 1.4
28 R = 8.314
29 M_IG = 0.029
30 c_v = 717

```



```
31 R_G=R/M_IG
32 # inflow outflow
33 M_inf = 8.1
34 T_inf = 63.73
35 p_inf = 370.6
36 rho_inf=p_inf/(R_G*T_inf)
37 a_inf=sqrt(gamma*R_G*T_inf)
38 u_inf=M_inf*a_inf
39
40 rho_after_shock=0.116
41 p_after_shock=28465
42 M_after_shock=0.392
43 T_after_shock=849.75
44 a_after_shock=sqrt(gamma*R_G*T_after_shock)
45 u_after_shock=a_after_shock*M_after_shock
46 # p_b=101325
47 p_b=p_inf
48
49 T_total=T_inf*(1+((gamma-1)/2)*M_inf^2)
50 p_total=p_inf*(1+((gamma-1)/2)*M_inf^2)^(gamma/(gamma-1))
51
52 # initial_conditions
53 function initial_conditions ()
54     rho = fill (rho_after_shock , IMAX+3, JMAX+3)
55     u = fill (0,IMAX+3, JMAX+3) #M_inf * sqrt(gamma * R_G * T_inf)
56     v = zeros(IMAX+3, JMAX+3)
57     p = fill (p_after_shock , IMAX+3, JMAX+3)
58     return rho, u, v, p
59 end
60
61
62 function U_flux_decompose(U)
63     rho = U[1]
64     u = U[2] / U[1]
65     v = U[3] / U[1]
66     p = (U[4] - 0.5 * rho * (u^2 + v^2)) * (gamma - 1)
67     return rho, u, v, p
68 end
69
70 function area(I, J)
71     I=I-2
72     J=J-2
73     p1 = [X_coordinate[I, J][1], Y_coordinate[I, J][1],0]
74     p2 = [X_coordinate[I, J+1][1], Y_coordinate[I, J+1][1],0]
75     p3 = [X_coordinate[I+1, J+1][1], Y_coordinate[I+1, J+1][1],0]
76     p4 = [X_coordinate[I+1, J][1], Y_coordinate[I+1, J][1],0]
77     r_1 = p1 - p3
```



```
78     r_2 = p2 - p4
79     cross_product = cross(r_1,r_2)
80     magnitude = norm(cross_product)
81     return magnitude
82 end
83
84 function length(a, b, c, d)
85     a=a-2
86     b=b-2
87     c=c-2
88     d=d-2
89     p1 = [X_coordinate[a, b][1], Y_coordinate[a, b][1]]
90     p2 = [X_coordinate[c, d][1], Y_coordinate[c, d][1]]
91     p = p1 - p2
92     mag = norm(p)
93     return mag
94 end
95
96 function AUSM(U_L, U_R, n)
97     rho_L, u_L, v_L, p_L = U_flux_decompose(U_L)
98     rho_R, u_R, v_R, p_R = U_flux_decompose(U_R)
99     if p_L/rho_L*p_R/rho_R < 0
100         println ( "rho=",rho_L, "u=",u_L, "v=",v_L, "p=",p_L)
101         println ( "rho=",rho_R, "u=",u_R, "v=",v_R, "p=",p_R)
102         println (U_L)
103         println (U_R)
104     end
105     a_L = sqrt(gamma * p_L/rho_L)
106     a_R = sqrt(gamma * p_R/rho_R)
107     a = (a_L + a_R) / 2
108     M_L = (u_L*n[1]+v_L*n[2]) / a_L
109     M_R = (u_R*n[1]+v_R*n[2]) / a_R
110
111     M_plus = M_L > 1 ? M_L : M_L < -1 ? 0 : (M_L + 1)^2 / 4
112     M_minus = M_R > 1 ? 0 : M_R < -1 ? M_R : -(M_R - 1)^2 / 4
113
114     u = (M_plus + M_minus) * a
115     if u >= 0
116         PHI =U_L
117         PHI[end]=p_L * (gamma / (gamma - 1)) + (rho_L * 0.5 * (u_L^2 + v_L^2))
118     else
119         PHI =U_R
120         PHI[end]=p_R * (gamma / (gamma - 1)) + (rho_R * 0.5 * (u_R^2 + v_R^2))
121     end
122
123     F_c = u * PHI
124
```




```
125 p_plus = M_L > 1 ? 1 : M_L < -1 ? 0 : (M_L + 1)^2 * (2 - M_L) / 4
126 p_minus = M_R > 1 ? 0 : M_R < -1 ? (M_R - 1)^2 * (2 + M_R) / 4 : 1
127
128 p_ASUM = p_plus * p_L + p_minus * p_R
129
130 F_p = [0, p_ASUM*n[1], p_ASUM*n[2], 0]
131
132 # M_L = v_L / a_L
133 # M_R = v_R / a_R
134
135 # M_plus = M_L > 1 ? M_L : M_L < -1 ? 0 : (M_L + 1)^2 / 4
136 # M_minus = M_R > 1 ? 0 : M_R < -1 ? M_R : -(M_R - 1)^2 / 4
137
138 # v = (M_plus + M_minus) * a
139 # if v >= 0
140 #     PHI = U_L
141 #     PHI[end] = p_L * (gamma / (gamma - 1)) * (rho_L * 0.5 * (u_L^2 + v_L^2))
142 # else
143 #     PHI = U_R
144 #     PHI[end] = p_R * (gamma / (gamma - 1)) * (rho_R * 0.5 * (u_R^2 + v_R^2))
145 # end
146
147 # G_c = v * PHI
148
149 # p_plus = M_L > 1 ? 1 : M_L < -1 ? 0 : (M_L + 1)^2 * (2 - M_L) / 4
150 # p_minus = M_R > 1 ? 0 : M_R < -1 ? (M_R - 1)^2 * (2 + M_R) / 4 : 1
151
152 # p_ASUM = p_plus * p_L + p_minus * p_R
153
154 # G_p = [0, 0, p_ASUM, 0]
155
156 return F_c, F_p #*n[1] + G_p*n[2]
157 end
158
159 function normal_vector(a, b, c, d)
160     a=a-2
161     b=b-2
162     c=c-2
163     d=d-2
164     p1 = [X_coordinate[a, b][1], Y_coordinate[a, b][1]]
165     p2 = [X_coordinate[c, d][1], Y_coordinate[c, d][1]]
166     p = p1 - p2
167     magnitude = norm(p)
168     unit_vector = p / magnitude
169     return unit_vector
170 end
171
```



