

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:

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

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:

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

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

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:

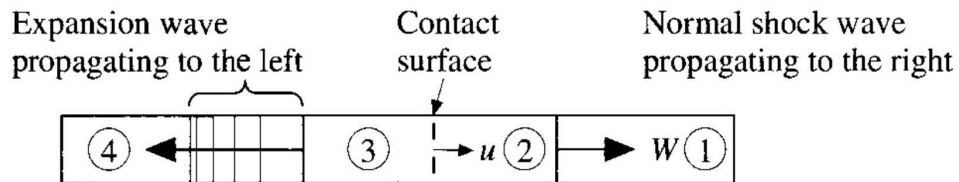


Figure 1: Regions in sod shock tube

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

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

in which

$$c^L = \sqrt{\gamma p^* / \rho^{*L}} \quad (5)$$

In Region 2 and 4

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

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:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} = 0 \quad (7)$$

in which

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

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

$$\frac{\partial \mathbf{U}}{\partial t} + A \frac{\partial \mathbf{U}}{\partial x} = 0 \quad (9)$$

in which

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

it should be noticed that E can be written as

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

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

$$\mathbf{F} = R\Lambda^+ R^{-1}\mathbf{U} + R\Lambda^- R^{-1}\mathbf{U} \quad (12)$$

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

where

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

the right eigenvectors

$$\mathbf{R} = \begin{bmatrix} 1 & 1 & 1 \\ u - a & u & u + a \\ H - ua & u^2/2 & H + ua \end{bmatrix} \quad (15)$$

4.2 Second order upwind scheme

Consider a general scheme in conservative form

$$\frac{\partial \mathbf{U}}{\partial t} = -\mathbf{A} \frac{\mathbf{U}_{i+\frac{1}{2}}^n - \mathbf{U}_{i-\frac{1}{2}}^n}{\Delta x} \quad (16)$$

Consider the second order upwind scheme with limiter

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

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

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

substitute equation 18 into equation 13 we have

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

Which is

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

4.3 Van Leer limiter

Use Van Leer's limiter,

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

4.4 Lax-Wendroff scheme

Discretize equation 9 with Lax-Wendroff scheme

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

4.5 Steger-Warming flux vector splitting

As mentioned before, the Jacobian matrix \mathbf{A} for Euler equations can be diagonalized.

$$\mathbf{A} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^{-1} \quad (23)$$

with the left eigenvectors

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

split the eigenvalues as

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

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

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

4.6 Lax-Friedrichs flux vector splitting

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

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

as well as

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

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.

