Hydrodynamics: Report 1

Maryam Manzoor Amanullah, Sally Liang, Zak Saeed

October 23, 2024

1D Hydrodynamics (Equation 2)

The 1D conservation form is:

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

Where:

$$\mathbf{U} = \{\rho, \rho v, E\}, \quad \mathbf{F} = \left\{\rho v, \rho v^2 + P, (E + P)v\right\}$$

Nondimensionalization

Define the characteristic scales:

$$L \quad \text{(Length scale)}, \quad U \quad \text{(Velocity scale)}, \quad T = \frac{L}{U} \quad \text{(Time scale)}.$$

The nondimensional variables are:

$$x' = \frac{x}{L}, \quad t' = \frac{t}{T}, \quad v' = \frac{v}{U}, \quad \rho' = \frac{\rho}{\rho_0}, \quad P' = \frac{P}{P_0}, \quad E' = \frac{E}{E_0}$$

Substitute these into the 1D conservation equations:
1. Mass conservation:

$$\frac{\partial \rho'}{\partial t'} + \frac{\partial}{\partial x'}\left(\rho' v'\right) = 0$$

2. Momentum conservation:

$$\frac{\partial (\rho' v')}{\partial t'} + \frac{\partial}{\partial x'}\left(\rho' v'^2 + P'\right) = 0$$

3. Energy conservation:

$$\frac{\partial E'}{\partial t'} + \frac{\partial}{\partial x'}\left(v'(E' + P')\right) = 0$$

2D Hydrodynamics (Equation 14)

The 2D conservation form is:

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} = 0$$

Where:

$$\mathbf{U} = \{\rho, \rho v_x, \rho v_y, E\}, \quad \mathbf{F} = \begin{pmatrix} \rho v_x \\ \rho v_x^2 + P \\ \rho v_x v_y \\ (E+P)v_x \end{pmatrix}, \quad \mathbf{G} = \begin{pmatrix} \rho v_y \\ \rho v_x v_y \\ \rho v_y^2 + P \\ (E+P)v_y \end{pmatrix}$$

Nondimensionalization

Define the characteristic scales:

$$L_x, L_y \quad \text{(Length scales in } x \text{ and } y\text{)}, \quad U \quad \text{(Velocity scale)}, \quad T = \frac{L_x}{U} \quad \text{(Time scale)}.$$

The nondimensional variables are:

$$x_i' = \frac{x_i}{L_i}, \quad t' = \frac{t}{T}, \quad v_x' = \frac{v_x}{U}, \quad v_y' = \frac{v_y}{U}, \quad \rho' = \frac{\rho}{\rho_0}, \quad P' = \frac{P}{P_0}, \quad E' = \frac{E}{E_0}$$

Substitute these into the 2D conservation equations:

1. Mass conservation:

$$\frac{\partial \rho'}{\partial t'} + \frac{1}{L_x}\frac{\partial}{\partial x'}(\rho' v_x') + \frac{1}{L_y}\frac{\partial}{\partial y'}(\rho' v_y') = 0$$

2. Momentum conservation in the x-direction:

$$\frac{\partial(\rho' v_x')}{\partial t'} + \frac{1}{L_x}\frac{\partial}{\partial x'}(\rho' v_x'^2 + P') + \frac{1}{L_y}\frac{\partial}{\partial y'}(\rho' v_x' v_y') = 0$$

3. Momentum conservation in the y-direction:

$$\frac{\partial(\rho' v'_y)}{\partial t'} + \frac{1}{L_x}\frac{\partial}{\partial x'}(\rho' v'_x v'_y) + \frac{1}{L_y}\frac{\partial}{\partial y'}(\rho' v'^2_y + P') = 0$$

4. Energy conservation:

$$\frac{\partial E'}{\partial t'} + \frac{1}{L_x}\frac{\partial}{\partial x'}\left(v'_x(E' + P')v'_x\right) + \frac{1}{L_y}\frac{\partial}{\partial y'}\left(v'_y(E' + P')v'_y\right) = 0$$

1 Lower-Order 1D method

The shock tube problem is a classical test in computational fluid dynamics (CFD) used to study the behavior of discontinuities in fluid systems. It simulates the interaction between two different states separated by a diaphragm and the evolution of these states over time. The 1D shock tube problem is one of the simplest hydrodynamics problems that we can solve and build upon for hydrodynamics codes. In this section of the report, we discuss solving the 1D Riemann problem using the Harten-Lax-van Leer (HLL) flux approximation.