```
172 function main(dt, t_end)
173     # initialize
174     rho, u, v, p = initial_conditions ()
175     U_bar = cat(rho, rho .* u, rho .* v, 0.5 .* rho .* (u.^2 .+ v.^2) .+ p ./ (
        gamma - 1); dims=3)
176     for t in 0:dt:t_end
177         println ("t=$t")
178         for i in 3:IMAX+1
179             @threads for j in 3:JMAX+1
180                 U = U_bar[i, j, :]
181                 omega = area(i, j)
182                 # println (i,j)
183                 U_L = U_bar[i, j+1,:] .+ 0.5 .* (U_bar[i, j+1,:] .- U_bar[i, j+2,:])
184                 U_R = U_bar[i, j,:] .+ 0.5 .* (U_bar[i, j,:] .- U_bar[i, j-1,:])
185                 # println (U_bar[i, j,:])
186                 # println (U_bar[i, j-1,:])
187                 # left face
188                 S_left = length(i, j+1, i+1, j+1)
189                 n_left = normal_vector(i, j+1, i, j)
190                 F_c, F_p = AUSM(U_R, U_L, n_left)
191                 F_left = F_c .+ F_p
192                 # if j==JMAX
193                 #     println (F_left)
194                 # end
195
196                 U_L = U_bar[i, j,:] .+ 0.5 .* (U_bar[i, j,:] .- U_bar[i, j+1,:])
197                 U_R = U_bar[i, j-1,:] .+ 0.5 .* (U_bar[i, j-1,:] .- U_bar[i, j-2,:])
198                 # println ("right face")
199                 # right face
200                 S_right = length(i, j, i+1, j)
201                 n_right = normal_vector(i, j, i, j+1)
202                 F_c, F_p = AUSM(U_L, U_R, n_right)
203                 F_right = F_c .+ F_p
204                 # if j==JMAX
205                 #     println (F_right)
206                 # end
207
208                 U_L = U_bar[i, j,:] .+ 0.5 .* (U_bar[i, j,:] .- U_bar[i-1, j,:])
209                 U_R = U_bar[i+1, j,:] .+ 0.5 .* (U_bar[i+1, j,:] .- U_bar[i+2, j,:])
210                 # println (U_R)
211                 # upper face
212                 S_upper = length(i+1, j, i+1, j+1)
213                 n_upper = normal_vector(i+1, j, i, j)
214                 F_c, F_p = AUSM(U_L, U_R, n_upper)
215                 F_upper = F_c .+ F_p
216
217                 U_L = U_bar[i-1, j,:] .+ 0.5 .* (U_bar[i-1, j,:] .- U_bar[i-2, j,:])
```



```
218     U_R = U_bar[i, j, :] .+ 0.5 .* (U_bar[i, j, :] .- U_bar[i+1, j, :])
219     # println (U_R)
220     # down face
221     S_down = length(i, j, i, j+1)
222     n_down = normal_vector(i, j, i+1, j)
223     F_c, F_p = AUSM(U_R, U_L, n_down)
224     F_down = F_c .+ F_p
225
226     U_temp = U .- dt / omega .* (S_left .* F_left .+ S_right .* F_right
227     .+ S_upper .* F_upper .+ S_down .* F_down)
228     U_bar[i, j, :] = U_temp
229     # if j==JMAX
230     #     println (F_left)
231     #     println (dt / omega .* (S_left .* F_left .+ S_right .* F_right
232     .+ S_upper .* F_upper .+ S_down .* F_down))
233     # end
234 end
235
236 #define some monitor
237 println ("rho=", U_bar[80,40,1], "u=", U_bar[80,40,2]/U_bar[80,40,1], "p=", (
238     U_bar[80,40,4] - 0.5 * U_bar[80,40,1] * ((U_bar[80,40,2]/U_bar[80,40,1])
239     ^2 + (U_bar[80,40,3]/U_bar[80,40,1])^2)) * (gamma - 1))
240 println ("rho=", U_bar[80,end,1], "u=", U_bar[80,end,2]/U_bar[80,end,1], "p=", (
241     U_bar[80,end,4] - 0.5 * U_bar[80,end,1] * ((U_bar[80,end,2]/U_bar[80,end,1])
242     ^2 + (U_bar[80,end,3]/U_bar[80,end,1])^2)) * (gamma - 1))
243 println ("rho=", U_bar[80,end-3,1], "u=", U_bar[80,end-3,2]/U_bar[80,end-3,1], "
244     p=", (U_bar[80,end-3,4] - 0.5 * U_bar[80,end-3,1] * ((U_bar[80,end-3,2]/
245     U_bar[80,end-3,1])^2 + (U_bar[80,end-3,3]/U_bar[80,end-3,1])^2)) * (
246     gamma - 1))
247
248 # boundary conditions
249 for i in 3:IMAX+1
250     # U_bar[i, 2, :] = U_bar[i, 3, :]
251     # Inviscid wall
252     n_right = normal_vector(i, 3, i, 4)
253     n_upper = normal_vector(i+1, 3, i, 3)
254     A = [1 0 0 0; 0 n_right[1] n_right[2] 0; 0 n_upper[1] n_upper[2] 0; 0 0
255         0 1]
256     B1 = [1 0 0 0; 0 0 0 0; 0 0 1 0; 0 0 0 1]
257     B2 = [1 0 0 0; 0 -1 0 0; 0 0 1 0; 0 0 0 1]
258     # println (A)
259     C = [1 0 0 0; 0 n_right[1] n_upper[1] 0; 0 n_right[2] n_upper[2] 0; 0 0
260         0 1]
261     # coe_matrix = C * B1 * A
262     # println (U_bar[i, 3, :])
263     # println (B2 * A * U_bar[i, 3, :])
```



