THE HONG KONG POLYTECHNIC UNIVERSITY
香港理工大學

Project report: 1D Sod Shock Tube
Subject: Advanced Computational Fluid Dynamics **(AAE6201-20241-A)**
Date: 28/10/2024

1 Problem description

Considering the 1D Euler equations governing the flow of an ideal gas:

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:

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

where ρ is the density, u is the velocity, p is the pressure, and E is the total energy per unit volume.

2 Requirements

1. Compare the numerical solution with the exact solution at different time instants (do not let the wave arrive at the boundaries).
2. Use different schemes for space discretization (around 100 grid points).
3. Use first-order difference formula for time discretization.
4. Use S-W or L-F flux vector splitting for original flux and characteristic flux, and compare their difference.

3 Exact solution

According to the knowledge of gas dynamics, three types of waves may appear in the Sod shock tube:

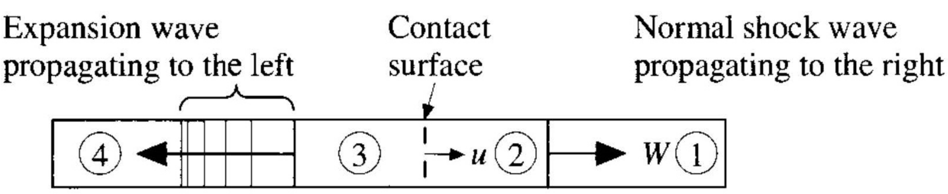

Figure 1: Regions in sod shock tube

1. shock wave. After passing through the shock wave, the density, velocity, and pressure of the fluid all experience sudden changes, satisfying the Rankine-Hugoniot (R-H) relation;
2. contact discontinuity. After passing through the contact discontinuity, only the density of the fluid changes suddenly, while the velocity and pressure remain unchanged;

3. expansion wave or rarefaction wave. It is an entropy wave with continuous and smooth internal physical quantities, and the physical quantities at the head and tail are continuous but the derivatives are discontinuous (weak discontinuity), with the Riemann invariants remaining invariant. Considering the general case, there are five possibilities of combination waves in the tube. According to the conservation of mass flux, momentum flux, and energy flux, by taking a control volume moving with the shock wave and sufficiently small in thickness, equations can be written and solved for analysis following different scenarios.
In this project we only consider a circumstance that expansion waves and shock wave occur at two end end of the shock tube and propagate to different direction. The relationships can be categorized and written as follows for the shock wave at right and expansion waves at left : In Region 1 and 3

$$p^* / \left(\rho^{*L}\right)^\gamma = p_1 / \left(\rho_1\right)^\gamma u_1 + \frac{2c_1}{\gamma-1} = u^* + \frac{2c^L}{\gamma-1} \tag{4}$$

in which

$$c^L = \sqrt{\gamma p^* / \rho^{*L}} \tag{5}$$

In Region 2 and 4

$$\begin{aligned} &\rho_2\left(u_2 - Z_2\right) = \rho^{*R}\left(u^* - Z_2\right) \\ &\rho_2 u_2\left(u_2 - Z_2\right) + p_2 = \rho^{*R} u^*\left(u^* - Z_2\right) + p^* \\ &E_2\left(u_2 - Z_2\right) + u_2 p_2 = E^{*R}\left(u^* - Z_2\right) + p^* u^* \end{aligned} \tag{6}$$

In this project, analytic solutions was calculated using nucci2023's code[1].

4 Numerical simulation

4.1 Basic equations

The original equations can be written as follow:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} = 0 \tag{7}$$

in which

$$\mathbf{U} = \begin{bmatrix} \rho \\ \rho u \\ \rho E \end{bmatrix}, \mathbf{F} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho u E + p u \end{bmatrix} \tag{8}$$

set A=$\frac{\partial F}{\partial U}$, we have

$$\frac{\partial \mathbf{U}}{\partial t} + A\frac{\partial \mathbf{U}}{\partial x} = 0 \tag{9}$$

in which

$$A = \begin{bmatrix} 0 & 1 & 0 \\ \left(\frac{\gamma-3}{2}\right) u^2 & (3-\gamma)u & \gamma-1 \\ (\gamma-1)u^3 - \gamma u E & \gamma E - \frac{3(\gamma-1)}{2} u^2 & \gamma u \end{bmatrix} \tag{10}$$

it should be noticed that E can be written as

$$E = \frac{p}{(\gamma - 1)\rho} + \frac{u^2}{2} \tag{11}$$

assuming that the fluid is ideal gas. A is diagonalizable,

$$\mathbf{F} = R\Lambda^{+}R^{-1}\mathbf{U} + R\Lambda^{-}R^{-1}\mathbf{U} \tag{12}$$

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}^{+}}{\partial x} + \frac{\partial \mathbf{F}^{-}}{\partial x} = 0 \tag{13}$$

where

$$\boldsymbol{F}^{+} = R\Lambda^{+}R^{-1}\mathbf{U}, \quad \boldsymbol{F}^{-} = R\Lambda^{-}R^{-1}\mathbf{U} \tag{14}$$

the right eigenvectors

$$\boldsymbol{R} = \begin{bmatrix} 1 & 1 & 1 \\ u - a & u & u + a \\ H - ua & u^2/2 & H + ua \end{bmatrix} \tag{15}$$

4.2 Second order upwind scheme

Consider a general scheme in conservative form

$$\frac{\partial \mathbf{U}}{\partial t} = -\mathbf{A}\frac{\mathbf{U}^n_{i+\frac{1}{2}} - \mathbf{U}^n_{i-\frac{1}{2}}}{\Delta x} \tag{16}$$

