THE HONG KONG POLYTECHNIC UNIVERSITY
香港理工大學

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:

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

2. Conservation of momentum:

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

3. Conservation of energy:

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

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:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} = 0 \tag{4}$$

in which

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

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:

$$\frac{\partial}{\partial t}\int_{\Omega} U \mathrm{d}\Omega + \oint_{S} \vec{F} \cdot \vec{n} \mathrm{d}S = 0 \tag{6}$$

define

$$\overline{U} = \frac{1}{\Omega}\int_{\Omega} U \mathrm{d}\Omega \tag{7}$$

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:

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

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:

$$U^L_{I+1/2} = g^L(U_{I-1}, U_I, U_{I+1}), \;\; U^R_{I+1/2} = g^R(U_I, U_{I+1}, U_{I+2}) \tag{9}$$

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

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

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

$$U^L_{I+1/2} = U_I + s_1/4\Big[(1 - s_1/3)(U_I - U_{I-1}) + (1 + s_1/3)(U_{I+1} - U_I)\Big] \tag{11}$$

$$U^R_{I+1/2} = U_{I+1} - s_2/4\Big[(1 - s_2/3)(U_{I+2} - U_{I+1}) + (1 + s_2/3)(U_{I+1} - U_I)\Big] \tag{12}$$

in which

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

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

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

4 Interface flux

Consider a 2-D mesh,

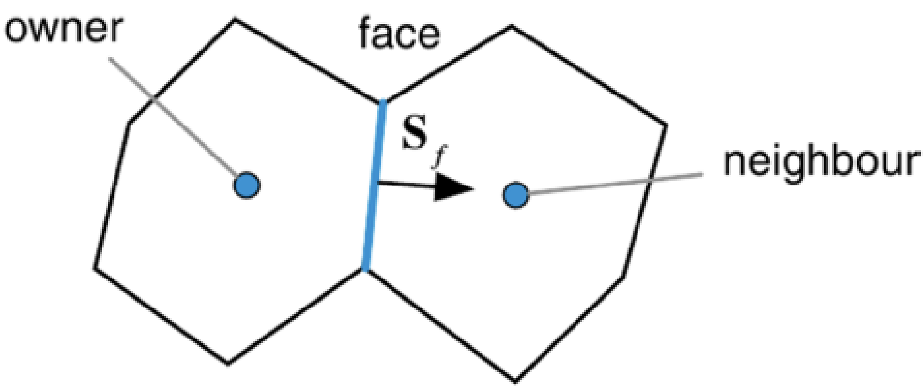


Figure 1: 2D mesh and their relationship

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.

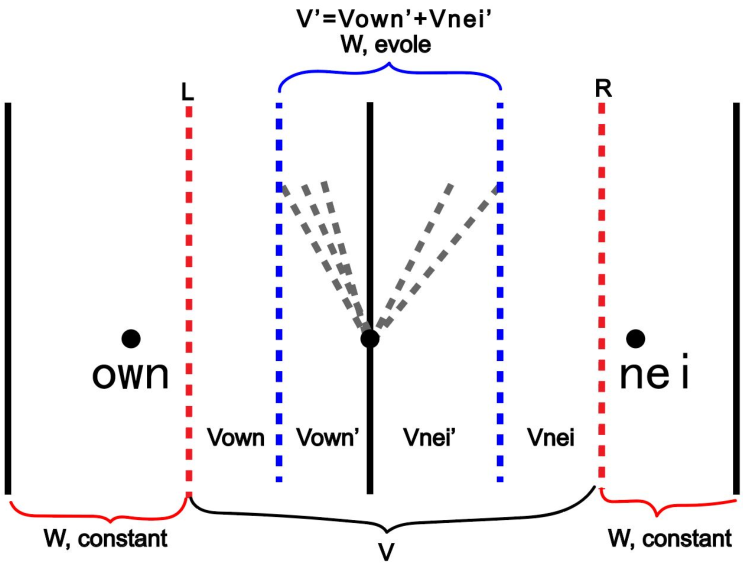


Figure 2: HLL Riemann solver

Consider an original equation:

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} = 0 \tag{15}$$

Integrated along x:

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

Integrated along t

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

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

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

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

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

The left-hand side becomes

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

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

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

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

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

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.

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

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:

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

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.

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

Similarly, we have

$$p = \frac{c^2}{\gamma_f}\rho_f = p^+ + p^- \tag{26}$$

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

5 Instancing in 2D problem

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

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

in which

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

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

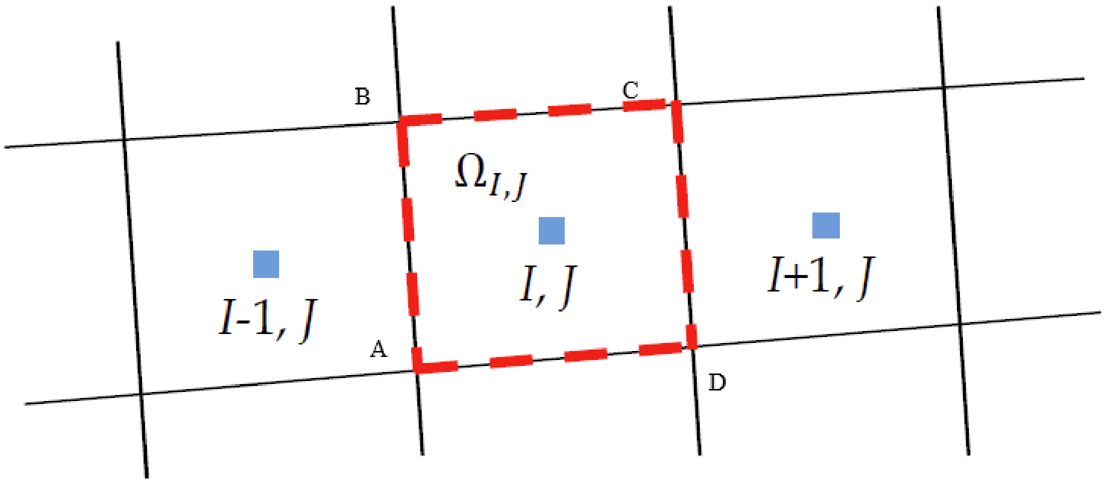


Figure 3: Regions in sod shock tube with different space discretization schemes

$$\begin{aligned} \Omega_{\text{IJ}} &= \frac{1}{2}|\vec{r}_{\text{AC}} \times \vec{r}_{\text{BD}}| \\ &= \frac{1}{2}[(x_{\text{C}} - x_{\text{A}})(y_{\text{D}} - y_{\text{B}}) - (x_{\text{D}} - x_{\text{B}})(y_{\text{C}} - y_{\text{A}})] \\ &= \frac{1}{2}(\Delta x_{\text{AC}}\Delta y_{\text{BD}} - \Delta x_{\text{BD}}\Delta y_{\text{AC}}) \end{aligned} \tag{30}$$

and length of edge

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

Consider the flux vector **F** on the face CD, separate it as the convective and pressure terms.

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

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

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

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

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

Further,

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

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

where

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

Similarly, we have

$$p_{CD} == p^+ p_L + p^- p_R \tag{38}$$

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

Similarly, we have

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

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

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

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

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

substitute all equations back to general equation we have:

$$U_{IJ}^{n+1} = U_{IJ}^{n} + \frac{\Delta t}{\Omega_{\mathrm{IJ}}} \sum F_n \Delta S \tag{43}$$

where

$$F_n = F \times n_x + G \times n_y \tag{44}$$

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.