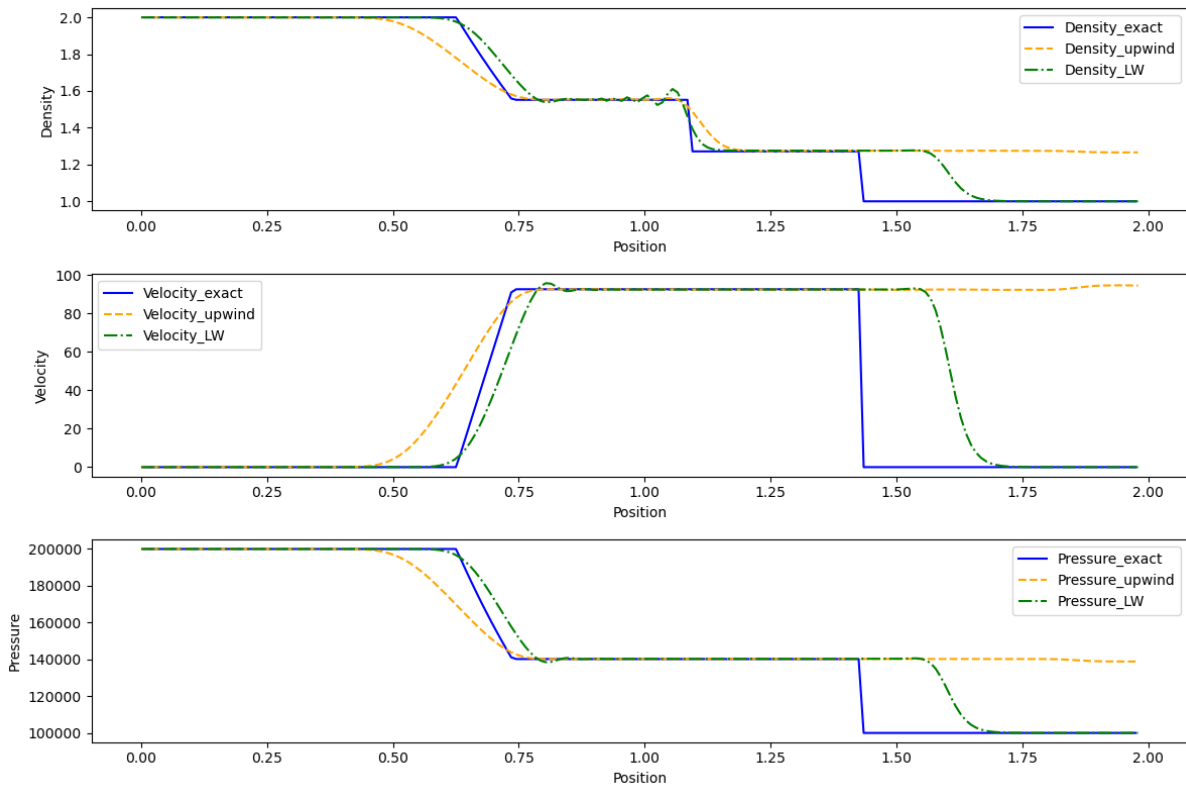


Figure 2: Regions in sod shock tube with different space discretization schemes

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.

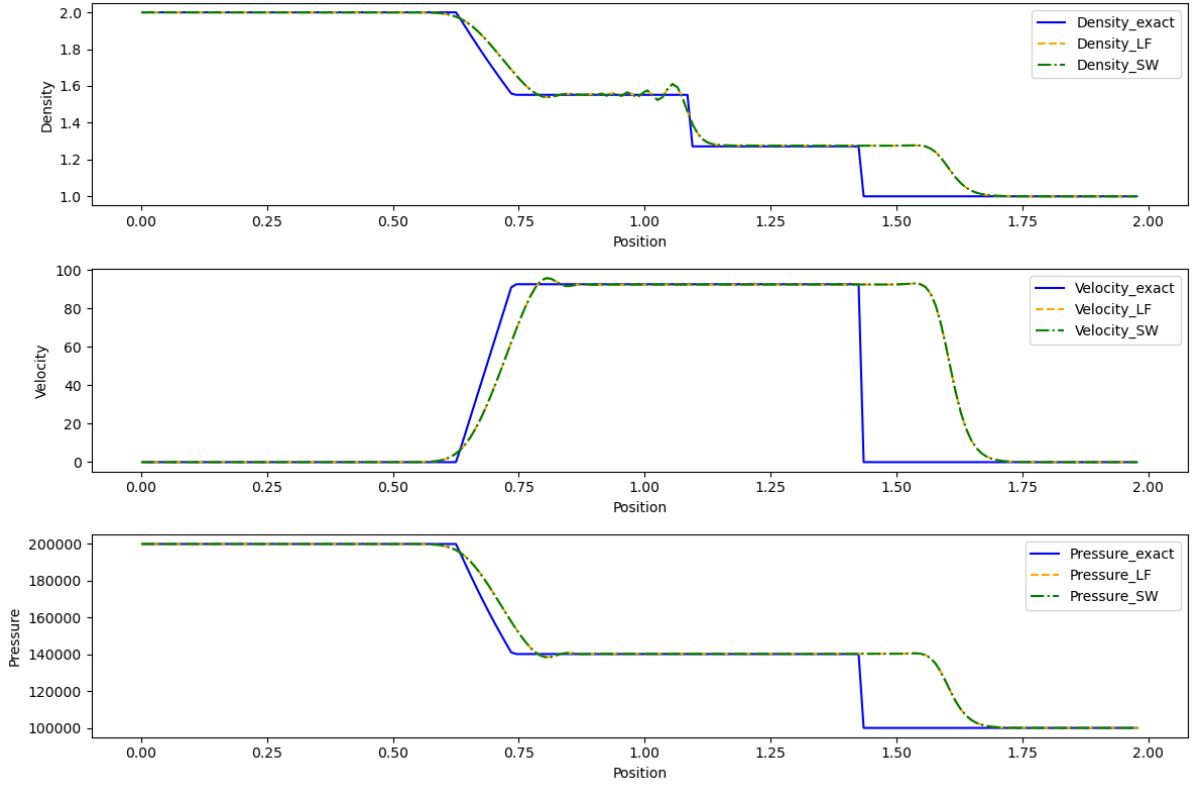


Figure 3: Regions in sod shock tube with different flux vector splitting schemes

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 better 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.