Consider the second order upwind scheme with limiter

$$\mathbf{U}^n_{i+1/2} = \mathbf{U}^n_i + \frac{1}{2}\varphi(r_i)(\mathbf{U}^n_i - \mathbf{U}^n_{i-1}), \quad r_i = \frac{\mathbf{U}_{i+1} - \mathbf{U}_i}{\mathbf{U}_i - \mathbf{U}_{i-1}} \tag{17}$$

if $\mathbf{U}_i - \mathbf{U}_{i-1} = 0$ set r=1.

$$\frac{\partial \mathbf{U}}{\partial t} = -\frac{\mathbf{A}}{\Delta x}\left[\left(1 + \frac{1}{2}\varphi(r_i) - \frac{1}{2}\frac{\varphi(r_{i-1})}{r_{i-1}}\right)(\mathbf{U}^n_i - \mathbf{U}^n_{i-1})\right] \tag{18}$$

substitute equation 18 into equation 13 we have

$$\begin{aligned} \frac{\partial \mathbf{U}}{\partial t} = -\frac{\mathbf{A}^{+}}{\Delta x}\left[\left(1 + \frac{1}{2}\varphi(r_i) - \frac{1}{2}\frac{\varphi(r_{i-1})}{r_{i-1}}\right)(\mathbf{U}^n_i - \mathbf{U}^n_{i-1})\right] \\ +\frac{\mathbf{A}^{-}}{\Delta x}\left[\left(1 + \frac{1}{2}\varphi(r_i) - \frac{1}{2}\frac{\varphi(r_{i+1})}{r_{i+1}}\right)(\mathbf{U}^n_i - \mathbf{U}^n_{i+1})\right] \end{aligned} \tag{19}$$

Which is

$$\begin{aligned} \mathbf{U}^{n+1}_i = -\mathbf{A}^{+}\frac{\Delta t}{\Delta x}\left[\left(1 + \frac{1}{2}\varphi(r_i) - \frac{1}{2}\frac{\varphi(r_{i-1})}{r_{i-1}}\right)(\mathbf{U}^n_i - \mathbf{U}^n_{i-1})\right] \\ +\mathbf{A}^{-}\frac{\Delta t}{\Delta x}\left[\left(1 + \frac{1}{2}\varphi(r_i) - \frac{1}{2}\frac{\varphi(r_{i+1})}{r_{i+1}}\right)(\mathbf{U}^n_i - \mathbf{U}^n_{i+1})\right] + \mathbf{U}^n_i \end{aligned} \tag{20}$$

4.3 Van Leer limiter

Use Van Leer's limiter,

$$\varphi(r) = \frac{r + |r|}{1 + |r|} \tag{21}$$

4.4 Lax-Wendroff scheme

Discretize equation 9 with Lax-Wendroff scheme

$$\mathbf{U}_i^{n+1} = \mathbf{U}_i^n - \mathbf{A}\frac{\Delta t}{2\Delta x}(\mathbf{U}_{i+1}^n - \mathbf{U}_{i-1}^n) + \mathbf{A}^2\frac{\Delta t^2}{2\Delta x^2}(\mathbf{U}_{i+1}^n - 2\mathbf{U}_i^n + \mathbf{U}_{i-1}^n) \tag{22}$$

4.5 Steger-Warming flux vector splitting

As mentioned before, the Jacobian matrix A for Euler equations can be diagonalized.

$$\mathbf{A} = \mathbf{P}\Lambda\mathbf{P}^{-1} \tag{23}$$

with the left eigenvectors

$$\boldsymbol{R}^{-1} = \frac{(\gamma-1)}{2a^2}\begin{bmatrix} H + \frac{a(u-a)}{\gamma-1} & -u - \frac{a}{\gamma-1} & 1 \\ \frac{4}{\gamma-1}a^2 - 2H & 2u & -2 \\ H - \frac{a(u-a)}{\gamma-1} & -u + \frac{a}{\gamma-1} & 1 \end{bmatrix} \tag{24}$$

split the eigenvalues as

$$\mathbf{\Lambda} = \mathbf{\Lambda}^+ + \mathbf{\Lambda}^- \quad \mathbf{\Lambda}^+ = \begin{bmatrix} \lambda_1^+ & & \\ & \ddots & \\ & & \lambda_m^+ \end{bmatrix} \quad \mathbf{\Lambda}^- = \begin{bmatrix} \lambda_1^- & & \\ & \ddots & \\ & & \lambda_m^- \end{bmatrix} \tag{25}$$

use Steger-Warming scheme to evaluate λ^+ and λ^-

$$\lambda_i^+ = \frac{1}{2}(\lambda_i + |\lambda_i|), \lambda_i^- = \frac{1}{2}(\lambda_i - |\lambda_i|) \tag{26}$$

4.6 Lax-Friedrichs flux vector splitting

Lax-Friedrichs flux vector splitting the positive and negative Jacobian matrix are created following

$$A^+ = \frac{1}{2}(A + \lambda_{max}I) \tag{27}$$

as well as

$$A^- = \frac{1}{2}(A - \lambda_{max}I) \tag{28}$$

For a local L-F splitting, λ_{max} is evaluated at each point.

5 Simulation results

In this section, python code were established to solve Sod shock tube, the original code see also appendix. In numerical solving, Δt was set to 1×10^{-5}, and the solving area was split into 200 cells, simulation results at $t = 0.001$ were shown in subsections.