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

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 te be specified.The primitive variables at the farfield boundary are obtained from

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

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) × 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.

```
@pyimport plot3d as p3d

# read mesh
block = p3d.read_plot3D("Cylinder.dat", binary = false )

IMAX, JMAX, KMAX = size(block[1].X)
X_coordinate = block [1]. X
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].

```
    for j in 3:JMAX+1
    rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[IMAX+1, j,:])
    a_out=sqrt (gamma * p_out/rho_out)
    M_out=sqrt(u_out^2+v_out^2)/a_out
    n=normal_vector(IMAX+2,j,IMAX+1,j)
    if M_out >= 1
        U_bar[end, j ,:]  = U_bar[IMAX+1, j,:]
        U_bar[end-1, j ,:]  = U_bar[IMAX+1, j,:]
    else
        p=p_b
        rho=rho_out+(p_b-p_out)/a_out^2
        u=u_out+(p_b-p_out)*n[1]/(rho_out*a_out)
        v=v_out+(p_b-p_out)*n[2]/(rho_out*a_out)
        P_U_bar=[rho,rho*u,rho*v, 0.5* rho * (u^2 + v^2) + p / (gamma - 1)]
        U_bar[end-1, j ,:]=2 .*P_U_bar .- U_bar[IMAX+1, j,:]
        U_bar[end-1, j ,:]=4 .*P_U_bar .- 3 .*U_bar[IMAX+1, j,:]
    end
    rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[3, j ,:])
    a_out=sqrt (gamma * p_out/rho_out)
    M_out=sqrt(u_out^2+v_out^2)/a_out
    n=normal_vector(3,j ,4, j )
    if M_out >= 1
        U_bar[1, j ,:]  = U_bar[3, j ,:]
        U_bar[2, j ,:]  = U_bar[3, j ,:]
    else
        p=p_b
        rho=rho_out+(p_b-p_out)/a_out^2
        u=u_out+(p_b-p_out)*n[1]/(rho_out*a_out)
```

```
        v=v_out+(p_b−p_out)∗n[2]/(rho_out∗a_out)
        P_U_bar=[rho,rho∗u,rho∗v, 0.5∗ rho ∗ (u^2 + v^2) + p / (gamma − 1)]
        U_bar[2, j ,:]=2 .∗ P_U_bar .− U_bar[3, j ,:]
        U_bar[1, j ,:]=4 .∗ P_U_bar .− 3 .∗U_bar[3, j ,:]
    end
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.