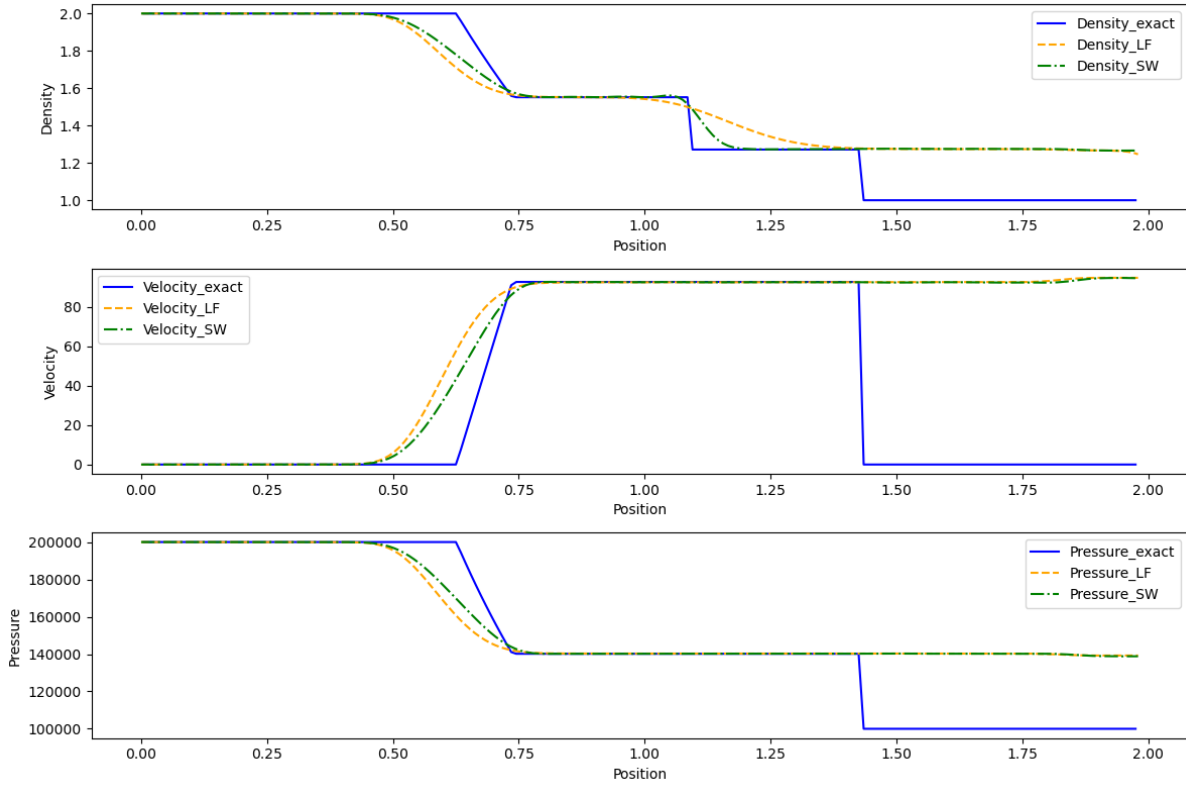


Figure 4: Regions in sod shock tube with different flux vector splitting schemes

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

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import sys
4 from scipy.linalg import eig
5 gamma=1.4
6
7 # Initialize
8 def initial_conditions (nx, x):
9     rho = np.ones(nx)
10    u = np.zeros(nx)
  
```