5.1 Difference between space discretization schemes

As can be seen in Figure. 2, there are 4 density stage in the shock tube, the density changes linearly in the expansion wave between region 3 and 4, but changes suddenly in the contact surface and shock. Similar trend also occur in velocity distribution and pressure distribution, the difference is, velocity and pressure do not change at the contact surface.
Firstly, with the help from Van Leer's limiter, oscillation was avoided in second order upwind scheme. Secondly, second order upwind scheme didn't have high enough accuracy to capture shock wave at right. In other wards, second order upwind scheme is to dissipative shock waves evanish after iterations. Thirdly, second order upwind scheme with Van Leer limiter is a faster scheme in this problem if flow direction is taken into consideration.
On the contrary, Lax-Wendroff scheme without limiter do have oscillation, but oscillation will not occur at every discontinuity, the most clear oscillation shew up at the contact surface. Compare with second order upwind scheme, Lax-Wendroff scheme can capture the shock wave structure with out oscillation and the dissipation is also suppressed well.

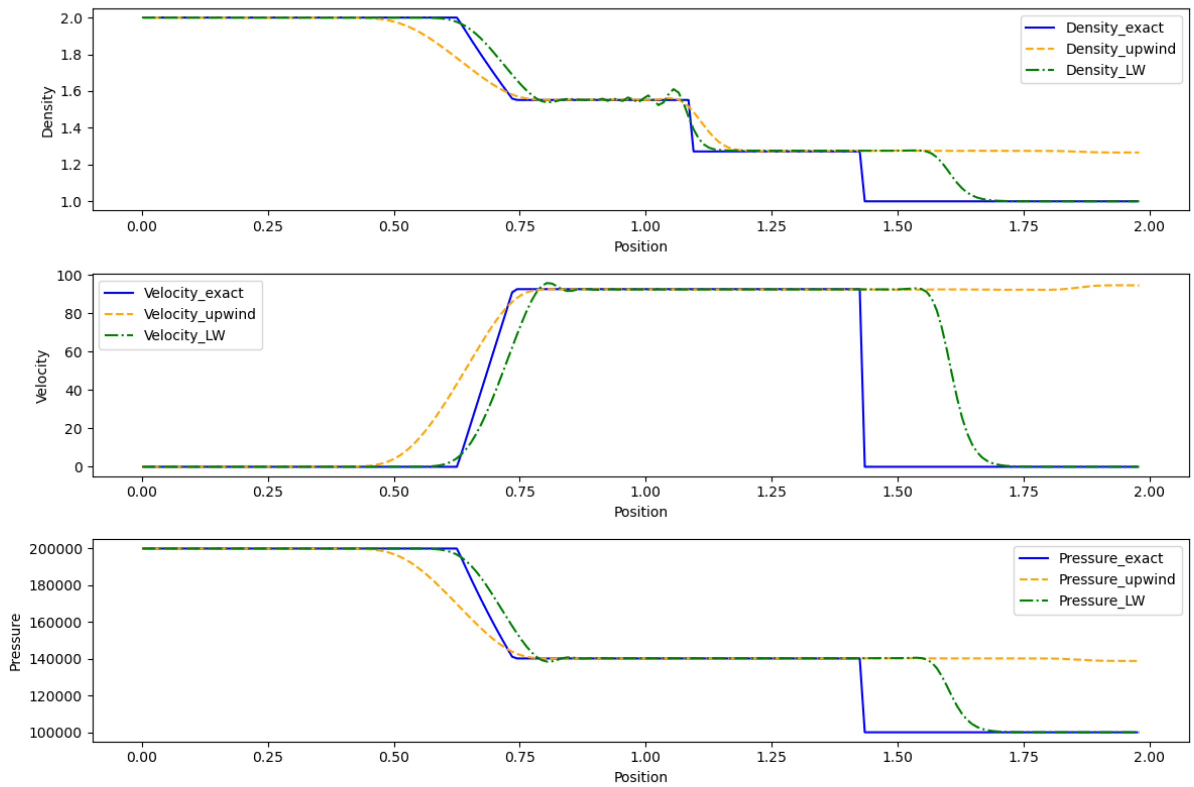

Figure 2: Regions in sod shock tube with different space discretization schemes

5.2 Difference between flux vector splitting schemes

S-W or L-F flux vector splitting as used and Lax-Wendroff scheme was used for the space discretization. As is shown in Figure. 3, there no difference between Lax-Friedrichs scheme and Steger-Warming scheme, this result come up with the characteristic of Lax-Wendroff scheme. Lax-Wendroff scheme can be viewed as a combination of central scheme of first derivative and second derivative both central scheme have no dissipation, and don't need flux vector splitting.