```
254     U_bar[i, 2,:]= C * B2 * A*U_bar[i, 3,:]  
255     U_bar[i, 1,:]= C * B2 * A*U_bar[i, 4,:]  
256     # P_U_bar = coe_matrix * U_bar[i, 3,:]  
257     # U_bar[i, 1,:]= 3 .* U_bar[i, 2,:] - 2 .* P_U_bar  
258 end  
259 # Supersonic inflow  
260 for i in 3:IMAX+1  
261     rho_in = p_inf / (T_inf* R_G)  
262     u_in = M_inf * sqrt(gamma * R_G * T_inf)  
263     v_in = 0  
264     p_in = p_inf  
265     U_bar[i, end,1]=U_bar[i, end-1,1]=rho_in  
266     U_bar[i, end,2]=U_bar[i, end-1,2]=rho_in*u_in  
267     U_bar[i, end,3]=U_bar[i, end-1,3]=rho_in*v_in  
268     U_bar[i, end,4]=U_bar[i, end-1,4]=0.5 * rho_in * (u_in^2 + v_in^2) +  
        p_in / (gamma - 1)  
269     # println (U_bar[i, end,:])  
270 end  
271 # trans-sonic outlet boundary condition  
272 for j in 3:JMAX+1  
273     U_bar[1, j ,:] = U_bar[3, j ,:]  
274     U_bar[2, j ,:] = U_bar[3, j ,:]  
275     U_bar[end, j ,:] = U_bar[IMAX+1, j ,:]  
276     U_bar[end-1, j ,:] = U_bar[IMAX+1, j ,:]  
277     #=  
278     rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[IMAX+1, j ,:])  
279     a_out=sqrt(gamma * p_out/rho_out)  
280     M_out=sqrt(u_out^2+v_out^2)/a_out  
281     n=normal_vector(IMAX+2,j,IMAX+1,j)  
282     if M_out >= 1  
283         U_bar[end, j ,:] = U_bar[IMAX+1, j ,:]  
284         U_bar[end-1, j ,:] = U_bar[IMAX+1, j ,:]  
285     else  
286         p=p_b  
287         rho=rho_out+(p_b-p_out)/a_out^2  
288         u=u_out+(p_b-p_out)*n[1]/(rho_out*a_out)  
289         v=v_out+(p_b-p_out)*n[2]/(rho_out*a_out)  
290         P_U_bar=[rho,rho*u,rho*v, 0.5* rho * (u^2 + v^2) + p / (gamma - 1)]  
291         U_bar[end-1, j ,:] = 2 .* P_U_bar .- U_bar[IMAX+1, j ,:]  
292         U_bar[end-1, j ,:] = 4 .* P_U_bar .- 3 .* U_bar[IMAX+1, j ,:]  
293     end  
294     rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[3, j ,:])  
295     a_out=sqrt(gamma * p_out/rho_out)  
296     M_out=sqrt(u_out^2+v_out^2)/a_out  
297     n=normal_vector(3,j,4,j)  
298     if M_out >= 1  
299         U_bar[1, j ,:] = U_bar[3, j ,:]
```



```
300         U_bar[2, j, :] = U_bar[3, j, :]  
301     else  
302         p=p_b  
303         rho=rho_out+(p_b-p_out)/a_out^2  
304         u=u_out+(p_b-p_out)*n[1]/(rho_out*a_out)  
305         v=v_out+(p_b-p_out)*n[2]/(rho_out*a_out)  
306         P_U_bar=[rho,rho*u,rho*v, 0.5* rho * (u^2 + v^2) + p / (gamma - 1)]  
307         U_bar[2, j, :]=2 .* P_U_bar ./ U_bar[3, j, :]  
308         U_bar[1, j, :]=4 .* P_U_bar ./ 3 .*U_bar[3, j, :]  
309     end  
310     =#  
311 end  
312  
313 end  
314  
315 return U_bar  
316 end  
317  
318 U_bar = main(1e-6, 1e-1)  
319  
320 rho_final = U_bar[:, :,1]  
321 u_final = U_bar[:, :,2] ./ rho_final  
322 v_final = U_bar[:, :,3] ./ rho_final  
323 p_final = (U_bar[:, :,4] ./ 0.5 .* rho_final .* (u_final.^2 .+ v_final.^2)) .* (  
324     gamma - 1)  
325 # println ( rho_final )  
326 # plot colored cells  
327 function plot_colored_square (x_coords, y_coords, color)  
328     plot !(x_coords, y_coords, seriestype = :shape, fillcolor = color, linecolor = :  
329         transparent )  
330 end  
331 # get color  
332 function get_color(value, min_value, delta)  
333     return cgrad(:rainbow)[(value - min_value) / delta]  
334 end  
335 # plotrho  
336 # normalized_rho = ( rho_final ./ minimum(rho_final)) ./ (maximum(rho_final) -  
337     minimum(rho_final))  
338 p_rho = plot( figsize = (800, 800), legend=false )  
339 count_greater_than_10 = 0  
340 for i in 3:IMAX+1  
341     for j in 3:JMAX+1  
342         X = X_coordinate  
343         Y = Y_coordinate  
344         # define cell  
345         I_coordinate=i-2
```