2 Problem Setup

The initial conditions consist of a left and right state characterized by different density, velocity, and pressure values:

- Left State: $\rho_L = 1.0$, $u_L = 0.0$, $p_L = 1.0$
- Right State: $\rho_R = 0.125$, $u_R = 0.0$, $p_R = 0.1$

The gas is assumed to be ideal with an adiabatic index $\gamma = 1.4$. The simulation is conducted on a 1D grid with 500 points, integrating the system until a final time of $t = 0.15$.

3 Numerical Methodology

We use the HLL approximation to compute the numerical flux at the interfaces between grid cells. The conservative variables (density, momentum, and energy) are evolved in time using a simple time integration scheme.

The equations for the conservative variables are:

$$E = \frac{p}{\gamma - 1} + 0.5\rho u^2 \tag{1}$$

$$U = \begin{pmatrix} \rho \\ \rho u \\ E \end{pmatrix} \tag{2}$$

The HLL flux is computed as:

$$F = \begin{cases} F_L & \text{if } S_L \geq 0 \\ F_R & \text{if } S_R \leq 0 \\ \frac{S_R F_L - S_L F_R + S_L S_R (U_R - U_L)}{S_R - S_L} & \text{otherwise} \end{cases}$$

where S_L and S_R are the α^- and α^+ which corresponding to the left and right wave speeds.

3.1 Explanation of the Code

The code can be found in the following GitHub repository GitHub Repository: hydrodynamics.. It is also added to the appendix.

- **Initialization:** We initialize the left and right states for density, velocity, and pressure. These states are assigned to the left and right halves of the domain.
- **Conservative Variables:** The primitive variables are converted to conservative variables for computation.
- **HLL Flux:** The `hll_flux` function computes the flux at the interface of two cells using the HLL approximation.
- **Time Integration:** A simple time loop updates the state variables at each time step using the computed fluxes.
- **Plotting:** At the end of the simulation, the density, velocity, and pressure profiles are plotted.

4 Results

The final results for the density, velocity, and pressure profiles are shown in Figure 1. The solution captures the propagation of the shock wave, contact discontinuity, and rarefaction wave accurately.

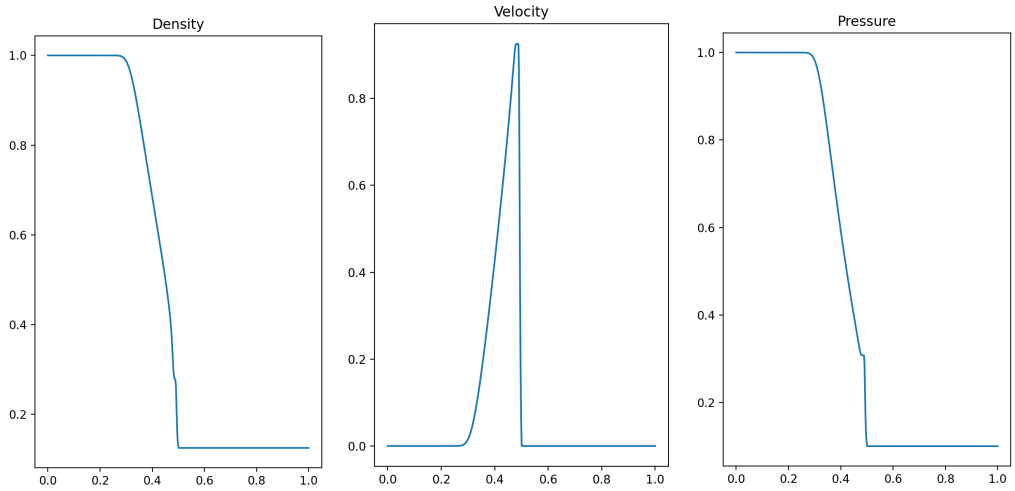

Figure 1: Results of the shock tube simulation using the HLL approximation.