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

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

```
$ 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%.

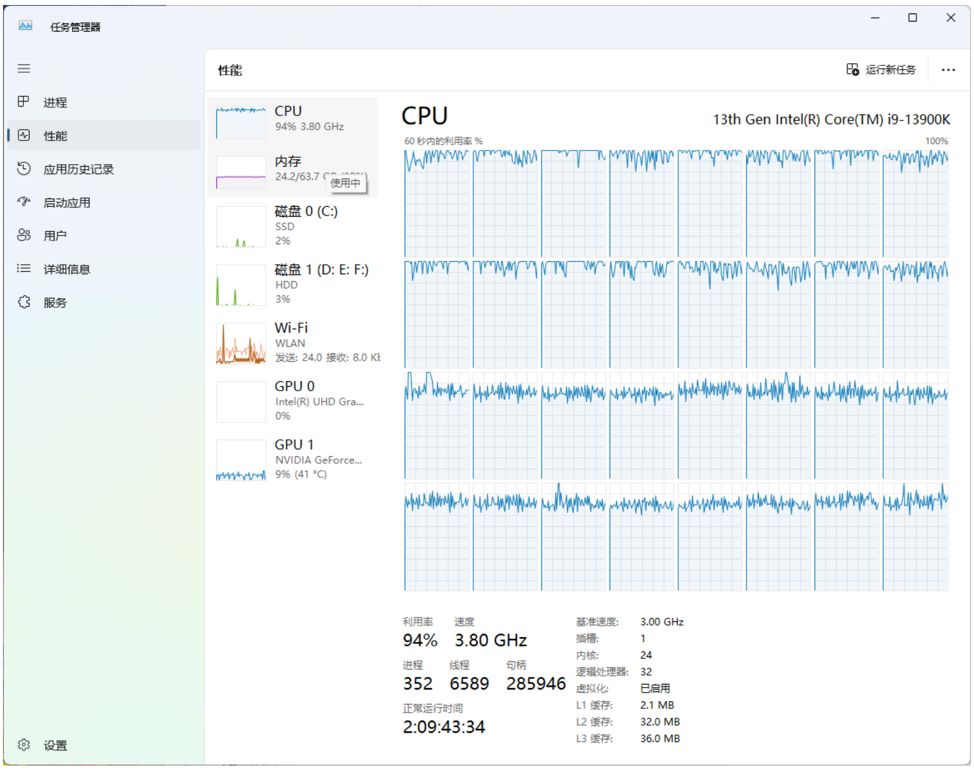


Figure 4: CPU usage

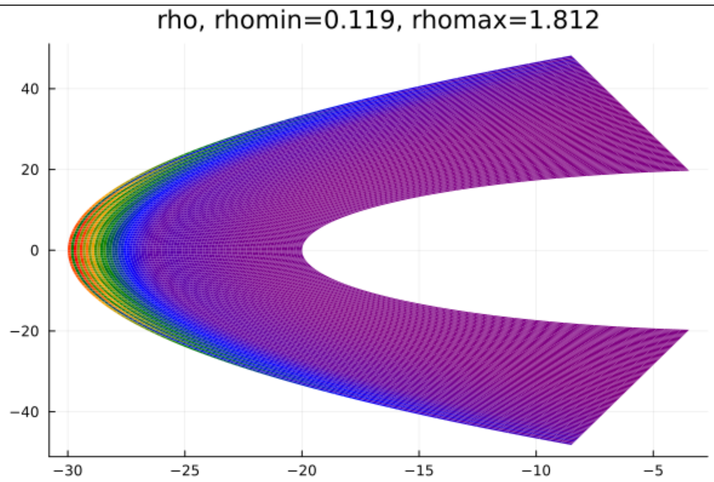
7 Simulation details

7.1 Initialization

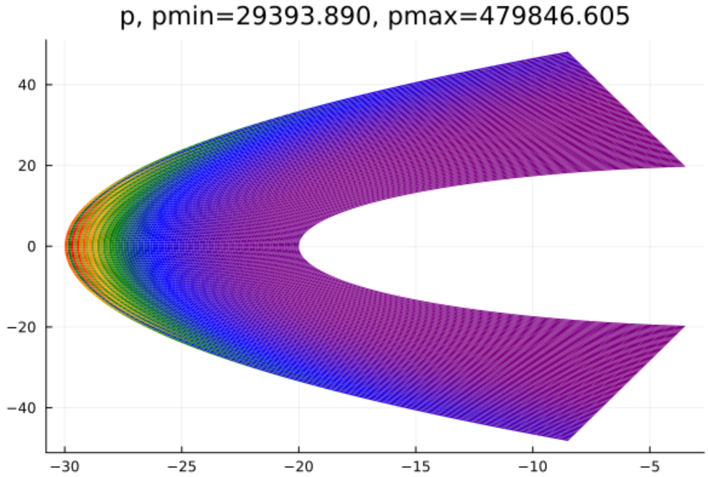
In order to accelerate the simulation process, the density field was initialized using precalculated density after shock wave of around $0.11\ \mathrm{kg/m^3}$. So as to pressure filed, but velocity field was initialized as zero. With serval attempts, overall time step was set to 1×10^{-6}, simulation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

7.2 Simulation results

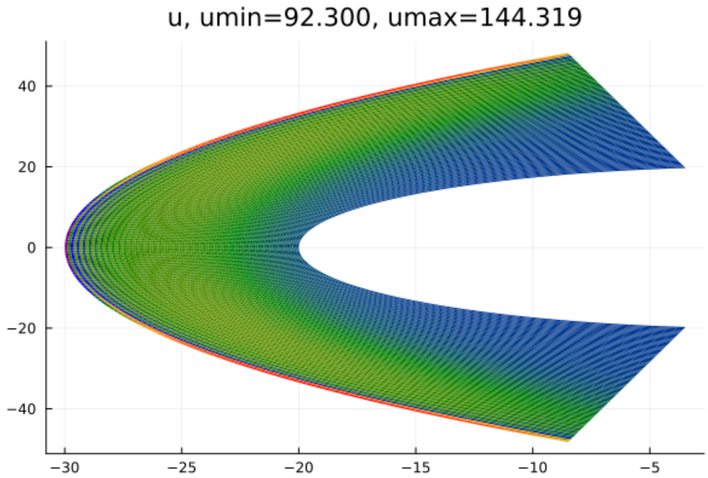
Figure 5, shows the simulation results of density, pressure and velocity field 0.1 second initial condition. For convenance, the two outflow are was set to supersonic outflow. The shape of shock wave is very clear, especially in density field, the density of free stream is really small, in shock wave, 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.



(a) Density



(b) Pressure



(c) Horizontal velocity

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. Sense the empirical shock wave is stead-state, but our simulation is not long enough to reach stead-state condition, the error is reasonable.

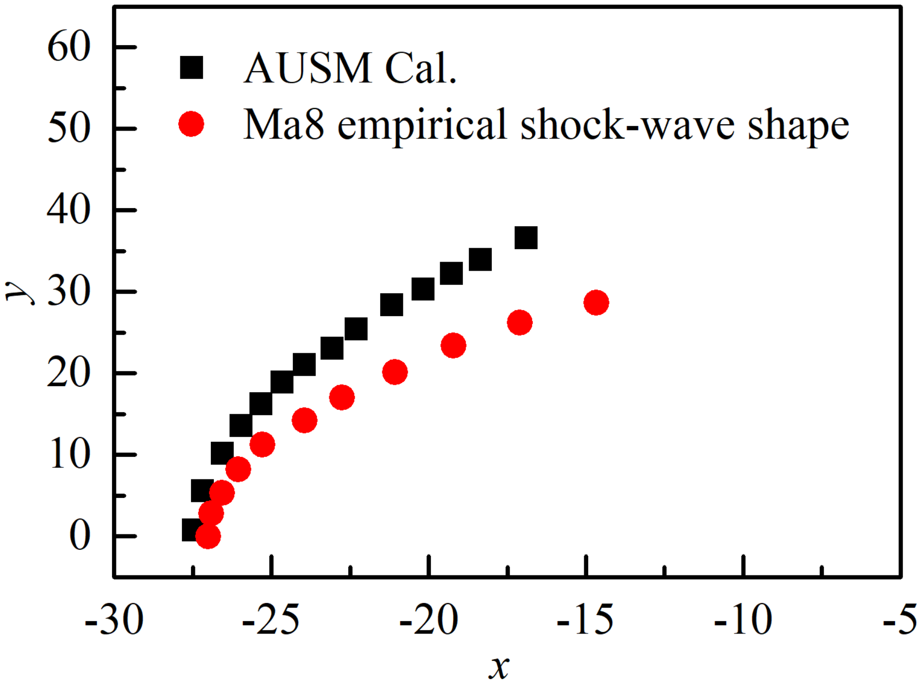
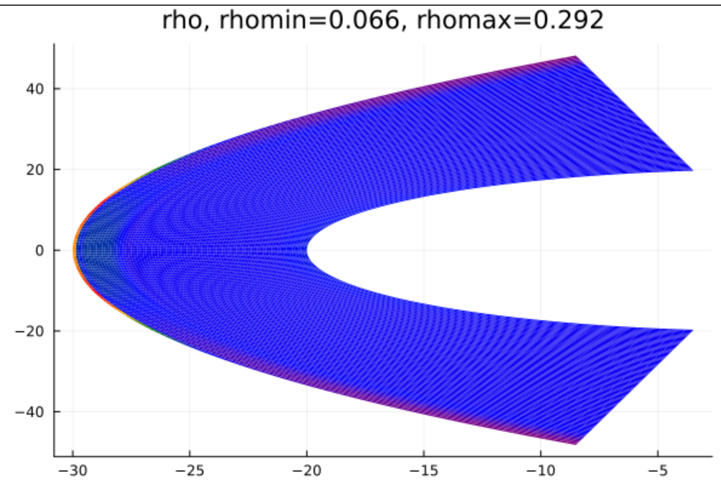
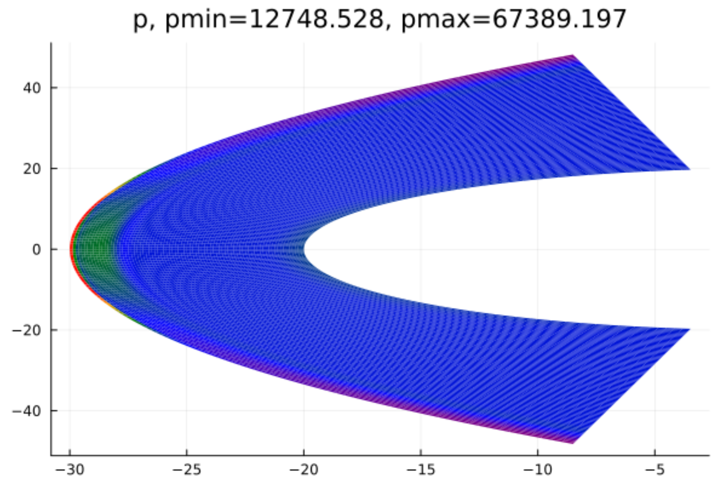


Figure 6: Regions in sod shock tube with different space discretization schemes

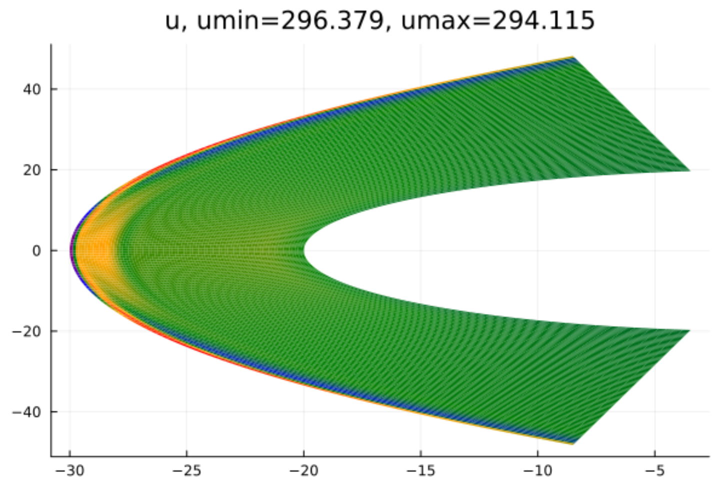
Further reach shows that, the stability and robustness of HLL scheme, is not as good as that of AUSM schemes. During the simulation, negative pressure always occurs and its frequency depend on initial condition. Finally, a shock wave shape at 0.005s was obtained, as can be seen in Figure 7. However,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



(a) Density



(b) Pressure



(c) Horizontal velocity

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