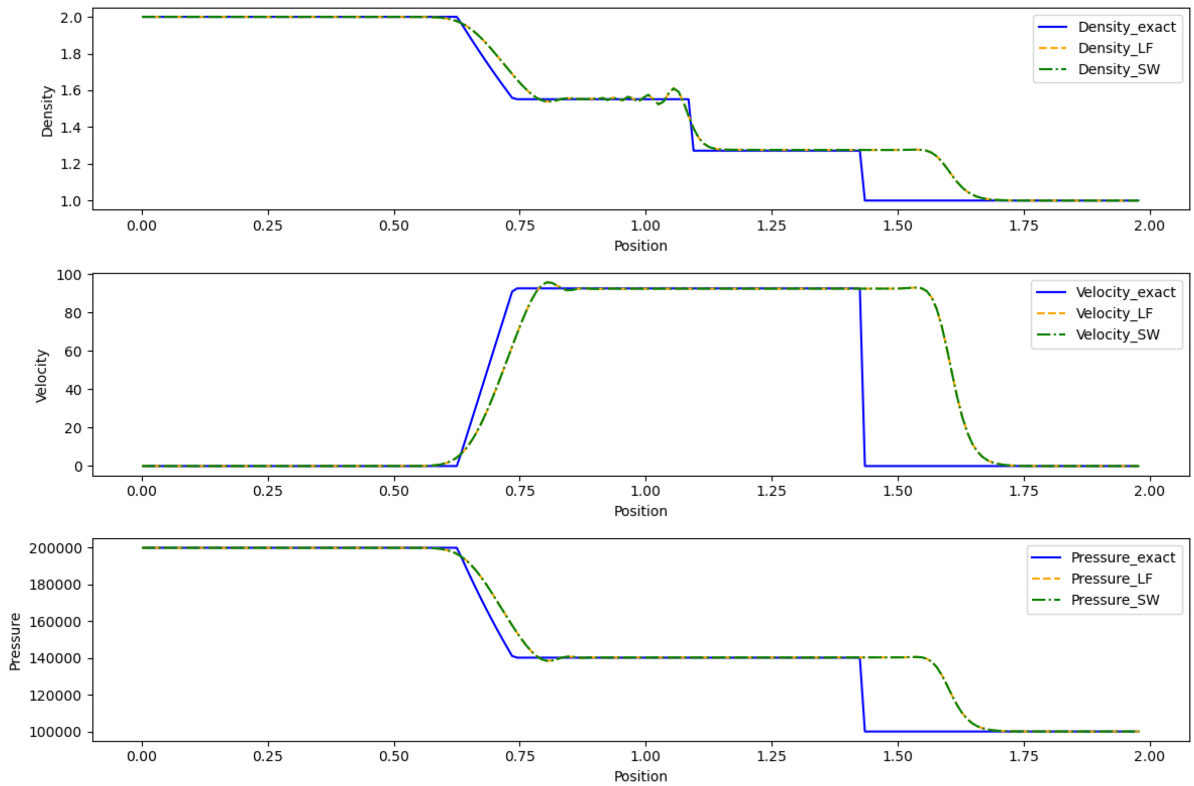

Figure 3: Regions in sod shock tube with different flux vector splitting schemes

It can be easily proved, in equation 22, discretization the second term at the right hand side.

$$\begin{aligned}
&-\mathbf{A}\frac{\Delta t}{2\Delta x}(\mathbf{U}_{i+1}^n-\mathbf{U}_{i-1}^n)\\
&=-\mathbf{A}^+\frac{\Delta t}{2\Delta x}(\mathbf{U}_{i+1}^n-\mathbf{U}_{i-1}^n)+\mathbf{A}^-\frac{\Delta t}{2\Delta x}(\mathbf{U}_{i-1}^n-\mathbf{U}_{i+1}^n)\\
&=-\mathbf{A}^+\frac{\Delta t}{2\Delta x}(\mathbf{U}_{i+1}^n-\mathbf{U}_{i-1}^n)-\mathbf{A}^-\frac{\Delta t}{2\Delta x}(\mathbf{U}_{i+1}^n-\mathbf{U}_{i-1}^n)\\
&=-(\mathbf{A}^+\mathbf{A}^-)\frac{\Delta t}{2\Delta x}(\mathbf{U}_{i+1}^n-\mathbf{U}_{i-1}^n)\\
&=-\mathbf{A}\frac{\Delta t}{2\Delta x}(\mathbf{U}_{i+1}^n-\mathbf{U}_{i-1}^n)
\end{aligned} \tag{29}$$

for the third term on the right hand side, the square of $\mathbf{A}^+ + \mathbf{A}^-$ gives $\mathbf{A}^2$. Flux vector splitting makes no difference as for Lax-Wendroff scheme.

In order to indicate the difference between flux vector splitting schemes, second order upwind scheme was used as a space discretization scheme. As shown in Figure 4 Steger-Warming scheme generally provides a batter discontinuity-capturing capabilities but theoretically it can be less accurate in resolving contact discontinuities and shear waves. On the opposite Lax-Friedrichs tends to introduce more numerical dissipation, which can smooth out sharp features but provides robust stability.