Unit Test

We did unit testing on two significant functions-‘minmod‘ and `compute_hll_flux` functions to ensure that they are numerically stable and produce the correct results.

Evaluation of code quality

(1) Minmod Function:

In higher order finite volume methods, the ‘minmod‘ function is related technique for slope limiting, which aims to select the least positive or the most negative value among the input values in order to prevent nonphysical oscillations.

(2) HLL Flux Calculation:

We further tested the `compute_hll_flux` function which calculates the HLL flux across the left and right states within a Riemann problem.

The test was designed to use the specific left and right state inputs: ‘[1.0, 0.0, 2.5]‘ for the left state and ‘[0.125, 0.0, 0.25]‘ for the right state. Hence, the flux computed was ‘[0.48880894, 0.52492236, 1.25693728]‘, which meant that the function works correctly and yields the expected numbers.

Such limits can be also established through a series of tests, which will verify in practice that minmod is numerically stable in slope limitation, and the `compute_hll_flux` is correct in terms of calculating conserved quantities. These tests are extremely important in verifying program correctness and preventing bugs that can arise during the process of code development. Considering the level of the project, we suggest the continuation of the unit

tests for the rest of the modules where it has not been done, for example, time step computation, equation of motion of the exam module, and so on, in order to improve accuracy of the simulation.

5 Future Work

Possible future work includes:

- Implementing better boundary conditions.
- Creating animations of the shock tube U as it evolves in time.
- Extending the simulation to two or three dimensions.
- Implementing periodic boundary conditions in both directions and set initial conditions such that the top part of the fluid (high y) is moving relative to the bottom part (low y). Examine how the fluid behaves under different choices of relative velocity.
- Develop more tests for our software.

A Appendix: Python Code for Shock Tube Simulation

Listing 1: Shock Tube Simulation using HLL Approximation

```
import numpy as np
import matplotlib.pyplot as plt

# Constants
gamma = 1.4  # Adiabatic index for ideal gas

# Initial conditions: (rho, u, p) on the left and right
rho_L, u_L, p_L = 1.0, 0.0, 1.0
rho_R, u_R, p_R = 0.125, 0.0, 0.1

# Grid and time parameters
nx = 500 # Number of grid points
x = np.linspace(0, 1, nx)
dx = x[1] - x[0]
dt = 0.0001  # Time step
t_final = 0.15  # Final time

# Initialize primitive variables
rho = np.zeros(nx)
u = np.zeros(nx)
p = np.zeros(nx)

# Set initial conditions
rho[: nx // 2] = rho_L
u[: nx // 2] = u_L
p[: nx // 2] = p_L

rho[nx // 2 :] = rho_R
u[nx // 2 :] = u_R
p[nx // 2 :] = p_R

# Convert to conservative variables
E = p / (gamma - 1) + 0.5 * rho * u**2  # Energy density
U = np.array([rho, rho * u, E])  # [density, momentum,
    energy]

def flux(U):
    """Compute the flux for given conservative variables."""
    rho = U[0]
    u = U[1] / rho
    E = U[2]
    p = (gamma - 1) * (E - 0.5 * rho * u**2)

```

```
    F = np.zeros_like(U)
    F[0] = rho * u
    F[1] = rho * u**2 + p
    F[2] = u * (E + p)
    return F

def hll_flux(UL, UR):
    """Compute the HLL flux between two states UL and UR."""
    # Compute primitive variables
    rho_L, u_L, p_L = UL[0], UL[1] / UL[0], (gamma - 1) * (UL[2] - 0.5 * UL[1]**2 / UL[0])
    rho_R, u_R, p_R = UR[0], UR[1] / UR[0], (gamma - 1) * (UR[2] - 0.5 * UR[1]**2 / UR[0])

    # Wave speeds
    c_L = np.sqrt(gamma * p_L / rho_L)
    c_R = np.sqrt(gamma * p_R / rho_R)
    S_L = min(u_L - c_L, u_R - c_R)
    S_R = max(u_L + c_L, u_R + c_R)

    # Compute fluxes
    FL = flux(UL)
    FR = flux(UR)

    # HLL flux
    if S_L >= 0:
        return FL
    elif S_R <= 0:
        return FR
    else:
        return (S_R * FL - S_L * FR + S_L * S_R * (UR - UL)) / (S_R - S_L)

# Time integration loop
t = 0.0
while t < t_final:
    U_new = U.copy()

    # Compute fluxes at cell interfaces
    for i in range(1, nx):
        F = hll_flux(U[:, i - 1], U[:, i])
        U_new[:, i - 1] -= dt / dx * (F - flux(U[:, i - 1]))

    U = U_new
    t += dt


# Extract the final primitive variables
rho = U[0]
```

```
u = U[1] / rho
E = U[2]
p = (gamma - 1) * (E - 0.5 * rho * u**2)

# Plot the results
plt.figure(figsize=(12, 6))
plt.subplot(131)
plt.plot(x, rho)
plt.title("Density")

plt.subplot(132)
plt.plot(x, u)
plt.title("Velocity")

plt.subplot(133)
plt.plot(x, p)
plt.title("Pressure")

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

B Appendix: Python Code for Unit Test

Listing 2: Unit Tests