```
11 p = np.ones(nx)
12
13 # left
14 rho[:nx//2] = 2.0
15 p[:nx//2] = 200000
16
17 # right
18 rho[nx//2:] = 1
19 p[nx//2:] = 100000
20
21 return rho, u, p
22
23 # flux function
24 def flux_F(rho, u, p):
25     F1 = rho * u
26     F2 = rho * u**2 + p
27     F3 = (p / (gamma - 1)) + 0.5 * rho * u**2 + p * u
28     return np.array([F1, F2, F3])
29
30 def flux_U(rho, u, p):
31     U1 = rho
32     U2 = rho * u
33     U3 = (p / (gamma - 1)) + 0.5 * rho * u**2
34     return np.array([U1, U2, U3])
35
36 # set up Steger-Warming flux vector splitting
37 def steger_warming_flux_split(rho, u, p):
38     c = np.sqrt(gamma * p / rho)
39     lam = np.array([[u - c, 0, 0],
40                     [0, u, 0],
41                     [0, 0, u + c]])
42
43     lam_plus = 0.5 * (lam + np.abs(lam))
44     lam_minus = 0.5 * (lam - np.abs(lam))
45     # eigen values and vectors
46
47     # Jacobian Matrix
48     A1 = np.array([0, 1, 0])
49     A2 = np.array([(gamma - 3) / 2 * u**2, (3 - gamma) * u, (gamma - 1)])
50     A3 = np.array([(gamma - 1) * u**3 - gamma * u * (p / ((gamma - 1) * rho) + 0.5 * u**2),
51                     gamma * (p / ((gamma - 1) * rho) + 0.5 * u**2) - (3 * (gamma - 1) / 2) * u**2, gamma * u])
52     A = np.array([A1, A2, A3])
53     # eigenvalues, eigenvectors = eig(A)
54
55     H = c**2 / (gamma - 1) + 0.5 * u**2
56     P1 = np.array([1, 1, 1])
```




```
58 P2=np.array([u-c,u,u+c])
59 P3=np.array([H-u*c,0.5*u**2,H+u*c])
60
61 P= np. array ([ P1,P2,P3])
62
63 A_plus=P @ lam_plus @ np.linalg.inv(P)
64 A_minus=P @ lam_minus @ np.linalg.inv(P)
65 # print (lam_plus,lam_minus)
66
67 return A,A_plus, A_minus
68
69 # set up Van Leer limiter
70 def van_leer_limiter (x,U, direction ):
71     if direction ==-1:
72         numerator = U[:,x+1] - U[:,x]
73         denominator = U[:,x] - U[:,x-1]
74     if direction ==1:
75         numerator = U[:,x-1] - U[:,x]
76         denominator = U[:,x] - U[:,x+1]
77     if np.any(denominator == 0):
78         r=np.array ([1,1,1])
79     else :
80         r=numerator/denominator
81     if np.any(r == 0):
82         r=np.array ([1,1,1])
83     phi = (r + np.abs(r)) / (1 + np.abs(r))
84     return r,phi
85
86 # set up superBee limiter
87 def superBee(x,U, direction ):
88     if direction ==-1:
89         numerator = U[:,x+1] - U[:,x]
90         denominator = U[:,x] - U[:,x-1]
91     if direction ==1:
92         numerator = U[:,x-1] - U[:,x]
93         denominator = U[:,x] - U[:,x+1]
94     if np.any(denominator == 0):
95         r=np.array ([1,1,1])
96     else :
97         r=numerator/denominator
98     if np.any(r == 0):
99         r=np.array ([1,1,1])
100
101     phi1=max(0, min(2 * r [0], 1), min(r [0], 2))
102     phi2=max(0, min(2 * r [1], 1), min(r [1], 2))
103     phi3=max(0, min(2 * r [2], 1), min(r [2], 2))
104     phi=np.array ([ phi1 ,phi2,phi3 ])
```



```
105     return r, phi
106
107
108
109 # set up main function
110 def solve_sod_shock_tube_upwind(nx, L, dt, t_end):
111     dx = L / (nx - 1)
112     # dt = t_end / (nt - 1)
113     nt=int((t_end/dt)+1)
114     CFL=dt/dx
115     x_list = np.linspace(0, L, nx)
116
117     # initialize values
118     rho, u, p = initial_conditions(nx, x_list)
119     U= flux_U(rho, u, p)
120     U_prime=U
121     # time loop
122     for n in range(nt):
123         for x in range(nx-2):#
124             # print(" Iteration report nx={},nt={} ".format(x,n))
125             if x ==0 :
126                 continue
127             # cell flux
128             A,A_plus, A_minus =
129             steger_warming_flux_split(rho[x], u[x], p[x])
130
131             r_x_positive, phi_x_positive= van_leer_limiter(x,U,1)
132             r_x_negative, phi_x_negative= van_leer_limiter(x,U,-1)
133             r_x_minus1, phi_x_minus1=van_leer_limiter(x-1,U,-1)
134             r_x_plus1, phi_x_plus1= van_leer_limiter(x+1,U,1)
135
136             #positive eigenvalue
137             positive_term=-1*CFL*((np.array([1,1,1])+0.5*phi_x_positive-0.5*
138                 phi_x_minus1/r_x_minus1)
139
140             *np.dot(A_plus,(U[:,x]-U[:,x-1])))
141             #negative eigenvalue
142             negative_term=CFL*((1+0.5*phi_x_negative-0.5*phi_x_plus1/r_x_plus1))*
143
144             np.dot(A_minus,(U[:,x]-U[:,x+1]))
145
146             # print(phi_x_minus1,r_x_minus1)
147
148             U_prime[:,x]=positive_term+negative_term+U[:,x]
149
150             U[:,x]=U_prime[:,x]
151             rho[x]=U[0,x]
```



