Computational Methods for Compressible and Incompressible Fluid Mechanics

Final Project

Shachar Charash
316555713

Table of Contents

Figure List

Abstract

This paper presents numerical simulations of the one-dimensional Euler equations for a calorically perfect gas, conducted using a custom-built code. The code employs the Godunov method, incorporating three distinct Riemann solvers: an exact solver, the HLLC solver, and the Roe-Pike solver. To evaluate the accuracy of these implementations, simulations are performed for two well-established test cases (Tests 1 and 4) from Toro's Riemann problem benchmarks **[1]** with the results subsequently compared against Toro's reference solutions. Furthermore, a comparative analysis is undertaken to assess the relative performance and accuracy characteristics of the three Riemann solvers against each other and an exact solution for the selected test cases. This study aims to provide insights into the strengths and limitations of each solver in the context of these specific problems.

This paper's results confirm the successful validation of the numerical code. They show that while all solvers accurately capture shock waves, they exhibit smearing for contact discontinuities. Notably, the Roe-Pike solver struggled with rarefaction waves. In terms of efficiency, the HLLC solver proved most effective, striking an optimal balance between accuracy and computational cost.

Introduction

The Euler equations describe the conservation of mass, momentum, and energy in a compressible fluid. In one dimension, they form a system of nonlinear hyperbolic partial differential equations (PDEs), which can be expressed in conservative form as such.

$$(1)\ U_t + F(U)_x = 0$$

Where:

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

Here vector U represents conservative parameters of mass momentum and energy and F represents the flux of these parameters through the gas. The parameters of ρ represents density, u velocity and p is pressure. The value of E (energy) is calculated using the equation of state for calorically perfect gas

$$(3)\ E = \rho\left(\frac{1}{2}u^2 + \frac{p}{(\gamma-1)\rho}\right)$$

Where the parameter γ is the Ratio of specific heats

The hyperbolic nature of the Euler equations means that information propagates along characteristic lines at finite speeds, that is to say, any changes in the flow, such as pressure or velocity, move through

the domain gradually as waves. The change can form sharp features like shock waves or smooth transitions like rarefactions.

A central problem in solving the Euler equations numerically is handling discontinuities in the initial conditions. This leads to the Riemann problem, a special case of the initial value problem where the initial state consists of two constant states separated by a discontinuity. The Riemann problem and its initial conditions are expressed generally as follows:

$$(4)\ PDE:\ \ u_t + au_x = 0$$

$$(5)\ IC:\ \ u(x,0) = \begin{cases} u_L : x < x_0 \\ u_R : x > x_0 \end{cases}$$

Here the discontinuity is represented as the point x_0 and the initial conditions are constant and different from the right and left of this point.

When dealing with systems of hyperbolic equations, Equation (4) can be represented in a vector form as:

$$U_t + AU_x = 0$$

To reflect the quasi-linear nature of the Euler equations, we consider the matrix A to be a function of time, space, and velocity: $A \equiv A(t, x, u)$

In this context, comparing this form to the Euler equations (Equation (1)), the matrix A corresponds to the Jacobian matrix.

$$(6)\ A = \frac{\partial F}{\partial U} = \begin{bmatrix} \partial f_1/\partial u_1 & \partial f_1/\partial u_2 & \partial f_1/\partial u_3 \\ \partial f_2/\partial u_1 & \partial f_2/\partial u_2 & \partial f_2/\partial u_3 \\ \partial f_3/\partial u_1 & \partial f_3/\partial u_2 & \partial f_3/\partial u_3 \end{bmatrix}$$

The solution to the Riemann problem involves the evolution of this discontinuity over time and typically results in the formation of three waves: a left-moving wave (shock or rarefaction), a contact discontinuity, and a right-moving wave (shock or rarefaction).

In the test cases considered, the Riemann problem has an exact solution, which is computed at a given end time using the initial conditions and the point of discontinuity in the specific case. However, in more complex scenarios where an analytical solution is not feasible, a numerical approach becomes necessary. To address this, the Godunov method is also implemented.