```
344     J_coordinate=j-2
345     cell_x = [X[I_coordinate, J_coordinate ], X[I_coordinate+1, J_coordinate ], X
               [I_coordinate+1, J_coordinate+1], X[I_coordinate, J_coordinate+1], X[
               I_coordinate, J_coordinate ]]
346     cell_y = [Y[I_coordinate, J_coordinate ], Y[I_coordinate+1, J_coordinate ], Y
               [I_coordinate+1, J_coordinate+1], Y[I_coordinate, J_coordinate+1], Y[
               I_coordinate, J_coordinate ]]
347
348     Delta = maximum(rho_final[3:IMAX+1,3:JMAX+1]) - minimum(rho_final[3:
               IMAX+1,3:JMAX+1]) == 0 ? 1 : maximum(rho_final[3:IMAX+1,3:JMAX+1])
               - minimum(rho_final[3:IMAX+1,3:JMAX+1])
349     color = get_color( rho_final [i, j ], minimum(rho_final[3:IMAX+1,3:JMAX+1]),
               Delta)
350
351     # plot everycell
352     plot_colored_square ( cell_x , cell_y , color )
353     global count_greater_than_10
354     if rho_final [i, j] > 1
355         count_greater_than_10 += 1
356     end
357 end
358 end
359
360 println ( "Number of rho values greater than 10: ", count_greater_than_10 )
361
362 colorbar_ticks = range(minimum(rho_final[3:IMAX+1,3:JMAX+1]), stop=maximum(
               rho_final[3:IMAX+1,3:JMAX+1]), length=11)
363 plot !(p_rho, color=:rainbow, colorbar=:right , colorbar_ticks = colorbar_ticks ,
               colorbar_tick_labels =[@sprintf( "%0.2f", x) for x in colorbar_ticks ])
364 # Colorbar(p_rho, pltobj)
365 title !(p_rho, "rho, rhomin=$( @sprintf( \"%0.3f\", minimum(rho_final[3:IMAX+1,3:JMAX
               +1])) ), rhomax=$( @sprintf( \"%0.3f\", maximum(rho_final[3:IMAX+1,3:JMAX+1])) )")
366 savefig(p_rho, "rho.png")
367
368
369 # plotp
370 p_p=plot( figsize = (800, 800), legend=false )
371 for i in 3:IMAX+1
372     for j in 3:JMAX+1
373         X = X_coordinate
374         Y = Y_coordinate
375         I_coordinate=i-2
376         J_coordinate=j-2
377         cell_x = [X[I_coordinate, J_coordinate ], X[I_coordinate+1, J_coordinate ], X
               [I_coordinate+1, J_coordinate+1], X[I_coordinate, J_coordinate+1], X[
               I_coordinate, J_coordinate ]]
378         cell_y = [Y[I_coordinate, J_coordinate ], Y[I_coordinate+1, J_coordinate ], Y
```



```
[I_coordinate+1, J_coordinate+1], Y[I_coordinate, J_coordinate+1], Y[
I_coordinate, J_coordinate]]
379
380 Delta = maximum(p_final[3:IMAX+1,3:JMAX+1]) - minimum(p_final[3:IMAX
+1,3:JMAX+1]) == 0 ? 1 : maximum(p_final[3:IMAX+1,3:JMAX+1]) -
minimum(p_final[3:IMAX+1,3:JMAX+1])
381 color = get_color( p_final [ i, j ], minimum(p_final[3:IMAX+1,3:JMAX+1]),
Delta)
382
383 # plot everycell
384 plot_colored_square ( cell_x , cell_y , color )
385 end
386 end
387 title !(p_p,"p, pmin=$( @sprintf("%.3f", minimum(p_final[3:IMAX+1,3:JMAX+1])) ),
pmax=$( @sprintf("%.3f", maximum(p_final[3:IMAX+1,3:JMAX+1])) )" )
388 savefig (p_p,"p.png")
389
390
391 ## plot u
392 p_u=plot( figsize = (800, 800),legend=false)
393 for i in 3:IMAX+1
394     for j in 3:JMAX+1
395         X = X_coordinate
396         Y = Y_coordinate
397         I_coordinate=i-2
398         J_coordinate=j-2
399         cell_x = [X[I_coordinate, J_coordinate ], X[I_coordinate+1, J_coordinate ], X
[I_coordinate+1, J_coordinate+1], X[I_coordinate, J_coordinate+1], X[
I_coordinate, J_coordinate]]
400         cell_y = [Y[I_coordinate, J_coordinate ], Y[I_coordinate+1, J_coordinate ], Y
[I_coordinate+1, J_coordinate+1], Y[I_coordinate, J_coordinate+1], Y[
I_coordinate, J_coordinate]]
401
402 Delta = maximum(u_final[3:IMAX+1,3:JMAX+1]) - minimum(u_final[3:IMAX
+1,3:JMAX+1]) == 0 ? 1 : maximum(u_final[3:IMAX+1,3:JMAX+1]) -
minimum(u_final[3:IMAX+1,3:JMAX+1])
403 color = get_color( u_final [ i, j ], minimum(u_final[3:IMAX+1,3:JMAX+1]),
Delta)
404
405 # plot everycell
406 plot_colored_square ( cell_x , cell_y , color )
407 end
408 end
409 title !(p_u,"u, umin=$( @sprintf("%.3f", abs(minimum(u_final[3:IMAX+1,3:JMAX+1])) ),
umax=$( @sprintf("%.3f", abs(minimum(u_final[3:IMAX+1,3:JMAX+1])) )" )
410 savefig (p_u,"u.png")
```

8.2 HLL based 2D Riemann solver