```
151         u[x]=U[1,x]/U[0,x]
152         p[x]=(U[2,x]-0.5*rho[x]*u[x]*u[x])*(gamma-1)
153
154         # print (rho[x], u[x], p[x])
155     return x_list , rho,u,p
156
157 # set up main function
158 def solve_sod_shock_tube_SW(nx, L, dt, t_end):
159     dx = L / (nx - 1)
160     # dt = t_end / (nt - 1)
161     nt=int((t_end/dt)+1)
162     CFL=dt/dx
163     x_list = np.linspace (0, L, nx)
164
165     # initialize values
166     rho, u, p = initial_conditions (nx, x_list )
167     U= flux_U(rho, u, p)
168     U_prime=U
169     # time loop
170     for n in range(nt):
171         for x in range(nx-2):#
172             # print (" Iteration report nx={},nt={} ".format(x,n))
173             if x ==0 :
174                 continue
175             # cell flux
176             A,A_plus, A_minus = steger_warming_flux_split(rho[x], u[x], p[x])
177
178
179             #positive propagating
180             positive_term = -1*CFL/2*np.dot(A_plus,(U[:,x+1]-U[:,x-1]))
181             #negative propagating
182             negative_term= CFL/2*np.dot(A_minus,(U[:,x-1]-U[:,x+1]))
183
184             # print (A_minus+A_plus,A)
185             #positive propagating
186             Linear_term= -1*CFL/2*np.dot(A,(U[:,x+1]-U[:,x-1]))
187             A_square=A @ A
188             square_term= CFL**2/2*np.dot(A_square,(U[:,x+1]-2*U[:,x]+U[:,x-1]))
189
190             # square_term= CFL**2/2*np.dot(A_square,(phi_x_plus1*(U[:,x+1]-U[:,x])
191                 +phi_x_positive*(U[:,x]-U[:,x-1])))
192
193             U_prime[:,x]=positive_term+negative_term+square_term+U[:,x]
194
195             U[:,x]=U_prime[:,x]
196             rho[x]=U[0,x]
```



```
197     u[x]=U[1,x]/U[0,x]
198     p[x]=(U[2,x]-0.5*rho[x]*u[x]*u[x])*(gamma-1)
199
200     # print (U[:,x+1]-U[:,x-1],A,np.dot((U[:,x+1]-U[:,x-1]),A),Linear_term)
201     return x_list , rho,u,p
202
203
204 # solve Sod shock tube
205 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)
206 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)
207
208 ##read exact solution
209 data = np.loadtxt( 'exact.dat' , skiprows=1)
210
211 X_POS_PLOT = data[:, 0]
212 DENSITY = data[:, 1]
213 VELOCITY_X = data[:, 2]
214 PRESSURE = data[:, 3]
215
216 # plot
217 plt . figure ( figsize =(12, 8))
218
219 plt . subplot (3, 1, 1)
220 plt . plot (X_POS_PLOT[:-2], DENSITY[:-2],linestyle='solid', label='Density_exact' ,
    color='blue')
221 plt . plot ( x_list[:-2], rho_upwind[:-2], linestyle='dashed', label='Density_upwind',
    color='orange')
222 plt . plot ( x_list[:-2], rho_SW[:-2], linestyle='dashdot', label='Density_LW', color='
    green')
223 plt . xlabel ( 'Position')
224 plt . ylabel ( 'Density')
225 plt . legend()
226
227 plt . subplot (3, 1, 2)
228 plt . plot (X_POS_PLOT[:-2], VELOCITY_X[:-2],linestyle='solid', label='Velocity_exact' ,
    color='blue')
229 plt . plot ( x_list[:-2], u_upwind[:-2], label='Velocity_upwind' , linestyle='dashed',
    color='orange')
230 plt . plot ( x_list[:-2], u_SW[:-2], linestyle='dashdot', label='Velocity_LW', color='
    green')
231 plt . xlabel ( 'Position')
232 plt . ylabel ( 'Velocity')
233 plt . legend()
234
235 plt . subplot (3, 1, 3)
```