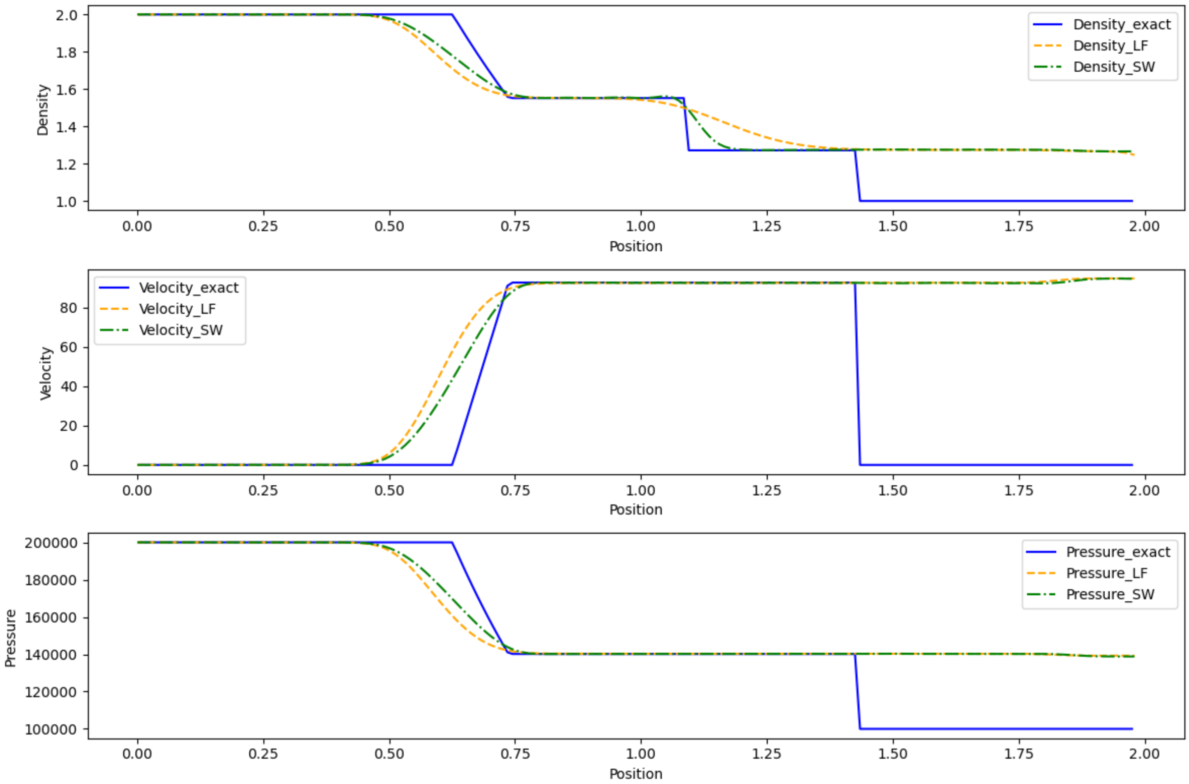

Figure 4: Regions in sod shock tube with different flux vector splitting schemes

References

[1] M. Nucci, mnucci32/SodShockTube, original-date: 2017-01-30T02:49:24Z (Jul. 2023). URL `https://github.com/mnucci32/SodShockTube`