```

1  #!/usr/local/bin/julia
2  #coded based on julia v1.9
3  #copyright © ZHANG Ting, polyU, Hong Kong
4  #e-mail ting123.zhang@connect.polyu.hk
5  #updated at 05/Dec/2024
6  using CSV
7  using DataFrames
8  using Plots
9  using PyCall
10 using LinearAlgebra
11 using Colors
12 using Printf
13 @pyimport plot3d as p3d
14 using Base.Threads: @threads, nthreads
15
16
17 println ( "Number of threads: ", nthreads () )
18
19 block = p3d.read_plot3D( "Cylinder.dat", binary = false )
20
21 IMAX, JMAX, KMAX = size(block[1].X)
22 X_coordinate = block [1]. X
23 Y_coordinate = block [1]. Y
24 U_bar = zeros(Float64, IMAX+3, JMAX+3, 4)
25
26
27 gamma = 1.4
28 R = 8.314
29 M_IG = 0.029
30 c_v = 717
31 R_G=R/M_IG
32
33 M_inf = 8.1
34 T_inf = 63.73
35 p_inf = 370.6
36 rho_inf=p_inf/(R_G*T_inf)
37 a_inf=sqrt(gamma*R_G*T_inf)
38 u_inf=M_inf*a_inf
39
40 rho_after_shock=0.116
41 p_after_shock=28465
42 M_after_shock=0.392
43 T_after_shock=849.75
44 a_after_shock=sqrt(gamma*R_G*T_after_shock)
45 u_after_shock=a_after_shock*M_after_shock

```




```
46 # p_b=101325
47 p_b=p_inf
48
49 T_total=T_inf*(1+((gamma-1)/2)*M_inf^2)
50 p_total=p_inf*(1+((gamma-1)/2)*M_inf^2)^(gamma/(gamma-1))
51
52
53 function initial_conditions ()
54     rho = fill (rho_after_shock , IMAX+3, JMAX+3)
55     u = fill (0, IMAX+3, JMAX+3) #M_inf * sqrt(gamma * R_G * T_inf)
56     v = zeros(IMAX+3, JMAX+3)
57     p = fill (p_after_shock , IMAX+3, JMAX+3)
58     return rho, u, v, p
59 end
60
61
62 function U_flux_decompose(U)
63     rho = U[1]
64     u = U[2] / U[1]
65     v = U[3] / U[1]
66     p = (U[4] - 0.5 * rho * (u^2 + v^2)) * (gamma - 1)
67     return rho, u, v, p
68 end
69
70 function area(I, J)
71     I=I-2
72     J=J-2
73     p1 = [X_coordinate[I, J][1], Y_coordinate[I, J][1],0]
74     p2 = [X_coordinate[I, J+1][1], Y_coordinate[I, J+1][1],0]
75     p3 = [X_coordinate[I+1, J+1][1], Y_coordinate[I+1, J+1][1],0]
76     p4 = [X_coordinate[I+1, J][1], Y_coordinate[I+1, J][1],0]
77     r_1 = p1 - p3
78     r_2 = p2 - p4
79     cross_product = cross(r_1,r_2)
80     magnitude = norm(cross_product)
81     return magnitude
82 end
83
84 function length(a, b, c, d)
85     a=a-2
86     b=b-2
87     c=c-2
88     d=d-2
89     p1 = [X_coordinate[a, b][1], Y_coordinate[a, b][1]]
90     p2 = [X_coordinate[c, d][1], Y_coordinate[c, d][1]]
91     p = p1 - p2
92     mag = norm(p)
```



```
93     return mag
94 end
95
96 function HLL(U_L, U_R, n)
97     rho_L, u_L, v_L, p_L = U_flux_decompose(U_L)
98     rho_R, u_R, v_R, p_R = U_flux_decompose(U_R)
99     if p_L/rho_L*p_R/rho_R < 0
100         println ( "rho=",rho_L, "u=",u_L, "v=",v_L, "p=",p_L)
101         println ( "rho=",rho_R, "u=",u_R, "v=",v_R, "p=",p_R)
102         println (U_L)
103         println (U_R)
104     end
105     a_L = sqrt (gamma * p_L/rho_L)
106     a_R = sqrt (gamma * p_R/rho_R)
107     a_tilde = (a_L + a_R) / 2
108     un_L=u_L*n[1]+v_L*n[2]
109     un_R=u_R*n[1]+v_R*n[2]
110     u_tilde=(un_L+un_R)/2
111
112     S_L=min(un_L-a_L,u_tilde-a_tilde)
113     S_R=max(un_R+a_R,u_tilde+a_tilde)
114
115     PHI_L=U_L
116     PHI_L[end]=p_L * (gamma / (gamma - 1)) + (rho_L * 0.5 * (u_L^2 + v_L^2))
117     F_L=un_L .* PHI_L .+ [0, p_L*n[1], p_L*n[2], 0]
118
119     PHI_R=U_R
120     PHI_R[end]=p_R * (gamma / (gamma - 1)) + (rho_R * 0.5 * (u_R^2 + v_R^2))
121     F_R=un_R .* PHI_R .+ [0, p_R*n[1], p_R*n[2], 0]
122
123     if S_L >= 0
124         F = F_L
125     elseif S_R <= 0
126         F = F_R
127     else
128         F=(S_R .* F_L .- S_L .* F_R .+ S_L*S_R .* (U_R-U_L)) ./ (S_R-S_L)
129     end
130
131     return F
132 end
133
134 function normal_vector(a, b, c, d)
135     a=a-2
136     b=b-2
137     c=c-2
138     d=d-2
139     p1 = [X_coordinate[a, b][1], Y_coordinate[a, b][1]]
```