```
#/usr/local/bin/julia
#coded based on julia v1.9
#copyright © ZHANG Ting, polyU, Hong Kong
#e-mail ting123.zhang@connect.polyu.hk
#updated at 05/Dec/2024
using CSV
using DataFrames
using Plots
using PyCall
using LinearAlgebra
using Colors
using Printf
@pyimport plot3d as p3d
using Base.Threads: @threads, nthreads

# confirm the number of threads
 println("Number of threads: ", nthreads())
# read mesh
block = p3d.read_plot3D("Cylinder.dat", binary = false)

IMAX, JMAX, KMAX = size(block[1].X)
X_coordinate = block[1].X
Y_coordinate = block[1].Y
U_bar = zeros(Float64, IMAX+3, JMAX+3, 4)


gamma = 1.4
R = 8.314
M_IG = 0.029
c_v = 717
```

```
R_G=R/M_IG
# inflow  outflow
M_inf= 8.1
T_inf = 63.73
p_inf = 370.6
rho_inf=p_inf/(R_G*T_inf)
a_inf=sqrt(gamma*R_G*T_inf)
u_inf=M_inf*a_inf

rho_after_shock=0.116
p_after_shock=28465
M_after_shock=0.392
T_after_shock=849.75
a_after_shock=sqrt(gamma*R_G*T_after_shock)
u_after_shock=a_after_shock*M_after_shock
# p_b=101325
p_b=p_inf

T_total=T_inf*(1+((gamma-1)/2)*M_inf^2)
p_total=p_inf*(1+((gamma-1)/2)*M_inf^2)^(gamma/(gamma-1))

#  initial_conditions
function  initial_conditions ()
    rho = fill (rho_after_shock, IMAX+3, JMAX+3)
    u = fill (0,IMAX+3, JMAX+3) #M_inf * sqrt(gamma * R_G * T_inf)
    v = zeros(IMAX+3, JMAX+3)
    p = fill (p_after_shock, IMAX+3, JMAX+3)
    return rho, u, v, p
end


function U_flux_decompose(U)
    rho = U[1]
    u = U[2] / U[1]
    v = U[3] / U[1]
    p = (U[4] - 0.5 * rho * (u^2 + v^2)) * (gamma - 1)
    return rho, u, v, p
end

function area(I, J)
    I=I-2
    J=J-2
    p1 = [X_coordinate[I, J][1], Y_coordinate[I, J][1],0]
    p2 = [X_coordinate[I, J+1][1], Y_coordinate[I, J+1][1],0]
    p3 = [X_coordinate[I+1, J+1][1], Y_coordinate[I+1, J+1][1],0]
    p4 = [X_coordinate[I+1, J][1], Y_coordinate[I+1, J][1],0]
    r_1 = p1 - p3
```

```
    r_2 = p2 - p4
    cross_product = cross(r_1,r_2)
    magnitude = norm(cross_product)
    return magnitude
end

function length(a, b, c, d)
    a=a-2
    b=b-2
    c=c-2
    d=d-2
    p1 = [X_coordinate[a, b][1], Y_coordinate[a, b][1]]
    p2 = [X_coordinate[c, d][1], Y_coordinate[c, d][1]]
    p = p1 - p2
    mag = norm(p)
    return mag
end

function AUSM(U_L, U_R, n)
    rho_L, u_L, v_L, p_L = U_flux_decompose(U_L)
    rho_R, u_R, v_R, p_R = U_flux_decompose(U_R)
    if p_L/rho_L*p_R/rho_R < 0
        println ("rho=",rho_L, "u=",u_L, "v=",v_L, "p=",p_L)
        println ("rho=",rho_R, "u=",u_R, "v=",v_R, "p=",p_R)
        println (U_L)
        println (U_R)
    end
    a_L = sqrt(gamma * p_L/rho_L)
    a_R = sqrt(gamma * p_R/rho_R)
    a = (a_L + a_R) / 2
    M_L = (u_L*n[1]+v_L*n[2]) / a_L
    M_R = (u_R*n[1]+v_R*n[2]) / a_R

    M_plus = M_L > 1 ? M_L : M_L < -1 ? 0 : (M_L + 1)^2 / 4
    M_minus = M_R > 1 ? 0 : M_R < -1 ? M_R : -(M_R - 1)^2 / 4

    u = (M_plus + M_minus) * a
    if u >= 0
        PHI =U_L
        PHI[end]=p_L * (gamma / (gamma - 1)) + (rho_L * 0.5 * (u_L^2 + v_L^2))
    else
        PHI =U_R
        PHI[end]=p_R* (gamma / (gamma - 1)) + (rho_R * 0.5 * (u_R^2 + v_R^2))
    end

    F_c = u * PHI

```

```
    p_plus = M_L > 1 ? 1 : M_L < −1 ? 0 : (M_L + 1)^2 ∗ (2 − M_L) / 4
    p_minus = M_R > 1 ? 0 : M_R < −1 ? (M_R − 1)^2 ∗ (2 + M_R) / 4 : 1

    p_ASUM = p_plus ∗ p_L + p_minus ∗ p_R

    F_p = [0, p_ASUM∗n[1], p_ASUM∗n[2], 0]

    # M_L = v_L / a_L
    # M_R = v_R / a_R

    # M_plus = M_L > 1 ? M_L : M_L < −1 ? 0 : (M_L + 1)^2 / 4
    # M_minus = M_R > 1 ? 0 : M_R < −1 ? M_R : −(M_R − 1)^2 / 4

    # v = (M_plus + M_minus) ∗ a
    # if v >= 0
    #     PHI =U_L
    #     PHI[end]=p_L ∗ (gamma / (gamma − 1)) ∗ (rho_L ∗ 0.5 ∗ (u_L^2 + v_L^2))
    # else
    #     PHI =U_R
    #     PHI[end]=p_R ∗ (gamma / (gamma − 1)) ∗ (rho_R ∗ 0.5 ∗ (u_R^2 + v_R^2))
    # end

    # G_c = v ∗ PHI

    # p_plus = M_L > 1 ? 1 : M_L < −1 ? 0 : (M_L + 1)^2 ∗ (2 − M_L) / 4
    # p_minus = M_R > 1 ? 0 : M_R < −1 ? (M_R − 1)^2 ∗ (2 + M_R) / 4 : 1

    # p_ASUM = p_plus ∗ p_L + p_minus ∗ p_R

    # G_p = [0, 0, p_ASUM, 0]

    return F_c , F_p #∗n[1] +G_p∗n[2]
end

function normal_vector(a, b, c, d)
    a=a−2
    b=b−2
    c=c−2
    d=d−2
    p1 = [X_coordinate[a, b ][1], Y_coordinate[a, b ][1]]
    p2 = [X_coordinate[c, d ][1], Y_coordinate[c, d ][1]]
    p = p1 − p2
    magnitude = norm(p)
    unit_vector = p / magnitude
    return unit_vector
end
```