```
236 plt.plot(X_POS_PLOT[:-2], PRESSURE[:-2], label='Pressure_exact', linestyle='solid',  
           color='blue')  
237 plt.plot(x_list[:-2], p_upwind[:-2], label='Pressure_upwind', linestyle='dashed',  
           color='orange')  
238 plt.plot(x_list[:-2], p_SW[:-2], linestyle='dashdot', label='Pressure_LW', color='  
           green')  
239 plt.xlabel('Position')  
240 plt.ylabel('Pressure')  
241 plt.legend()  
242  
243 plt.tight_layout()  
244 plt.show()
```

6.2 Flux vector splitting

```
1 import numpy as np  
2 import matplotlib.pyplot as plt  
3 import sys  
4 from scipy.linalg import eig  
5 gamma=1.4  
6  
7 # Initialize  
8 def initial_conditions (nx, x):  
9     rho = np.ones(nx)  
10    u = np.zeros(nx)  
11    p = np.ones(nx)  
12  
13    # left  
14    rho[:nx//2] = 2.0  
15    p[:nx//2] = 200000  
16  
17    # right  
18    rho[nx//2:] = 1  
19    p[nx//2:] = 100000  
20  
21    return rho, u, p  
22  
23 # flux function  
24 def flux_F(rho, u, p):  
25     F1 = rho * u  
26     F2 = rho * u**2 + p  
27     F3 = (p / (gamma - 1)) + 0.5 * rho * u**2 + p * u  
28     return np.array([F1, F2, F3])  
29  
30 def flux_U(rho, u, p):  
31     U1 = rho  
32     U2 = rho * u
```



```
33     U3 = (p / (gamma - 1)) + 0.5 * rho * u**2
34     return np.array ([U1, U2, U3])
35
36     # set up Steger-Warming flux vector splitting
37     def steger_warming_flux_split (rho, u, p):
38         c = np.sqrt (gamma * p / rho)
39         lam = np.array ([[ u - c, 0,0],
40                         [0, u, 0],
41                         [0, 0, u + c]]
42                         )
43         lam_plus=0.5*(lam+np.abs(lam))
44         lam_minus=0.5*(lam-np.abs(lam))
45         # eigen values and vectors
46
47         # Jacobian Metrix
48         A1=np.array ([0, 1, 0])
49         A2=np.array ((( gamma-3)/2)*u**2, (3-gamma)*u, (gamma-1)])
50         A3=np.array ([ (gamma-1)*u**3-gamma*u*(p/((gamma-1)*rho)+0.5*u**2),
51                     gamma*(p/((gamma-1)*rho)+0.5*u**2)-(3*(gamma-1)/2)*u**2, gamma*u])
52         A= np.array ([ A1,A2,A3])
53         # eigenvalues, eigenvectors = eig(A)
54
55         H=c**2/(gamma-1)+0.5*u**2
56         P1=np.array ([1,1,1])
57         P2=np.array ([ u-c,u,u+c])
58         P3=np.array ([ H-u*c,0.5*u**2,H+u*c])
59
60         P= np.array ([ P1,P2,P3])
61
62         A_plus=P @ lam_plus @ np.linalg.inv(P)
63         A_minus=P @ lam_minus @ np.linalg.inv(P)
64         # print (lam_plus,lam_minus)
65
66         return A,A_plus, A_minus
67
68     # set up Lax-Friedrichs flux vector splitting
69     def lax_friedrichs_flux_split (rho, u, p):
70         c = np.sqrt (gamma * p / rho)
71         lam = np.array ([[ u - c, 0,0],
72                         [0, u, 0],
73                         [0, 0, u + c]]
74                         )
75         # eigen values and vectors
76
77         # Jacobian Metrix
78         A1=np.array ([0, 1, 0])
```