```
140 p2 = [X_coordinate[c, d][1], Y_coordinate[c, d][1]]
141 p = p1 - p2
142 magnitude = norm(p)
143 unit_vector = p / magnitude
144 return unit_vector
145 end
146
147 function main(dt, t_end)
148     # initialize
149     rho, u, v, p = initial_conditions ()
150     U_bar = cat(rho, rho .* u, rho .* v, 0.5 .* rho .* (u.^2 .+ v.^2) .+ p ./ (
        gamma - 1); dims=3)
151     for t in 0:dt:t_end
152         println ("t=$t")
153         for i in 3:IMAX+1
154             @threads for j in 3:JMAX+1
155                 U = U_bar[i, j, :]
156                 omega = area(i, j)
157                 # println (i, j)
158                 U_L = U_bar[i, j+1, :] .+ 0.5 .* (U_bar[i, j+1, :] .- U_bar[i, j+2, :])
159                 U_R = U_bar[i, j, :] .+ 0.5 .* (U_bar[i, j, :] .- U_bar[i, j-1, :])
160                 # println (U_bar[i, j, :])
161                 # println (U_bar[i, j-1, :])
162                 # left face
163                 S_left = length(i, j+1, i+1, j+1)
164                 n_left = normal_vector(i, j+1, i, j)
165                 F_left = HLL(U_R, U_L, n_left)
166                 # if j==JMAX
167                 #     println (F_left)
168                 # end
169
170                 U_L = U_bar[i, j, :] .+ 0.5 .* (U_bar[i, j, :] .- U_bar[i, j+1, :])
171                 U_R = U_bar[i, j-1, :] .+ 0.5 .* (U_bar[i, j-1, :] .- U_bar[i, j-2, :])
172                 # println ("right face")
173                 # right face
174                 S_right = length(i, j, i+1, j)
175                 n_right = normal_vector(i, j, i, j+1)
176                 F_right = HLL(U_L, U_R, n_right)
177                 # if j==JMAX
178                 #     println (F_right)
179                 # end
180
181                 U_L = U_bar[i, j, :] .+ 0.5 .* (U_bar[i, j, :] .- U_bar[i-1, j, :])
182                 U_R = U_bar[i+1, j, :] .+ 0.5 .* (U_bar[i+1, j, :] .- U_bar[i+2, j, :])
183                 # println (U_R)
184                 # upper face
185                 S_upper = length(i+1, j, i+1, j+1)
```



```
186     n_upper = normal_vector(i+1, j, i, j)
187     F_upper = HLL(U_L, U_R, n_upper)
188
189     U_L = U_bar[i-1, j, :] .+ 0.5 .* (U_bar[i-1, j, :] .- U_bar[i-2, j, :])
190     U_R = U_bar[i, j, :] .+ 0.5 .* (U_bar[i, j, :] .- U_bar[i+1, j, :])
191     # println (U_R)
192     # down face
193     S_down = length(i, j, i, j+1)
194     n_down = normal_vector(i, j, i+1, j)
195     F_down = HLL(U_R, U_L, n_down)
196
197     U_temp = U .- dt / omega .* (S_left .* F_left .+ S_right .* F_right
198     .+ S_upper .* F_upper .+ S_down .* F_down)
199     U_bar[i, j, :] = U_temp
200     # if j==JMAX
201     #     println (F_left)
202     #     println (dt / omega .* (S_left .* F_left .+ S_right .* F_right
203     .+ S_upper .* F_upper .+ S_down .* F_down))
204     # end
205 end
206
207 #define some monitor
208 println ("rho=", U_bar[80,40,1], "u=", U_bar[80,40,2]/U_bar[80,40,1], "p=", (
209     U_bar[80,40,4] - 0.5 * U_bar[80,40,1] * ((U_bar[80,40,2]/U_bar[80,40,1])
210     ^2 + (U_bar[80,40,3]/U_bar[80,40,1])^2)) * (gamma - 1))
211 println ("rho=", U_bar[80,end,1], "u=", U_bar[80,end,2]/U_bar[80,end,1], "p=", (
212     U_bar[80,end,4] - 0.5 * U_bar[80,end,1] * ((U_bar[80,end,2]/U_bar[80,end,
213     1])^2 + (U_bar[80,end,3]/U_bar[80,end,1])^2)) * (gamma - 1))
214 println ("rho=", U_bar[80,end-3,1], "u=", U_bar[80,end-3,2]/U_bar[80,end-3,1], "
215     p=", (U_bar[80,end-3,4] - 0.5 * U_bar[80,end-3,1] * ((U_bar[80,end-3,2]/
216     U_bar[80,end-3,1])^2 + (U_bar[80,end-3,3]/U_bar[80,end-3,1])^2)) * (
217     gamma - 1))
218
219 # boundary conditions
220 for i in 3:IMAX+1
221     # U_bar[i, 2, :] = U_bar[i, 3, :]
222     # Inviscid wall
223     n_right = normal_vector(i, 3, i, 4)
224     n_upper = normal_vector(i+1, 3, i, 3)
225     A = [1 0 0 0; 0 n_right[1] n_right[2] 0; 0 n_upper[1] n_upper[2] 0; 0 0
226         0 1]
227     B1 = [1 0 0 0; 0 0 0 0; 0 0 1 0; 0 0 0 1]
228     B2 = [1 0 0 0; 0 -1 0 0; 0 0 1 0; 0 0 0 1]
229     # println (A)
230     C = [1 0 0 0; 0 n_right[1] n_upper[1] 0; 0 n_right[2] n_upper[2] 0; 0 0
231         0 1]
```