The Godunov method is a conservative numerical scheme for solving hyperbolic partial differential equations. It uses a finite volume approach, where the spatial domain is divided into a grid of control volumes and time is advanced in discrete steps. The solution is updated at each time step using an update equation that involves calculating fluxes (F) at the interfaces between control volumes. The specific way these fluxes are calculated depends on the chosen Riemann solver. In this work, an exact Riemann solver is implemented, which computes the flux precisely.

Additionally, two approximate Riemann solvers are used: the HLLC solver, which simplifies the solution by estimating the wave front speeds, and the Roe-Pike solver, which linearizes the problem by assuming a constant Jacobian matrix.

These methods offer different trade-offs between accuracy and computational cost. By implementing both the exact solution method and the Godunov-based numerical Riemann solvers, this code enables a direct comparison not only between analytical and numerical solutions but also among the different numerical solvers themselves.

Numerical Methods

As mentioned, we use the Riemann problem to solve the Euler equations, which are a system of quasi-linear hyperbolic equations, as represented in Equation (1) and with the initial conditions:

$$(7)\ IC:\ \ U(x,0) = \begin{cases} U_L: x<x_0 \\ U_R: x>x_0 \end{cases}$$

To solve the equations more conveniently, we use the vector W, which contains the primitive variables (density, velocity and pressure), instead of U, which represents the conserved variables

$$(8)\ W = \begin{bmatrix} \rho \\ u \\ p \end{bmatrix}$$

Therefore, we denote the vector W of the initial conditions as such:

$$\mathrm{W}_L = \begin{bmatrix} \rho_L \\ u_L \\ p_L \end{bmatrix}, \qquad W_R = \begin{bmatrix} \rho_R \\ u_R \\ p_R \end{bmatrix}$$

Before solving the Riemann problem, it is important to understand the structure of the final state, as shown in Figure 1

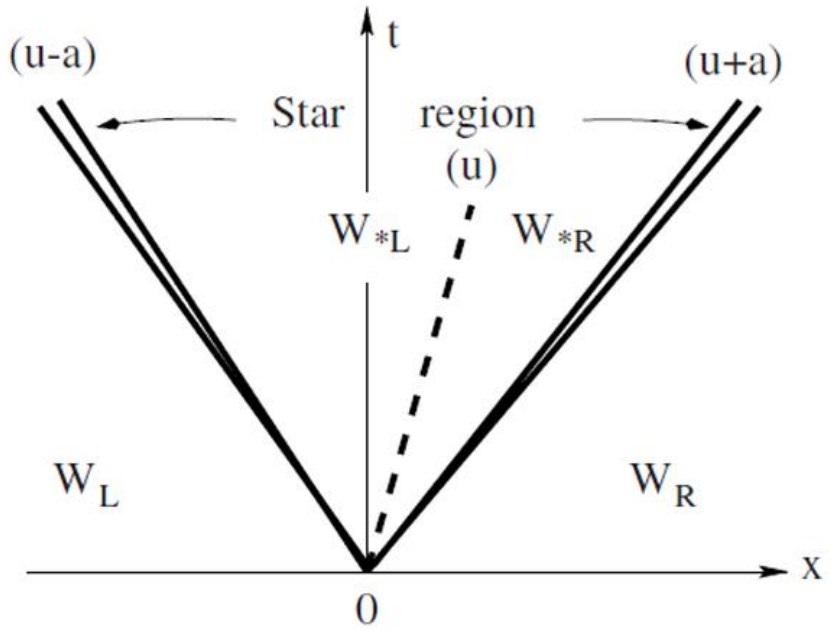

Figure 1 - general final state

In the final state the solution typically consists of three waves separating four constant states. From left to right, these are: the initial left state, a wave (either a shock or a rarefaction), the left side of the star region, a contact discontinuity, the right side of the star region, another wave (shock or rarefaction), and the initial right state.