6 Appendix

6.1 Space discretization

```
import numpy as np
import matplotlib.pyplot as plt
import sys
from scipy.linalg import eig
gamma=1.4

# Initialize
def initial_conditions(nx, x):
    rho = np.ones(nx)
    u = np.zeros(nx)
```

```
    p = np.ones(nx)

    # left
    rho[:nx//2] = 2.0
    p[:nx//2] = 200000

    # right
    rho[nx//2:] = 1
    p[nx//2:] = 100000

    return rho, u, p

# flux function
def flux_F(rho, u, p):
    F1 = rho * u
    F2 = rho * u**2 + p
    F3 = (p / (gamma - 1)) + 0.5 * rho * u**2 + p * u
    return np.array([F1, F2, F3])

def flux_U(rho, u, p):
    U1 = rho
    U2 = rho * u
    U3 = (p / (gamma - 1)) + 0.5 * rho * u**2
    return np.array([U1, U2, U3])

# set up Steger-Warming flux vector splitting
def steger_warming_flux_split(rho, u, p):
    c = np.sqrt(gamma * p / rho)
    lam = np.array([[u - c, 0,0],
                    [0, u, 0],
                    [0, 0, u + c]]
                   )
    lam_plus=0.5*(lam+np.abs(lam))
    lam_minus=0.5*(lam-np.abs(lam))
    # eigen values and vectors

    # Jocobian Metrix
    A1=np.array([0, 1, 0])
    A2=np.array([[((gamma-3)/2)*u**2, (3-gamma)*u, (gamma-1)]])
    A3=np.array([(gamma-1)*u**3-gamma*u*(p/((gamma-1)*rho)+0.5*u**2),

    gamma*(p/((gamma-1)*rho)+0.5*u**2)-(3*(gamma-1)/2)*u**2, gamma*u])
    A= np.array([A1,A2,A3])
    # eigenvalues, eigenvectors = eig(A)

    H=c**2/(gamma-1)+0.5*u**2
    P1=np.array([1,1,1])
```

```
    P2=np.array([u-c,u,u+c])
    P3=np.array([H-u*c,0.5*u**2,H+u*c])

    P= np.array([P1,P2,P3])

    A_plus=P @ lam_plus @ np.linalg.inv(P)
    A_minus=P @ lam_minus @ np.linalg.inv(P)
    # print(lam_plus,lam_minus)

    return A,A_plus, A_minus

# set up Van Leer limiter
def van_leer_limiter(x,U,direction):
    if direction==-1:
        numerator = U[:,x+1] - U[:,x]
        denominator = U[:,x] - U[:,x-1]
    if direction==1:
        numerator = U[:,x-1] - U[:,x]
        denominator = U[:,x] - U[:,x+1]
    if np.any(denominator == 0):
        r=np.array([1,1,1])
    else:
        r=numerator/denominator
    if np.any(r == 0):
        r=np.array([1,1,1])
    phi = (r + np.abs(r)) / (1 + np.abs(r))
    return r,phi

# set up superBee limiter
def superBee(x,U,direction):
    if direction==-1:
        numerator = U[:,x+1] - U[:,x]
        denominator = U[:,x] - U[:,x-1]
    if direction==1:
        numerator = U[:,x-1] - U[:,x]
        denominator = U[:,x] - U[:,x+1]
    if np.any(denominator == 0):
        r=np.array([1,1,1])
    else:
        r=numerator/denominator
    if np.any(r == 0):
        r=np.array([1,1,1])

    phi1=max(0, min(2 * r[0], 1), min(r[0], 2))
    phi2=max(0, min(2 * r[1], 1), min(r[1], 2))
    phi3=max(0, min(2 * r[2], 1), min(r[2], 2))
    phi=np.array([phi1,phi2,phi3])
```

```
    return r,phi



# set up main function
def solve_sod_shock_tube_upwind(nx, L, dt, t_end):
    dx = L / (nx − 1)
    # dt = t_end / (nt − 1)
    nt=int((t_end/dt)+1)
    CFL=dt/dx
    x_list = np.linspace(0, L, nx)

    # initialize values
    rho, u, p = initial_conditions(nx, x_list)
    U= flux_U(rho, u, p)
    U_prime=U
    # time loop
    for n in range(nt):
        for x in range(nx−2):#
            # print(" Iteration report nx={},nt={}".format(x,n))
            if x ==0 :
                continue
            # cell flux
            A,A_plus, A_minus =
             steger_warming_flux_split(rho[x], u[x], p[x])

             r_x_positive , phi_x_positive = van_leer_limiter(x,U,1)
            r_x_negtive , phi_x_negtive= van_leer_limiter(x,U,−1)
            r_x_minus1,phi_x_minus1=van_leer_limiter(x−1,U,−1)
            r_x_plus1,phi_x_plus1= van_leer_limiter(x+1,U,1)

            #positive eigenvalue
             positive_term =−1∗CFL∗((np.array([1,1,1])+0.5∗phi_x_positive−0.5∗
                 phi_x_minus1/r_x_minus1)

            ∗np.dot(A_plus,(U[:,x]−U[:,x−1])))
            #negtive eigenvalue
            negtive_term=CFL∗((1+0.5∗phi_x_negtive−0.5∗phi_x_plus1/r_x_plus1))∗

            np.dot(A_minus,(U[:,x]−U[:,x+1]))

            # print(phi_x_minus1,r_x_minus1)

            U_prime[:,x]=positive_term+negtive_term+U[:,x]

            U[:,x]=U_prime[:,x]
            rho[x]=U[0,x]
```

```
            u[x]=U[1,x]/U[0,x]
            p[x]=(U[2,x]−0.5∗rho[x]∗u[x]∗u[x])∗(gamma−1)

            # print (rho[x], u[x], p[x])
    return x_list , rho,u,p

# set up main function
def solve_sod_shock_tube_SW(nx, L, dt, t_end):
    dx = L / (nx − 1)
    # dt = t_end / (nt − 1)
    nt=int((t_end/dt)+1)
    CFL=dt/dx
    x_list = np. linspace (0, L, nx)

    # initialize values
    rho, u, p = initial_conditions (nx, x_list )
    U= flux_U(rho, u, p)
    U_prime=U
    # time loop
    for n in range(nt):
        for x in range(nx−2):#
            # print (" Iteration report nx={},nt={}".format(x,n))
            if x ==0 :
                continue
            # cell flux
            A,A_plus, A_minus = steger_warming_flux_split (rho[x], u[x], p[x])


            #positive propagating
            positive_term= −1∗CFL/2∗np.dot(A_plus,(U[:,x+1]−U[:,x−1]))
            #negtive propagating
            negtive_term= CFL/2∗np.dot(A_minus,(U[:,x−1]−U[:,x+1]))

            # print (A_minus+A_plus,A)
            #positive propagating
            Linear_term= −1∗CFL/2∗np.dot(A,(U[:,x+1]−U[:,x−1]))
            A_square=A @ A
            square_term= CFL∗∗2/2∗np.dot(A_square,(U[:,x+1]−2∗U[:,x]+U[:,x−1]))

            # square_term= CFL∗∗2/2∗np.dot(A_square,(phi_x_plus1∗(U[:,x+1]−U[:,x])
                +phi_x_positive∗(U[:,x]−U[:,x−1])))


            U_prime[:,x]=positive_term+negtive_term+square_term+U[:,x]

            U[:,x]=U_prime[:,x]
            rho[x]=U[0,x]
```

```
                u[x]=U[1,x]/U[0,x]
                p[x]=(U[2,x]−0.5∗rho[x]∗u[x]∗u[x])∗(gamma−1)

                # print (U[:,x+1]−U[:,x−1],A,np.dot((U[:,x+1]−U[:,x−1]),A),Linear_term)
        return  x_list , rho,u,p


# solve Sod shock tube
x_list , rho_upwind,u_upwind,p_upwind = solve_sod_shock_tube_upwind(nx=200, L=2.0,
    dt=1e−5, t_end=0.001)
x_list , rho_SW,u_SW,p_SW = solve_sod_shock_tube_SW(nx=200, L=2.0, dt=1e−5, t_end
    =0.001)

##read exact soution
data = np. loadtxt ( 'exact . dat ', skiprows=1)

X_POS_PLOT = data[:, 0]
DENSITY = data[:, 1]
VELOCITY_X = data[:, 2]
PRESSURE = data[:, 3]

# plot
plt . figure ( figsize =(12, 8))

plt . subplot (3, 1, 1)
plt . plot (X_POS_PLOT[:−2], DENSITY[:−2],linestyle='solid', label='Density_exact ',
    color='blue ')
plt . plot ( x_list [:−2], rho_upwind[:−2], linestyle ='dashed', label='Density_upwind',
    color='orange')
plt . plot ( x_list [:−2], rho_SW[:−2], linestyle ='dashdot ', label='Density_LW', color='
    green ')
plt . xlabel ( ' Position ')
plt . ylabel ( 'Density ')
plt . legend ()

plt . subplot (3, 1, 2)
plt . plot (X_POS_PLOT[:−2], VELOCITY_X[:−2],linestyle='solid', label='Velocity_exact ',
    color='blue ')
plt . plot ( x_list [:−2], u_upwind[:−2], label='Velocity_upwind ', linestyle ='dashed',
    color='orange')
plt . plot ( x_list [:−2], u_SW[:−2], linestyle ='dashdot ', label='Velocity_LW', color='
    green ')
plt . xlabel ( ' Position ')
plt . ylabel ( ' Velocity ')
plt . legend ()

plt . subplot (3, 1, 3)
```

```
plt.plot(X_POS_PLOT[:-2], PRESSURE[:-2], label='Pressure_exact',linestyle='solid',
    color='blue')
plt.plot( x_list [:-2], p_upwind[:-2], label='Pressure_upwind', linestyle ='dashed',
    color='orange')
plt.plot( x_list [:-2], p_SW[:-2], linestyle='dashdot', label='Pressure_LW', color='
    green')
plt.xlabel('Position')
plt.ylabel('Pressure')
plt.legend()

plt. tight_layout ()
plt.show()
```