```
222     # coe_matrix = C * B1 * A
223     # println (U_bar[i, 3,:])
224     # println (B2 * A * U_bar[i, 3,:])
225     U_bar[i, 2,:]= C * B2 * A * U_bar[i, 3,:]
226     U_bar[i, 1,:]= C * B2 * A * U_bar[i, 4,:]
227     # P_U_bar = coe_matrix * U_bar[i, 3,:]
228     # U_bar[i, 1,:]= 3 .* U_bar[i, 2,:] - 2 .* P_U_bar
229 end
230 # Supersonic inflow
231 for i in 3:IMAX+1
232     rho_in = p_inf / (T_inf * R_G)
233     u_in = M_inf * sqrt(gamma * R_G * T_inf)
234     v_in = 0
235     p_in = p_inf
236     U_bar[i, end,1]=U_bar[i, end-1,1]=rho_in
237     U_bar[i, end,2]=U_bar[i, end-1,2]=rho_in*u_in
238     U_bar[i, end,3]=U_bar[i, end-1,3]=rho_in*v_in
239     U_bar[i, end,4]=U_bar[i, end-1,4]=0.5 * rho_in * (u_in^2 + v_in^2) +
        p_in / (gamma - 1)
240     # println (U_bar[i, end,:])
241 end
242 # trans-sonic outlet boundary condition
243 for j in 3:JMAX+1
244     U_bar[1, j, :] = U_bar[3, j, :]
245     U_bar[2, j, :] = U_bar[3, j, :]
246     U_bar[end, j, :] = U_bar[IMAX+1, j, :]
247     U_bar[end-1, j, :] = U_bar[IMAX+1, j, :]
248     #=
249     rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[IMAX+1, j,:])
250     a_out=sqrt(gamma * p_out/rho_out)
251     M_out=sqrt(u_out^2+v_out^2)/a_out
252     n=normal_vector(IMAX+2,j,IMAX+1,j)
253     if M_out >= 1
254         U_bar[end, j, :] = U_bar[IMAX+1, j, :]
255         U_bar[end-1, j, :] = U_bar[IMAX+1, j, :]
256     else
257         p=p_b
258         rho=rho_out+(p_b-p_out)/a_out^2
259         u=u_out+(p_b-p_out)*n[1]/(rho_out*a_out)
260         v=v_out+(p_b-p_out)*n[2]/(rho_out*a_out)
261         P_U_bar=[rho,rho*u,rho*v, 0.5* rho * (u^2 + v^2) + p / (gamma - 1)]
262         U_bar[end-1, j, :]=2 .*P_U_bar.- U_bar[IMAX+1, j, :]
263         U_bar[end-1, j, :]=4 .*P_U_bar.- 3 .*U_bar[IMAX+1, j, :]
264     end
265     rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[3, j, :])
266     a_out=sqrt(gamma * p_out/rho_out)
267     M_out=sqrt(u_out^2+v_out^2)/a_out
```



```
268     n=normal_vector(3,j,4,j)
269     if M_out >= 1
270         U_bar[1, j, :] = U_bar[3, j, :]
271         U_bar[2, j, :] = U_bar[3, j, :]
272     else
273         p=p_b
274         rho=rho_out+(p_b-p_out)/a_out^2
275         u=u_out+(p_b-p_out)*n[1]/(rho_out*a_out)
276         v=v_out+(p_b-p_out)*n[2]/(rho_out*a_out)
277         P_U_bar=[rho,rho*u,rho*v, 0.5* rho * (u^2 + v^2) + p / (gamma - 1)]
278         U_bar[2, j, :]=2 .* P_U_bar ./ U_bar[3, j, :]
279         U_bar[1, j, :]=4 .* P_U_bar ./ 3 .* U_bar[3, j, :]
280     end
281     =#
282 end
283
284 end
285
286 return U_bar
287 end
288
289
290 U_bar = main(1e-6, 5e-3)
291
292 rho_final = U_bar[:, :,1]
293 u_final = U_bar[:, :,2] ./ rho_final
294 v_final = U_bar[:, :,3] ./ rho_final
295 p_final = (U_bar[:, :,4] ./ 0.5 .* rho_final .* (u_final.^2 .+ v_final.^2)) .* (
    gamma - 1)
296 # println (rho_final)
297
298
299 function plot_colored_square(x_coords, y_coords, color)
300     plot!(x_coords, y_coords, seriestype = :shape, fillcolor = color, linecolor = :
        transparent)
301 end
302
303 function get_color(value, min_value, delta)
304     return cgrad(:rainbow)[(value - min_value) / delta]
305 end
306
307 # normalized_rho = (rho_final ./ minimum(rho_final)) ./ (maximum(rho_final) -
    minimum(rho_final))
308 p_rho = plot(figsize = (800, 800), legend=false)
309 count_greater_than_10 = 0
310 for i in 3:IMAX+1
311     for j in 3:JMAX+1
```