```
function main(dt, t_end)
    # initialize
    rho, u, v, p = initial_conditions()
    U_bar = cat(rho, rho .* u, rho .* v, 0.5 .* rho .* (u.^2 .+ v.^2) .+ p ./ (
        gamma - 1); dims=3)
    for t in 0:dt:t_end
        println("t=$t")
        for i in 3:IMAX+1
            @threads for j in 3:JMAX+1
                U = U_bar[i, j, :]
                omega = area(i, j)
                # println(i,j)
                U_L = U_bar[i, j+1,:] .+ 0.5 .* (U_bar[i, j+1,:] .- U_bar[i, j+2,:])
                U_R = U_bar[i, j,:] .+ 0.5 .* (U_bar[i, j,:] .- U_bar[i, j-1,:])
                # println(U_bar[i, j,:])
                # println(U_bar[i, j-1,:])
                # left face
                S_left = length(i, j+1, i+1, j+1)
                n_left = normal_vector(i, j+1, i, j)
                F_c, F_p = AUSM(U_R, U_L, n_left)
                F_left = F_c .+ F_p
                # if j==JMAX
                #     println(F_left)
                # end

                U_L = U_bar[i, j,:] .+ 0.5 .* (U_bar[i, j,:] .- U_bar[i, j+1,:])
                U_R = U_bar[i, j-1,:] .+ 0.5 .* (U_bar[i, j-1,:] .- U_bar[i, j-2,:])
                # println("right face")
                # right face
                S_right = length(i, j, i+1, j)
                n_right = normal_vector(i, j, i, j+1)
                F_c, F_p = AUSM(U_L, U_R, n_right)
                F_right = F_c .+ F_p
                # if j==JMAX
                #     println(F_right)
                # end

                U_L = U_bar[i, j,:] .+ 0.5 .* (U_bar[i, j,:] .- U_bar[i-1, j,:])
                U_R = U_bar[i+1, j,:] .+ 0.5 .* (U_bar[i+1, j,:] .- U_bar[i+2, j,:])
                # println(U_R)
                # upper face
                S_upper = length(i+1, j, i+1, j+1)
                n_upper = normal_vector(i+1, j, i, j)
                F_c, F_p = AUSM(U_L, U_R, n_upper)
                F_upper = F_c .+ F_p

                U_L = U_bar[i-1, j,:] .+ 0.5 .* (U_bar[i-1, j,:] .- U_bar[i-2, j,:])
```

```
            U_R = U_bar[i, j ,:] .+ 0.5 .* (U_bar[i, j ,:] .- U_bar[i+1, j ,:])
            # println (U_R)
            # down face
            S_down = length(i, j, i, j+1)
            n_down = normal_vector(i, j, i+1, j)
            F_c, F_p = AUSM(U_R, U_L, n_down)
            F_down = F_c .+ F_p

            U_temp = U .- dt / omega .* ( S_left .* F_left .+ S_right .* F_right .+ S_upper .* F_upper .+ S_down .* F_down)
            U_bar[i, j ,:] = U_temp
            # if j==JMAX
            #     println (F_left)
            #     println (dt / omega .*(S_left .* F_left .+ S_right .* F_right .+ S_upper .* F_upper .+ S_down .* F_down))
            # end
        end
    end

    #define some monitor
    println ("rho=",U_bar[80,40,1],"u=",U_bar[80,40,2]/U_bar [80,40,1], "p=",(U_bar[80,40,4] - 0.5 * U_bar[80,40,1] * ((U_bar [80,40,2]/ U_bar [80,40,1])^2 + (U_bar [80,40,3]/ U_bar [80,40,1]) ^2)) * (gamma - 1))
    println ("rho=",U_bar[80,end,1],"u=",U_bar[80,end,2]/U_bar[80,end ,1], "p=",(U_bar[80,end,4] - 0.5 * U_bar[80,end,1] * ((U_bar[80,end ,2]/ U_bar[80,end ,1])^2 + (U_bar[80,end ,3]/ U_bar[80,end ,1]) ^2)) * (gamma - 1))
    println ("rho=",U_bar[80,end-3,1],"u=",U_bar[80,end-3,2]/U_bar[80,end-3,1],"p=",(U_bar[80,end-3,4] - 0.5 * U_bar[80,end-3,1] * ((U_bar[80,end-3,2]/U_bar[80,end-3,1])^2 + (U_bar[80,end-3,3]/U_bar[80,end-3,1])^2)) * (gamma - 1))

    # boundary conditions
    for i in 3:IMAX+1
        # U_bar[i, 2,:] = U_bar[i, 3,:]
        # Inviscid wall
        n_right = normal_vector(i, 3, i, 4)
        n_upper = normal_vector(i+1, 3, i, 3)
        A = [1 0 0 0; 0 n_right [1] n_right [2] 0; 0 n_upper[1] n_upper[2] 0; 0 0 0 1]
        B1 = [1 0 0 0; 0 0 0 0; 0 0 1 0; 0 0 0 1]
        B2 = [1 0 0 0; 0 -1 0 0; 0 0 1 0; 0 0 0 1]
        # println (A)
        C = [1 0 0 0; 0 n_right [1] n_upper[1] 0; 0 n_right [2] n_upper[2] 0; 0 0 0 1]
        # coe_matrix = C * B1 * A
        # println (U_bar[i, 3,:])
        # println (B2 *A*U_bar[i, 3,:])
```

```
            U_bar[i, 2,:]= C * B2 * A*U_bar[i, 3,:]
            U_bar[i, 1,:]= C * B2 * A*U_bar[i, 4,:]
            # P_U_bar = coe_matrix * U_bar[i, 3,:]
            # U_bar[i, 1,:] = 3 .* U_bar[i, 2,:] - 2 .* P_U_bar
        end
        # Supersonic inflow
        for i in 3:IMAX+1
            rho_in = p_inf / (T_inf* R_G)
            u_in = M_inf * sqrt(gamma * R_G * T_inf)
            v_in = 0
            p_in = p_inf
            U_bar[i, end,1]=U_bar[i, end-1,1]=rho_in
            U_bar[i, end,2]=U_bar[i, end-1,2]=rho_in*u_in
            U_bar[i, end,3]=U_bar[i, end-1,3]=rho_in*v_in
            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)
            # println(U_bar[i, end,:])
        end
        # trans-sonic outlet boundary condition
        for j in 3:JMAX+1
            U_bar[1, j,:] = U_bar[3, j,:]
            U_bar[2, j,:] = U_bar[3, j,:]
            U_bar[end, j,:] = U_bar[IMAX+1, j,:]
            U_bar[end-1, j,:] = U_bar[IMAX+1, j,:]
        #=
            rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[IMAX+1, j,:])
            a_out=sqrt(gamma * p_out/rho_out)
            M_out=sqrt(u_out^2+v_out^2)/a_out
            n=normal_vector(IMAX+2,j,IMAX+1,j)
            if M_out >= 1
                U_bar[end, j,:] = U_bar[IMAX+1, j,:]
                U_bar[end-1, j,:] = U_bar[IMAX+1, j,:]
            else
                p=p_b
                rho=rho_out+(p_b-p_out)/a_out^2
                u=u_out+(p_b-p_out)*n[1]/(rho_out*a_out)
                v=v_out+(p_b-p_out)*n[2]/(rho_out*a_out)
                P_U_bar=[rho,rho*u,rho*v, 0.5* rho * (u^2 + v^2) + p / (gamma - 1)]
                U_bar[end-1, j,:]=2 .*P_U_bar .- U_bar[IMAX+1, j,:]
                U_bar[end-1, j,:]=4 .*P_U_bar .- 3 .*U_bar[IMAX+1, j,:]
            end
            rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[3, j,:])
            a_out=sqrt(gamma * p_out/rho_out)
            M_out=sqrt(u_out^2+v_out^2)/a_out
            n=normal_vector(3,j,4,j)
            if M_out >= 1
                U_bar[1, j,:] = U_bar[3, j,:]
```