6.2 Flux vector splitting

```
import numpy as np
import matplotlib.pyplot as plt
import sys
from scipy.linalg import eig
gamma=1.4

# Initialize
def initial_conditions (nx, x):
    rho = np.ones(nx)
    u = np.zeros(nx)
    p = np.ones(nx)

    # left
    rho[:nx//2] = 2.0
    p[:nx//2] = 200000

    # right
    rho[nx//2:] = 1
    p[nx//2:] = 100000

    return rho, u, p

# flux function
def flux_F(rho, u, p):
    F1 = rho * u
    F2 = rho * u**2 + p
    F3 = (p / (gamma - 1)) + 0.5 * rho * u**2 + p * u
    return np.array([F1, F2, F3])

def flux_U(rho, u, p):
    U1 = rho
    U2 = rho * u
```

```
        U3 = (p / (gamma − 1)) + 0.5 ∗ rho ∗ u∗∗2
        return np.array([U1, U2, U3])

# set up Steger−Warming flux vector splitting
def steger_warming_flux_split(rho, u, p):
        c = np.sqrt(gamma ∗ p / rho)
        lam = np.array([[u − c, 0,0],
                        [0, u, 0],
                        [0, 0, u + c]]
                        )
        lam_plus=0.5∗(lam+np.abs(lam))
        lam_minus=0.5∗(lam−np.abs(lam))
        # eigen values and vectors

        # Jocobian Metrix
        A1=np.array([0, 1, 0])
        A2=np.array([((gamma−3)/2)∗u∗∗2, (3−gamma)∗u, (gamma−1)])
        A3=np.array([(gamma−1)∗u∗∗3−gamma∗u∗(p/((gamma−1)∗rho)+0.5∗u∗∗2),
            gamma∗(p/((gamma−1)∗rho)+0.5∗u∗∗2)−(3∗(gamma−1)/2)∗u∗∗2, gamma∗u])
        A= np.array([A1,A2,A3])
        # eigenvalues, eigenvectors = eig(A)

        H=c∗∗2/(gamma−1)+0.5∗u∗∗2
        P1=np.array ([1,1,1])
        P2=np.array([u−c,u,u+c])
        P3=np.array([H−u∗c,0.5∗u∗∗2,H+u∗c])

        P= np.array([P1,P2,P3])

        A_plus=P @ lam_plus @ np.linalg.inv(P)
        A_minus=P @ lam_minus @ np.linalg.inv(P)
        # print(lam_plus,lam_minus)

        return A,A_plus, A_minus


# set up Lax−Friedrichs flux vector splitting
def lax_friedrichs_flux_split (rho, u, p):
        c = np.sqrt(gamma ∗ p / rho)
        lam = np.array([[u − c, 0,0],
                        [0, u, 0],
                        [0, 0, u + c]]
                        )
        # eigen values and vectors

        # Jocobian Metrix
        A1=np.array([0, 1, 0])
```

```
        A2=np.array([((gamma-3)/2)*u**2, (3-gamma)*u, (gamma-1)])
        A3=np.array([(gamma-1)*u**3-gamma*u*(p/((gamma-1)*rho)+0.5*u**2),
            gamma*(p/((gamma-1)*rho)+0.5*u**2)-(3*(gamma-1)/2)*u**2, gamma*u])
        A= np.array([A1,A2,A3])
        # eigenvalues, eigenvectors = eig(A)

        lam_max=max(u-c,u,u+c)
        A_plus=0.5*(A+lam_max*np.eye(3))
        A_minus=0.5*(A-lam_max*np.eye(3))

        return A,A_plus, A_minus


# set up Van Leer limiter
def van_leer_limiter(x,U, direction):
    if direction ==-1:
        numerator = U[:,x+1] - U[:,x]
        denominator = U[:,x] - U[:,x-1]
    if direction ==1:
        numerator = U[:,x-1] - U[:,x]
        denominator = U[:,x] - U[:,x+1]
    if np.any(denominator == 0):
        r=np.array([1,1,1])
    else:
        r=numerator/denominator
    if np.any(r == 0):
        r=np.array([1,1,1])
    phi = (r + np.abs(r)) / (1 + np.abs(r))
    return r,phi

# set up superBee limiter
def superBee(x,U, direction):
    if direction ==-1:
        numerator = U[:,x+1] - U[:,x]
        denominator = U[:,x] - U[:,x-1]
    if direction ==1:
        numerator = U[:,x-1] - U[:,x]
        denominator = U[:,x] - U[:,x+1]
    if np.any(denominator == 0):
        r=np.array([1,1,1])
    else:
        r=numerator/denominator
    if np.any(r == 0):
        r=np.array([1,1,1])

    phi1=max(0, min(2 * r[0], 1), min(r[0], 2))
    phi2=max(0, min(2 * r[1], 1), min(r[1], 2))
```

```
        phi3=max(0, min(2 * r[2], 1), min(r[2], 2))
        phi=np.array([phi1,phi2,phi3])
        return r,phi


# set up main function
def solve_sod_shock_tube_SW(nx, L, dt, t_end):
    dx = L / (nx - 1)
    # dt = t_end / (nt - 1)
    nt=int((t_end/dt)+1)
    CFL=dt/dx
    x_list = np.linspace(0, L, nx)

    # initialize values
    rho, u, p = initial_conditions(nx, x_list)
    U= flux_U(rho, u, p)
    U_prime=U
    # time loop
    for n in range(nt):
        for x in range(nx-2):#
            # print(" Iteration report nx={},nt={}".format(x,n))
            if x ==0 :
                continue
            # cell flux
            A,A_plus, A_minus = steger_warming_flux_split(rho[x], u[x], p[x])

            r_x_positive, phi_x_positive=superBee(x,U,1)
            r_x_negtive,phi_x_negtive=superBee(x,U,-1)
            r_x_minus1,phi_x_minus1=superBee(x-1,U,-1)
            r_x_plus1,phi_x_plus1=superBee(x+1,U,1)

            #positive propagating
            positive_term= -1*CFL/2*np.dot(A_plus,(U[:,x+1]-U[:,x-1]))
            #negtive propagating
            negtive_term= CFL/2*np.dot(A_minus,(U[:,x-1]-U[:,x+1]))

            # print(A_minus+A_plus,A)
            #positive propagating
            Linear_term= -1*CFL/2*np.dot(A,(U[:,x+1]-U[:,x-1]))
            A_square=A @ A
            square_term= CFL**2/2*np.dot(A_square,(U[:,x+1]-2*U[:,x]+U[:,x-1]))

            # square_term= CFL**2/2*np.dot(A_square,(phi_x_plus1*(U[:,x+1]-U[:,x])+phi_x_positive*(U[:,x]-U[:,x-1])))


            U_prime[:,x]=positive_term+negtive_term+square_term+U[:,x]
```

```
                U[:,x]=U_prime[:,x]
                rho[x]=U[0,x]
                u[x]=U[1,x]/U[0,x]
                p[x]=(U[2,x]−0.5∗rho[x]∗u[x]∗u[x])∗(gamma−1)

                # print (U[:,x+1]−U[:,x−1],A,np.dot((U[:,x+1]−U[:,x−1]),A),
                    Linear_term)
        return  x_list , rho,u,p



# set up main function
def solve_sod_shock_tube_LF(nx, L, dt, t_end):
    dx = L / (nx − 1)
    # dt = t_end / (nt − 1)
    nt=int((t_end/dt)+1)
    CFL=dt/dx
    x_list = np.linspace (0, L, nx)

    # initialize values
    rho, u, p = initial_conditions (nx, x_list )
    U= flux_U(rho, u, p)
    U_prime=U
    # time loop
    for n in range(nt):
        for x in range(nx−2):#
            # print (" Iteration report nx={},nt={}".format(x,n))
            if x ==0 :
                continue
            # cell flux
            A,A_plus, A_minus = lax_friedrichs_flux_split (rho[x], u[x], p[x])


            #positive propagating
            positive_term = −1∗CFL/2∗np.dot(A_plus,(U[:,x+1]−U[:,x−1]))
            #negtive propagating
            negtive_term= CFL/2∗np.dot(A_minus,(U[:,x−1]−U[:,x+1]))

            # print (A_minus+A_plus,A)

            # Linear_term= −1∗CFL/2∗np.dot(A,(U[:,x+1]−U[:,x−1]))
            A_square=A @ A
            square_term= CFL∗∗2/2∗np.dot(A_square,(U[:,x+1]−2∗U[:,x]+U[:,x−1]))

            U_prime[:,x]=positive_term +negtive_term+square_term+U[:,x]