```
312     X = X_coordinate
313     Y = Y_coordinate
314
315     I_coordinate=i-2
316     J_coordinate=j-2
317     cell_x = [X[I_coordinate, J_coordinate ], X[I_coordinate+1, J_coordinate ], X
               [I_coordinate+1, J_coordinate+1], X[I_coordinate, J_coordinate+1], X[
               I_coordinate, J_coordinate ]]
318     cell_y = [Y[I_coordinate, J_coordinate ], Y[I_coordinate+1, J_coordinate ], Y
               [I_coordinate+1, J_coordinate+1], Y[I_coordinate, J_coordinate+1], Y[
               I_coordinate, J_coordinate ]]
319
320     Delta = maximum(rho_final[3:IMAX+1,3:JMAX+1]) - minimum(rho_final[3:
               IMAX+1,3:JMAX+1]) == 0 ? 1 : maximum(rho_final[3:IMAX+1,3:JMAX+1])
               - minimum(rho_final[3:IMAX+1,3:JMAX+1])
321     color = get_color( rho_final [i, j ], minimum(rho_final[3:IMAX+1,3:JMAX+1]),
               Delta)
322
323
324     plot_colored_square ( cell_x , cell_y , color)
325     global count_greater_than_10
326
327     if rho_final [i, j] > 1
328         count_greater_than_10 += 1
329     end
330 end
331 end
332
333
334 println ( "Number of rho values greater than 10: ", count_greater_than_10)
335
336 colorbar_ticks = range(minimum(rho_final[3:IMAX+1,3:JMAX+1]), stop=maximum(
               rho_final[3:IMAX+1,3:JMAX+1]), length=11)
337 plot!(p_rho, color=:rainbow, colorbar=:right , colorbar_ticks = colorbar_ticks ,
               colorbar_tick_labels =[@sprintf( "%.2f", x) for x in colorbar_ticks ])
338 # Colorbar(p_rho, pltobj)
339 title !(p_rho, "rho, rhomin=$( @sprintf( \"%.3f\", minimum(rho_final[3:IMAX+1,3:JMAX
               +1])) ), rhomax=$( @sprintf( \"%.3f\", maximum(rho_final[3:IMAX+1,3:JMAX+1])) )")
340 savefig(p_rho, "rho.png")
341
342
343
344 p_p=plot( figsize = (800, 800), legend=false)
345 for i in 3:IMAX+1
346     for j in 3:JMAX+1
347         X = X_coordinate
348         Y = Y_coordinate
```



```
349     I_coordinate=i-2
350     J_coordinate=j-2
351     cell_x = [X[I_coordinate, J_coordinate ], X[I_coordinate+1, J_coordinate ], X
               [I_coordinate+1, J_coordinate+1], X[I_coordinate, J_coordinate+1], X[
               I_coordinate, J_coordinate ]]
352     cell_y = [Y[I_coordinate, J_coordinate ], Y[I_coordinate+1, J_coordinate ], Y
               [I_coordinate+1, J_coordinate+1], Y[I_coordinate, J_coordinate+1], Y[
               I_coordinate, J_coordinate ]]
353
354     Delta = maximum(p_final[3:IMAX+1,3:JMAX+1]) - minimum(p_final[3:IMAX
               +1,3:JMAX+1]) == 0 ? 1 : maximum(p_final[3:IMAX+1,3:JMAX+1]) -
               minimum(p_final[3:IMAX+1,3:JMAX+1])
355     color = get_color ( p_final [ i , j ], minimum(p_final[3:IMAX+1,3:JMAX+1]),
               Delta)
356
357
358     plot_colored_square ( cell_x , cell_y , color )
359 end
360 end
361 title !(p_p,"p, pmin=$( @sprintf("%.3f", minimum(p_final[3:IMAX+1,3:JMAX+1])) ),
               pmax=$( @sprintf("%.3f", maximum(p_final[3:IMAX+1,3:JMAX+1])) )" )
362 savefig (p_p,"p.png")
363
364
365
366 p_u=plot( figsize = (800, 800),legend=false )
367 for i in 3:IMAX+1
368     for j in 3:JMAX+1
369         X = X_coordinate
370         Y = Y_coordinate
371         I_coordinate=i-2
372         J_coordinate=j-2
373         cell_x = [X[I_coordinate, J_coordinate ], X[I_coordinate+1, J_coordinate ], X
               [I_coordinate+1, J_coordinate+1], X[I_coordinate, J_coordinate+1], X[
               I_coordinate, J_coordinate ]]
374         cell_y = [Y[I_coordinate, J_coordinate ], Y[I_coordinate+1, J_coordinate ], Y
               [I_coordinate+1, J_coordinate+1], Y[I_coordinate, J_coordinate+1], Y[
               I_coordinate, J_coordinate ]]
375
376         Delta = maximum(u_final[3:IMAX+1,3:JMAX+1]) - minimum(u_final[3:IMAX
               +1,3:JMAX+1]) == 0 ? 1 : maximum(u_final[3:IMAX+1,3:JMAX+1]) -
               minimum(u_final[3:IMAX+1,3:JMAX+1])
377         color = get_color ( u_final [ i , j ], minimum(u_final[3:IMAX+1,3:JMAX+1]),
               Delta)
378
379
380         plot_colored_square ( cell_x , cell_y , color )
```



```
381     end
382 end
383 title !(p_u,"u, umin=$(@sprintf("%.3f", abs(minimum(u_final[3:IMAX+1,3:JMAX+1])))),
      umax=$(@sprintf("%.3f", abs(maximum(u_final[3:IMAX+1,3:JMAX+1])))) ")
384 savefig(p_u,"u.png")
```