```
                    U_bar[2, j ,:]  = U_bar[3, j ,:]
                else
                    p=p_b
                    rho=rho_out+(p_b−p_out)/a_out^2
                    u=u_out+(p_b−p_out)∗n[1]/(rho_out∗a_out)
                    v=v_out+(p_b−p_out)∗n[2]/(rho_out∗a_out)
                    P_U_bar=[rho,rho∗u,rho∗v, 0.5∗ rho ∗ (u^2 + v^2) + p / (gamma − 1)]
                    U_bar[2, j ,:]=2 .∗ P_U_bar .− U_bar[3, j ,:]
                    U_bar[1, j ,:]=4 .∗ P_U_bar .− 3 .∗U_bar[3, j ,:]
                end
                =#
            end

        end

    return U_bar
end

U_bar = main(1e−6, 1e−1)

rho_final = U_bar[:, :,1]
u_final = U_bar[:, :,2] ./ rho_final
v_final = U_bar[:, :,3] ./ rho_final
p_final = (U_bar[:, :,4] .− 0.5 .∗ rho_final .∗ (u_final .^2 .+ v_final .^2)) .∗ (gamma − 1)
# println(rho_final)

# plot colored cells
function plot_colored_square(x_coords, y_coords, color)
    plot!(x_coords, y_coords, seriestype = :shape, fillcolor = color, linecolor = :transparent)
end
# get color
function get_color(value, min_value, delta)
    return cgrad(:rainbow)[(value − min_value) / delta]
end
# plotrho
# normalized_rho = (rho_final .− minimum(rho_final)) ./ (maximum(rho_final) − minimum(rho_final))
p_rho = plot(figsize = (800, 800),legend=false)
count_greater_than_10 = 0
for i in 3:IMAX+1
    for j in 3:JMAX+1
        X = X_coordinate
        Y = Y_coordinate
        # define cell
        I_coordinate =i−2
```

```
            J_coordinate=j−2
            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]]
            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]]

            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])
            color = get_color(rho_final[i, j], minimum(rho_final[3:IMAX+1,3:JMAX+1]),
                 Delta)

            # plot everycell
            plot_colored_square(cell_x, cell_y, color)
            global count_greater_than_10
            if rho_final[i, j] > 1
                count_greater_than_10 += 1
            end
        end
end

println("Number of rho values greater than 10: ", count_greater_than_10)

colorbar_ticks = range(minimum(rho_final[3:IMAX+1,3:JMAX+1]), stop=maximum(
    rho_final[3:IMAX+1,3:JMAX+1]), length=11)
plot!(p_rho, color=:rainbow, colorbar=:right, colorbar_ticks=colorbar_ticks,
    colorbar_tick_labels=[@sprintf("%.2f", x) for x in colorbar_ticks])
# Colorbar(p_rho, pltobj)
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])))")
savefig(p_rho,"rho.png")


# plotp
p_p=plot(figsize = (800, 800),legend=false)
for i in 3:IMAX+1
    for j in 3:JMAX+1
            X = X_coordinate
            Y = Y_coordinate
            I_coordinate=i−2
            J_coordinate=j−2
            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]]
            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]]

            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])
            color = get_color(p_final[i, j], minimum(p_final[3:IMAX+1,3:JMAX+1]), Delta)

            # plot everycell
            plot_colored_square(cell_x, cell_y, color)
        end
end
 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])) )")
savefig(p_p,"p.png")


# # plot u
p_u=plot( figsize = (800, 800),legend=false)
for i in 3:IMAX+1
    for j in 3:JMAX+1
            X = X_coordinate
            Y = Y_coordinate
            I_coordinate=i-2
            J_coordinate=j-2
            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]]
            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]]

            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])
            color = get_color(u_final[i, j], minimum(u_final[3:IMAX+1,3:JMAX+1]), Delta)

            # plot everycell
            plot_colored_square(cell_x, cell_y, color)
        end
end
 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]))))")
savefig(p_u,"u.png")
```

8.2 HLL based 2D Riemann solver

```
#/usr/local/bin/julia
#coded based on julia v1.9
#copyright © ZHANG Ting, polyU, Hong Kong
#e-mail ting123.zhang@connect.polyu.hk
#updated at 05/Dec/2024
using CSV
using DataFrames
using Plots
using PyCall
using LinearAlgebra
using Colors
using Printf
@pyimport plot3d as p3d
using Base.Threads: @threads, nthreads


println("Number of threads: ", nthreads())

block = p3d.read_plot3D("Cylinder.dat", binary = false)

IMAX, JMAX, KMAX = size(block[1].X)
X_coordinate = block[1].X
Y_coordinate = block[1].Y
U_bar = zeros(Float64, IMAX+3, JMAX+3, 4)


gamma = 1.4
R = 8.314
M_IG = 0.029
c_v = 717
R_G=R/M_IG

M_inf = 8.1
T_inf = 63.73
p_inf = 370.6
rho_inf=p_inf/(R_G*T_inf)
a_inf=sqrt(gamma*R_G*T_inf)
u_inf=M_inf*a_inf

rho_after_shock=0.116
p_after_shock=28465
M_after_shock=0.392
T_after_shock=849.75
a_after_shock=sqrt(gamma*R_G*T_after_shock)
u_after_shock=a_after_shock*M_after_shock
```

```
# p_b=101325
p_b=p_inf

T_total=T_inf*(1+((gamma-1)/2)*M_inf^2)
p_total=p_inf*(1+((gamma-1)/2)*M_inf^2)^(gamma/(gamma-1))


function initial_conditions ()
    rho = fill (rho_after_shock, IMAX+3, JMAX+3)
    u = fill (0, IMAX+3, JMAX+3) #M_inf * sqrt(gamma * R_G * T_inf)
    v = zeros(IMAX+3, JMAX+3)
    p = fill (p_after_shock, IMAX+3, JMAX+3)
    return rho, u, v, p
end


function U_flux_decompose(U)
    rho = U[1]
    u = U[2] / U[1]
    v = U[3] / U[1]
    p = (U[4] - 0.5 * rho * (u^2 + v^2)) * (gamma - 1)
    return rho, u, v, p
end

function area(I, J)
    I=I-2
    J=J-2
    p1 = [X_coordinate[I, J][1], Y_coordinate[I, J][1],0]
    p2 = [X_coordinate[I, J+1][1], Y_coordinate[I, J+1][1],0]
    p3 = [X_coordinate[I+1, J+1][1], Y_coordinate[I+1, J+1][1],0]
    p4 = [X_coordinate[I+1, J][1], Y_coordinate[I+1, J][1],0]
    r_1 = p1 - p3
    r_2 = p2 - p4
    cross_product = cross(r_1,r_2)
    magnitude = norm(cross_product)
    return magnitude
end

function length(a, b, c, d)
    a=a-2
    b=b-2
    c=c-2
    d=d-2
    p1 = [X_coordinate[a, b][1], Y_coordinate[a, b][1]]
    p2 = [X_coordinate[c, d][1], Y_coordinate[c, d][1]]
    p = p1 - p2
    mag = norm(p)
```