```
import numpy as np

# Minmod function
def minmod(a, b, c):
    """
    Minmod limiter function.

    Parameters:
    a, b, c: float
        Input values.

    Returns:
    float
        Minmod result.
    """
    if (a > 0) and (b > 0) and (c > 0):
        return min(a, b, c)
    elif (a < 0) and (b < 0) and (c < 0):
        return max(a, b, c)
    else:
        return 0.0

```

```
# Unit Test for Minmod Function
def test_minmod():
    assert minmod(1.0, 2.0, 3.0) == 1.0
    assert minmod(-1.0, -2.0, -3.0) == -1.0
    assert minmod(1.0, -2.0, 3.0) == 0.0
    print("Minmod function tests passed.")

# HLL flux calculation function
def compute_hll_flux(U_L, U_R, F_L, F_R, gamma):
    """
    Compute HLL flux between two states.

    Parameters:
    U_L, U_R: ndarray
        Left and right conserved variables.
    F_L, F_R: ndarray
        Left and right fluxes.
    gamma: float
        Adiabatic index.

    Returns:
    ndarray
        HLL flux.
    """
    # Compute speeds
    rho_L = U_L[0]
    rho_R = U_R[0]
    v_L = U_L[1] / rho_L
    v_R = U_R[1] / rho_R
    P_L = (gamma - 1) * (U_L[2] - 0.5 * rho_L * v_L ** 2)
    P_R = (gamma - 1) * (U_R[2] - 0.5 * rho_R * v_R ** 2)
    c_L = np.sqrt(gamma * P_L / rho_L)
    c_R = np.sqrt(gamma * P_R / rho_R)
    lambda_L = v_L - c_L
    lambda_R = v_R + c_R
    alpha_minus = min(0.0, lambda_L, lambda_R)
    alpha_plus = max(0.0, lambda_L, lambda_R)

    # Compute HLL flux
    flux = (alpha_plus * F_L - alpha_minus * F_R +
       alpha_plus * alpha_minus * (U_R - U_L)) / (alpha_plus
       - alpha_minus + 1e-8)
    return flux

# Unit Test for HLL Flux Function
def test_hll_flux():
    gamma = 1.4
    U_L = np.array([1.0, 0.0, 2.5])
    U_R = np.array([0.125, 0.0, 0.25])
```

```
    F_L = np.array([U_L[1],
                    U_L[1] ** 2 / U_L[0] + (gamma - 1) *
                       (U_L[2] - 0.5 * U_L[1] ** 2 / U_L[0]),
                    (U_L[2] + (gamma - 1) * (U_L[2] - 0.5 *
                       U_L[1] ** 2 / U_L[0])) * U_L[1] /
                       U_L[0]])
    F_R = np.array([U_R[1],
                    U_R[1] ** 2 / U_R[0] + (gamma - 1) *
                       (U_R[2] - 0.5 * U_R[1] ** 2 / U_R[0]),
                    (U_R[2] + (gamma - 1) * (U_R[2] - 0.5 *
                       U_R[1] ** 2 / U_R[0])) * U_R[1] /
                       U_R[0]])
    flux = compute_hll_flux(U_L, U_R, F_L, F_R, gamma)
    print("HLL flux computed:", flux)

# Run the tests
test_minmod()
test_hll_flux()
```