Since each wave can be a shock or rarefaction wave according to the initial conditions there are four possible configurations to consider: (a) rarefaction- shock, (b) shock-rarefaction, (c) rarefaction-rarefaction, (d) shock- shock.

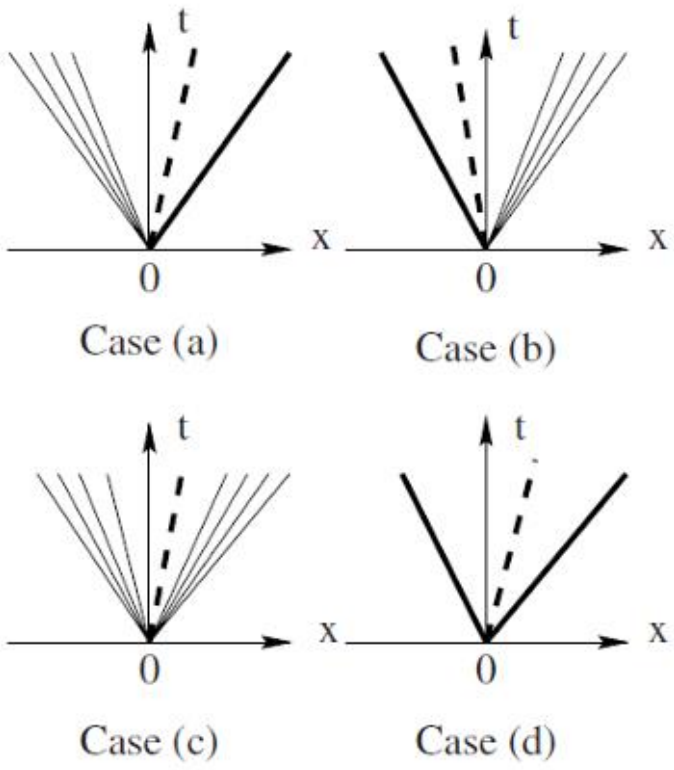

Figure 2 - four possible end states

Within the discontinuity region, also referred to as the star region, the pressure and velocity remain constant across both sides and are denoted by p_* , u_* respectively. However, the density can change across the discontinuity and is represented as ρ_{*L} , ρ_{*R} on the left and right side. Thus, the overall structure consists of three regions of constant pressure: two of which are known (the initial left and right states) and one, the star region, that must be determined. By calculating p_* we can determine whether the pressure between the initial left state and the star region increases or decreases, which indicates the formation of a shock wave or a rarefaction wave. The same reasoning applies between the star region and the initial right state. Thus, solving for the pressure p_* in the star region is the first step in solving the Riemann problem, as it dictates which wave structures form and which corresponding equations must be used to solve for the other flow parameters. This is done by solving for the root of the following function:

$$(9)\ f(p, W_L, W_R) \equiv f_L(p, W_L) + f_R(p, W_R) + (u_R - u_L) = 0$$

While we define the functions generally for the left and right sides ($I = R\ or\ L$) as:

$$(10)\ f_I(p, W_I) = \begin{cases} a(p - p_I) \cdot \left[\frac{A_I}{p + B_I}\right]^{\frac{1}{2}} : p > p_I\ (shock) \\ \frac{2a_I}{\gamma - 1} \cdot \left[\left(\frac{p}{p_I}\right)^{\frac{\gamma-1}{2\gamma}} - 1\right] : p < p_I\ (rarefaction) \end{cases}$$

Where:

$$A_I = \frac{2}{(\gamma + 1) \cdot \rho_I}\ , B_I = \frac{\gamma - 1}{\gamma + 1} \cdot p_I$$

And a represents the local speed of sound as:

$$a = \sqrt{\frac{\gamma \cdot p}{\rho}}$$

Since the function represented in Equation (9) is nonlinear, the simplest way to find the root is to use an iterative method, in this case the Newton-Raphson Method (NRM), as shown in Equation (11).