```
    return mag
end

function HLL(U_L, U_R, n)
    rho_L, u_L, v_L, p_L = U_flux_decompose(U_L)
    rho_R, u_R, v_R, p_R = U_flux_decompose(U_R)
    if p_L/rho_L*p_R/rho_R < 0
        println ("rho=",rho_L, "u=",u_L, "v=",v_L, "p=",p_L)
        println ("rho=",rho_R, "u=",u_R, "v=",v_R, "p=",p_R)
        println (U_L)
        println (U_R)
    end
    a_L = sqrt(gamma * p_L/rho_L)
    a_R = sqrt(gamma * p_R/rho_R)
    a_tilde = (a_L + a_R) / 2
    un_L=u_L*n[1]+v_L*n[2]
    un_R=u_R*n[1]+v_R*n[2]
    u_tilde=(un_L+un_R)/2

    S_L=min(un_L-a_L,u_tilde-a_tilde)
    S_R=max(un_R+a_R,u_tilde+a_tilde)

    PHI_L =U_L
    PHI_L[end]=p_L * (gamma / (gamma - 1)) + (rho_L * 0.5 * (u_L^2 + v_L^2))
    F_L=un_L .* PHI_L .+ [0, p_L*n[1], p_L*n[2], 0]

    PHI_R =U_R
    PHI_R[end]=p_R * (gamma / (gamma - 1)) + (rho_R * 0.5 * (u_R^2 + v_R^2))
    F_R=un_R .* PHI_R .+ [0, p_R*n[1], p_R*n[2], 0]

    if S_L >=0
        F = F_L
    elseif S_R <= 0
        F = F_R
    else
        F=(S_R .* F_L .- S_L .* F_R .+ S_L*S_R .* (U_R-U_L)) ./ (S_R-S_L)
    end

    return F
end

function normal_vector(a, b, c, d)
    a=a-2
    b=b-2
    c=c-2
    d=d-2
    p1 = [X_coordinate[a, b][1], Y_coordinate[a, b][1]]
```

```
    p2 = [X_coordinate[c, d][1], Y_coordinate[c, d][1]]
    p = p1 − p2
    magnitude = norm(p)
    unit_vector = p / magnitude
    return unit_vector
end

function main(dt, t_end)
    # initialize
    rho, u, v, p = initial_conditions ()
    U_bar = cat(rho, rho .∗ u, rho .∗ v, 0.5 .∗ rho .∗ (u.^2 .+ v.^2) .+ p ./ (
        gamma − 1); dims=3)
    for t in 0:dt:t_end
        println ("t=$t")
        for i in 3:IMAX+1
            @threads for j in 3:JMAX+1
                U = U_bar[i, j, :]
                omega = area(i, j)
                # println (i,j)
                U_L = U_bar[i, j +1,:] .+ 0.5 .∗ (U_bar[i, j +1,:] .− U_bar[i, j +2,:])
                U_R = U_bar[i, j ,:] .+ 0.5 .∗ (U_bar[i, j ,:] .− U_bar[i, j −1,:])
                # println (U_bar[i, j ,:])
                # println (U_bar[i, j −1,:])
                # left face
                S_left = length(i, j+1, i+1, j+1)
                n_left = normal_vector(i, j+1, i, j)
                F_left = HLL(U_R, U_L, n_left)
                # if j==JMAX
                #       println (F_left)
                # end

                U_L = U_bar[i, j ,:] .+ 0.5 .∗ (U_bar[i, j ,:] .− U_bar[i, j +1,:])
                U_R = U_bar[i, j −1,:] .+ 0.5 .∗ (U_bar[i, j −1,:] .− U_bar[i, j −2,:])
                # println ("right face")
                # right face
                S_right = length(i, j, i+1, j)
                n_right = normal_vector(i, j, i, j+1)
                F_right = HLL(U_L, U_R, n_right)
                # if j==JMAX
                #       println (F_right)
                # end

                U_L = U_bar[i, j ,:] .+ 0.5 .∗ (U_bar[i, j ,:] .− U_bar[i−1, j ,:])
                U_R = U_bar[i+1, j ,:] .+ 0.5 .∗ (U_bar[i+1, j ,:] .− U_bar[i+2, j ,:])
                # println (U_R)
                # upper face
                S_upper = length(i+1, j, i+1, j+1)
```

```
            n_upper = normal_vector(i+1, j, i, j)
            F_upper = HLL(U_L, U_R, n_upper)

            U_L = U_bar[i-1, j,:] .+ 0.5 .* (U_bar[i-1, j,:] .- U_bar[i-2, j,:])
            U_R = U_bar[i, j,:] .+ 0.5 .* (U_bar[i, j,:] .- U_bar[i+1, j,:])
            # println (U_R)
            # down face
            S_down = length(i, j, i, j+1)
            n_down = normal_vector(i, j, i+1, j)
            F_down = HLL(U_R, U_L, n_down)

            U_temp = U .- dt / omega .* (S_left .* F_left .+ S_right .* F_right
                .+ S_upper .* F_upper .+ S_down .* F_down)
            U_bar[i, j,:] = U_temp
            # if j==JMAX
            #     println (F_left)
            #     println (dt / omega .*(S_left .* F_left .+ S_right .* F_right
                .+ S_upper .* F_upper .+ S_down .* F_down))
            # end
        end
    end

    #define some monitor
    println ("rho=",U_bar[80,40,1],"u=",U_bar[80,40,2]/U_bar [80,40,1], "p=",(
        U_bar[80,40,4] - 0.5 * U_bar[80,40,1] * ((U_bar [80,40,2]/U_bar [80,40,1])
        ^2 + (U_bar [80,40,3]/U_bar [80,40,1])^2)) * (gamma - 1))
    println ("rho=",U_bar[80,end,1],"u=",U_bar[80,end,2]/U_bar[80,end,1], "p=",(
        U_bar[80,end,4] - 0.5 * U_bar[80,end,1] * ((U_bar[80,end,2]/U_bar[80,end
        ,1])^2 + (U_bar[80,end,3]/U_bar[80,end,1])^2)) * (gamma - 1))
    println ("rho=",U_bar[80,end-3,1],"u=",U_bar[80,end-3,2]/U_bar[80,end-3,1],"
        p=",(U_bar[80,end-3,4] - 0.5 * U_bar[80,end-3,1] * ((U_bar[80,end-3,2]/
        U_bar[80,end-3,1])^2 + (U_bar[80,end-3,3]/U_bar[80,end-3,1])^2)) * (
        gamma - 1))