```
79     A2=np.array([((gamma-3)/2)*u**2, (3-gamma)*u, (gamma-1)])
80     A3=np.array([(gamma-1)*u**3-gamma*u*(p/((gamma-1)*rho)+0.5*u**2),
81                 gamma*(p/((gamma-1)*rho)+0.5*u**2)-(3*(gamma-1)/2)*u**2, gamma*u])
82     A= np.array([A1,A2,A3])
83     # eigenvalues, eigenvectors = eig(A)
84
85     lam_max=max(u-c,u,u+c)
86     A_plus=0.5*(A+lam_max*np.eye(3))
87     A_minus=0.5*(A-lam_max*np.eye(3))
88
89     return A,A_plus, A_minus
90
91 # set up Van Leer limiter
92 def van_leer_limiter(x,U, direction ):
93     if direction ==-1:
94         numerator = U[:,x+1] - U[:,x]
95         denominator = U[:,x] - U[:,x-1]
96     if direction ==1:
97         numerator = U[:,x-1] - U[:,x]
98         denominator = U[:,x] - U[:,x+1]
99     if np.any(denominator == 0):
100         r=np.array ([1,1,1])
101     else :
102         r=numerator/denominator
103     if np.any(r == 0):
104         r=np.array ([1,1,1])
105     phi = (r + np.abs(r)) / (1 + np.abs(r))
106     return r,phi
107
108 # set up superBee limiter
109 def superBee(x,U, direction ):
110     if direction ==-1:
111         numerator = U[:,x+1] - U[:,x]
112         denominator = U[:,x] - U[:,x-1]
113     if direction ==1:
114         numerator = U[:,x-1] - U[:,x]
115         denominator = U[:,x] - U[:,x+1]
116     if np.any(denominator == 0):
117         r=np.array ([1,1,1])
118     else :
119         r=numerator/denominator
120     if np.any(r == 0):
121         r=np.array ([1,1,1])
122
123     phi1=max(0, min(2 * r [0], 1), min(r [0], 2))
124     phi2=max(0, min(2 * r [1], 1), min(r [1], 2))
```



```
125     phi3=max(0, min(2 * r [2], 1), min(r [2], 2))
126     phi=np.array ([ phi1 ,phi2 ,phi3 ])
127     return r,phi
128
129
130     # set up main function
131     def solve_sod_shock_tube_SW(nx, L, dt, t_end):
132         dx = L / (nx - 1)
133         # dt = t_end / (nt - 1)
134         nt=int ((t_end/dt)+1)
135         CFL=dt/dx
136         x_list = np.linspace (0, L, nx)
137
138         # initialize values
139         rho, u, p = initial_conditions (nx, x_list )
140         U= flux_U(rho, u, p)
141         U_prime=U
142         # time loop
143         for n in range(nt):
144             for x in range(nx-2):#
145                 # print(" Iteration report nx={},nt={} ".format(x,n))
146                 if x ==0 :
147                     continue
148                 # cell flux
149                 A,A_plus, A_minus = steger_warming_flux_split (rho[x], u[x], p[x])
150
151                 r_x_positive , phi_x_positive=superBee(x,U,1)
152                 r_x_negtive ,phi_x_negtive=superBee(x,U,-1)
153                 r_x_minus1,phi_x_minus1=superBee(x-1,U,-1)
154                 r_x_plus1,phi_x_plus1=superBee(x+1,U,1)
155
156                 #positive propagating
157                 positive_term = -1*CFL/2*np.dot(A_plus,(U[:,x+1]-U[:,x-1]))
158                 #negative propagating
159                 negative_term= CFL/2*np.dot(A_minus,(U[:,x-1]-U[:,x+1]))
160
161                 # print (A_minus+A_plus,A)
162                 #positive propagating
163                 Linear_term= -1*CFL/2*np.dot(A,(U[:,x+1]-U[:,x-1]))
164                 A_square=A @ A
165                 square_term= CFL**2/2*np.dot(A_square,(U[:,x+1]-2*U[:,x]+U[:,x-1]))
166
167                 # square_term= CFL**2/2*np.dot(A_square,(phi_x_plus1*(U[:,x+1]-U
168                     [:,x])+phi_x_positive*(U[:,x]-U[:,x-1])))
169
170                 U_prime[:,x]= positive_term+negative_term+square_term+U[:,x]
```