```

```
                U[:,x]=U_prime[:,x]
                rho[x]=U[0,x]
                u[x]=U[1,x]/U[0,x]
                p[x]=(U[2,x]-0.5*rho[x]*u[x]*u[x])*(gamma-1)

                # print (U[:,x+1]-U[:,x-1],A,np.dot((U[:,x+1]-U[:,x-1]),A),
                    Linear_term)
        return x_list, rho,u,p


# solve Sod shock tube
x_list, rho_LF,u_LF,p_LF = solve_sod_shock_tube_LF(nx=200, L=2.0, dt=1e-5, t_end
    =0.001)
x_list, rho_SW,u_SW,p_SW = solve_sod_shock_tube_SW(nx=200, L=2.0, dt=1e-5,
    t_end=0.001)

##read exact soution
data = np.loadtxt('exact.dat', skiprows=1)

X_POS_PLOT = data[:, 0]
DENSITY = data[:, 1]
VELOCITY_X = data[:, 2]
PRESSURE = data[:, 3]

# plot
plt.figure(figsize=(12, 8))

plt.subplot(3, 1, 1)
plt.plot(X_POS_PLOT[:-2], DENSITY[:-2],linestyle='solid', label='Density_exact',
    color='blue')
plt.plot(x_list[:-2], rho_LF[:-2], linestyle='dashed', label='Density_LF',color='
    orange')
plt.plot(x_list[:-2], rho_SW[:-2], linestyle='dashdot', label='Density_SW',color=
    'green')
plt.xlabel('Position')
plt.ylabel('Density')
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(X_POS_PLOT[:-2], VELOCITY_X[:-2],linestyle='solid', label='
    Velocity_exact',color='blue')
plt.plot(x_list[:-2], u_LF[:-2], label='Velocity_LF', linestyle='dashed',color='
    orange')
plt.plot(x_list[:-2], u_SW[:-2], linestyle='dashdot', label='Velocity_SW', color=
    'green')
plt.xlabel('Position')
plt.ylabel('Velocity')
```

```
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(X_POS_PLOT[:-2], PRESSURE[:-2], label='Pressure_exact',linestyle='solid', color='blue')
plt.plot(x_list[:-2], p_LF[:-2], label='Pressure_LF', linestyle ='dashed', color='orange')
plt.plot(x_list[:-2], p_SW[:-2], linestyle='dashdot', label='Pressure_SW', color='green')
plt.xlabel('Position')
plt.ylabel('Pressure')
plt.legend()

plt.tight_layout()
plt.show()
```