    # boundary conditions
    for i in 3:IMAX+1
        # U_bar[i, 2,:] = U_bar[i, 3,:]
        # Inviscid wall
        n_right = normal_vector(i, 3, i, 4)
        n_upper = normal_vector(i+1, 3, i, 3)
        A = [1 0 0 0; 0 n_right[1] n_right[2] 0; 0 n_upper[1] n_upper[2] 0; 0 0
            0 1]
        B1 = [1 0 0 0; 0 0 0 0; 0 0 1 0; 0 0 0 1]
        B2 = [1 0 0 0; 0 -1 0 0; 0 0 1 0; 0 0 0 1]
        # println (A)
        C = [1 0 0 0; 0 n_right[1] n_upper[1] 0; 0 n_right[2] n_upper[2] 0; 0 0
            0 1]
```

```
            # coe_matrix = C * B1 * A
            # println (U_bar[i, 3,:])
            # println (B2 *A *U_bar[i, 3,:])
            U_bar[i, 2,:]= C * B2 * A*U_bar[i, 3,:]
            U_bar[i, 1,:]= C * B2 * A*U_bar[i, 4,:]
            # P_U_bar = coe_matrix * U_bar[i, 3,:]
            # U_bar[i, 1,:] = 3 .* U_bar[i, 2,:] − 2 .* P_U_bar
        end
        # Supersonic inflow
        for i in 3:IMAX+1
            rho_in = p_inf / (T_inf* R_G)
            u_in = M_inf * sqrt (gamma * R_G * T_inf)
            v_in = 0
            p_in = p_inf
            U_bar[i, end,1]=U_bar[i, end−1,1]=rho_in
            U_bar[i, end,2]=U_bar[i, end−1,2]=rho_in*u_in
            U_bar[i, end,3]=U_bar[i, end−1,3]=rho_in*v_in
            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)
            # println (U_bar[i, end ,:])
        end
        # trans−sonic outlet boundary condition
        for j in 3:JMAX+1
            U_bar[1, j ,:] = U_bar[3, j ,:]
            U_bar[2, j ,:] = U_bar[3, j ,:]
            U_bar[end, j ,:] = U_bar[IMAX+1, j,:]
            U_bar[end−1, j ,:] = U_bar[IMAX+1, j,:]
        #=
            rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[IMAX+1, j,:])
            a_out=sqrt (gamma * p_out/rho_out)
            M_out=sqrt(u_out^2+v_out^2)/a_out
            n=normal_vector(IMAX+2,j,IMAX+1,j)
            if M_out >= 1
                U_bar[end, j ,:] = U_bar[IMAX+1, j,:]
                U_bar[end−1, j ,:] = U_bar[IMAX+1, j,:]
            else
                p=p_b
                rho=rho_out+(p_b−p_out)/a_out^2
                u=u_out+(p_b−p_out)*n[1]/(rho_out*a_out)
                v=v_out+(p_b−p_out)*n[2]/(rho_out*a_out)
                P_U_bar=[rho,rho*u,rho*v, 0.5* rho * (u^2 + v^2) + p / (gamma − 1)]
                U_bar[end−1, j ,:]=2 .*P_U_bar .− U_bar[IMAX+1, j,:]
                U_bar[end−1, j ,:]=4 .*P_U_bar .− 3 .*U_bar[IMAX+1, j,:]
            end
            rho_out, u_out, v_out, p_out=U_flux_decompose(U_bar[3, j ,:])
            a_out=sqrt (gamma * p_out/rho_out)
            M_out=sqrt(u_out^2+v_out^2)/a_out
```

```
            n=normal_vector(3,j ,4, j )
            if  M_out >= 1
                U_bar[1,  j ,:]  = U_bar[3,  j ,:]
                U_bar[2,  j ,:]  = U_bar[3,  j ,:]
            else
                p=p_b
                rho=rho_out+(p_b−p_out)/a_out^2
                u=u_out+(p_b−p_out)∗n[1]/(rho_out∗a_out)
                v=v_out+(p_b−p_out)∗n[2]/(rho_out∗a_out)
                P_U_bar=[rho,rho∗u,rho∗v, 0.5∗ rho ∗ (u^2 + v^2) + p /  (gamma − 1)]
                U_bar[2,  j ,:]=2  .∗  P_U_bar .− U_bar[3,  j ,:]
                U_bar[1,  j ,:]=4  .∗  P_U_bar .− 3  .∗U_bar[3,  j ,:]
            end
            =#
        end

    end

    return  U_bar
end


U_bar = main(1e−6, 5e−3)

rho_final  = U_bar[:,  :,1]
u_final  = U_bar[:,  :,2]  ./  rho_final
v_final  = U_bar[:,  :,3]  ./  rho_final
p_final  = (U_bar[:,  :,4]  .−  0.5  .∗  rho_final  .∗  ( u_final .^2  .+  v_final .^2) )  .∗  (
    gamma − 1)
#  println ( rho_final )


function  plot_colored_square (x_coords,  y_coords,  color )
    plot !(x_coords,  y_coords,  seriestype  =  :shape,  fillcolor  =  color ,  linecolor  =  :
        transparent )
end

function  get_color ( value ,  min_value,  delta )
    return  cgrad (: rainbow) [( value  −  min_value)  /  delta ]
end

# normalized_rho  =  ( rho_final  .−  minimum(rho_final))  ./  (maximum(rho_final) −
    minimum(rho_final))
p_rho  =  plot ( figsize  =  (800,  800), legend=false )
count_greater_than_10  =  0
for  i  in  3:IMAX+1
    for  j  in  3:JMAX+1
```

```
            X = X_coordinate
            Y = Y_coordinate

            I_coordinate=i-2
            J_coordinate=j-2
            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]]
            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]]

            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])
            color = get_color(rho_final[i, j], minimum(rho_final[3:IMAX+1,3:JMAX+1]),
                 Delta)


            plot_colored_square(cell_x, cell_y, color)
            global count_greater_than_10

            if rho_final[i, j] > 1
                 count_greater_than_10 += 1
            end
        end
end


println("Number of rho values greater than 10: ", count_greater_than_10)

colorbar_ticks = range(minimum(rho_final[3:IMAX+1,3:JMAX+1]), stop=maximum(
    rho_final[3:IMAX+1,3:JMAX+1]), length=11)
plot!(p_rho, color=:rainbow, colorbar=:right, colorbar_ticks=colorbar_ticks,
     colorbar_tick_labels=[@sprintf("%.2f", x) for x in colorbar_ticks])
# Colorbar(p_rho, pltobj)
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])))")
savefig(p_rho,"rho.png")



p_p=plot(figsize = (800, 800),legend=false)
for i in 3:IMAX+1
     for j in 3:JMAX+1
            X = X_coordinate
            Y = Y_coordinate
```

```
            I_coordinate=i−2
            J_coordinate=j−2
            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]]
            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]]

            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])
            color = get_color(p_final[i, j], minimum(p_final[3:IMAX+1,3:JMAX+1]),
                Delta)


            plot_colored_square(cell_x, cell_y, color)
      end
end
 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])))")
 savefig(p_p,"p.png")



p_u=plot(figsize = (800, 800),legend=false)
for i in 3:IMAX+1
      for j in 3:JMAX+1
            X = X_coordinate
            Y = Y_coordinate
            I_coordinate=i−2
            J_coordinate=j−2
            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]]
            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]]

            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])
            color = get_color(u_final[i, j], minimum(u_final[3:IMAX+1,3:JMAX+1]),
                Delta)


            plot_colored_square(cell_x, cell_y, color)
```

```
        end
end
 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]))))”)
 savefig(p_u,”u.png”)
```