```
171
172     U[:,x]=U_prime[:,x]
173     rho[x]=U[0,x]
174     u[x]=U[1,x]/U[0,x]
175     p[x]=(U[2,x]-0.5*rho[x]*u[x]*u[x])*(gamma-1)
176
177     # print (U[:,x+1]-U[:,x-1],A,np.dot((U[:,x+1]-U[:,x-1]),A),
178           Linear_term)
179
180     return x_list , rho,u,p
181
182 # set up main function
183 def solve_sod_shock_tube_LF(nx, L, dt, t_end):
184     dx = L / (nx - 1)
185     # dt = t_end / (nt - 1)
186     nt=int((t_end/dt)+1)
187     CFL=dt/dx
188     x_list = np.linspace (0, L, nx)
189
190     # initialize values
191     rho, u, p = initial_conditions (nx, x_list )
192     U= flux_U(rho, u, p)
193     U_prime=U
194     # time loop
195     for n in range(nt):
196         for x in range(nx-2):#
197             # print (" Iteration  report nx={},nt={} ".format(x,n))
198             if x ==0 :
199                 continue
200             # cell flux
201             A,A_plus, A_minus = lax_friedrichs_flux_split (rho[x], u[x], p[x])
202
203
204             #positive propagating
205             positive_term = -1*CFL/2*np.dot(A_plus,(U[:,x+1]-U[:,x-1]))
206             #negative propagating
207             negative_term= CFL/2*np.dot(A_minus,(U[:,x-1]-U[:,x+1]))
208
209             # print (A_minus+A_plus,A)
210
211             # Linear_term= -1*CFL/2*np.dot(A,(U[:,x+1]-U[:,x-1]))
212             A_square=A @ A
213             square_term= CFL**2/2*np.dot(A_square,(U[:,x+1]-2*U[:,x]+U[:,x-1]))
214
215             U_prime[:,x]= positive_term+negative_term+square_term+U[:,x]
216
```



```
217         U[:,x]=U_prime[:,x]
218         rho[x]=U[0,x]
219         u[x]=U[1,x]/U[0,x]
220         p[x]=(U[2,x]-0.5*rho[x]*u[x]*u[x])*(gamma-1)
221
222         # print (U[:,x+1]-U[:,x-1],A,np.dot((U[:,x+1]-U[:,x-1]),A),
223             Linear_term)
224     return x_list , rho,u,p
225
226 # solve Sod shock tube
227 x_list , rho_LF,u_LF,p_LF = solve_sod_shock_tube_LF(nx=200, L=2.0, dt=1e-5, t_end
228     =0.001)
229 x_list , rho_SW,u_SW,p_SW = solve_sod_shock_tube_SW(nx=200, L=2.0, dt=1e-5,
230     t_end=0.001)
231
232 ##read exact soution
233 data = np. loadtxt ( 'exact.dat' , skiprows=1)
234
235 X_POS_PLOT = data[:, 0]
236 DENSITY = data[:, 1]
237 VELOCITY_X = data[:, 2]
238 PRESSURE = data[:, 3]
239
240 # plot
241 plt . figure ( figsize =(12, 8))
242
243 plt . subplot (3, 1, 1)
244 plt . plot (X_POS_PLOT[:-2], DENSITY[:-2],linestyle='solid', label='Density_exact ',
245     color='blue ')
246 plt . plot ( x_list[:-2], rho_LF[:-2], linestyle = 'dashed', label='Density_LF',color='
247     orange')
248 plt . plot ( x_list[:-2], rho_SW[:-2], linestyle = 'dashdot', label='Density_SW',color=
249     'green')
250 plt . xlabel ( 'Position ')
251 plt . ylabel ( 'Density ')
252 plt . legend()
253
254 plt . subplot (3, 1, 2)
255 plt . plot (X_POS_PLOT[:-2], VELOCITY_X[:-2],linestyle='solid', label='
256     Velocity_exact ', color='blue ')
257 plt . plot ( x_list[:-2], u_LF[:-2], label='Velocity_LF', linestyle = 'dashed', color='
258     orange')
259 plt . plot ( x_list[:-2], u_SW[:-2], linestyle = 'dashdot', label='Velocity_SW', color=
260     'green')
261 plt . xlabel ( 'Position ')
262 plt . ylabel ( 'Velocity ')
```



```
255 plt.legend()
256
257 plt.subplot(3, 1, 3)
258 plt.plot(X_POS_PLOT[:-2], PRESSURE[:-2], label='Pressure_exact', linestyle='solid',
259          color='blue')
259 plt.plot(x_list[:-2], p_LF[:-2], label='Pressure_LF', linestyle='dashed', color='orange')
260 plt.plot(x_list[:-2], p_SW[:-2], linestyle='dashdot', label='Pressure_SW', color='green')
261 plt.xlabel('Position')
262 plt.ylabel('Pressure')
263 plt.legend()
264
265 plt.tight_layout()
266 plt.show()
```