$$(11)\ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

In our case the parameter to solve for is p_* therefore we solve for the root $f(p_*) = 0$ using NRM according to Equations (9) and (11) . The initial guess for p_* is the average of p_L and p_R. The iteration continues until reaching the relative pressure change is smaller than the desired tolerance (Equation (12)), in our case a tolerance of $CHA < 10^{-6}$

$$(12)\ CHA = \frac{|p_n - p_{n-1}|}{\frac{1}{2}[p_n + p_{n-1}]}$$

If p_* is greater than the pressure of its neighboring state it creates a shock wave and if it is smaller, it creates a rarefaction wave

$$(13)\ \begin{cases} p_* > p_I & \Rightarrow shock\ wave \\ p_* < p_I & \Rightarrow rarefaction\ wave \end{cases} : I = R\ or\ L$$

Once p_* is obtained u_* is calculated the following:

$$(14)\ u_* = \frac{1}{2}(u_L + u_R) + \frac{1}{2}[f_R(p_*) - f_L(p_*)]$$

In the case of a shock wave:

Density is calculated:

$$(15)\ \rho_{*I} = \rho_I \left[\frac{\frac{p_*}{p_I} + \frac{\gamma-1}{\gamma+1}}{\frac{\gamma-1}{\gamma+1}\frac{p_*}{p_I} + 1}\right]$$

Shock speed is calculated:

$$(16)\ S_I = \begin{cases} u_L - a_L \left[\frac{\gamma+1}{2\gamma}\frac{p_*}{p_L} + \frac{\gamma-1}{2\gamma}\right]^{0.5} : I = L \\ u_R + a_R \left[\frac{\gamma+1}{2\gamma}\frac{p_*}{p_R} + \frac{\gamma-1}{2\gamma}\right]^{0.5} : I = R \end{cases}$$

In the case of rarefaction:

Density is calculated:

$$(17)\ \rho_{*_I} = \rho_I \left(\frac{p_*}{p_I}\right)^{\frac{1}{\gamma}}$$

In the case of a rarefaction wave, there is no single wave speed S, because the wave consists of a continuous expansion where the fluid gradually spreads out, leading to a smooth decrease in pressure and density across the region. Unlike a shock wave, which propagates at a fixed speed, a rarefaction wave is characterized by a fan of characteristic lines, each carrying information at a different speed. Therefore, instead of a single speed, we calculate the head speed S_{HI} and the tail speedS_{TI} for each rarefaction wave, which defines the boundaries of the expanding wave

$$(18)\ S_{HI} = \begin{cases} u_L - a_L : I = L \\ u_R + a_R : I = R \end{cases},\quad S_{TI} = \begin{cases} u_* - a_{*I} : I = L \\ u_* + a_{*I} : I = R \end{cases}$$

Where the sound of speed in the star region a_{*I} is calculated as follows:

$$a_{*I} = a_I \left(\frac{p_*}{p_I}\right)^{\frac{\gamma-1}{2\gamma}}$$

Inside the rarefaction fan, the primitive variables vary smoothly and are calculated as functions of space and time:

$$(19)\ W_{Ifan} = \begin{cases} \begin{cases} \rho = \rho_L \left[\frac{2}{\gamma+1} + \frac{\gamma-1}{(\gamma+1)a_L} \cdot \left(u_L - \frac{x}{t}\right)\right]^{\frac{2}{\gamma-1}} \\ u = \frac{2}{\gamma+1}\left[a_L + \frac{\gamma-1}{2} \cdot u_L + \frac{x}{t}\right] \\ p = p_L \left[\frac{2}{\gamma+1} + \frac{\gamma-1}{(\gamma+1)a_L} \cdot \left(u_L - \frac{x}{t}\right)\right]^{\frac{2\gamma}{\gamma-1}} \end{cases} : I = L \\ \begin{cases} \rho = \rho_R \left[\frac{2}{\gamma+1} - \frac{\gamma-1}{(\gamma+1)a_R} \cdot \left(u_R - \frac{x}{t}\right)\right]^{\frac{2}{\gamma-1}} \\ u = \frac{2}{\gamma+1}\left[-a_R + \frac{\gamma-1}{2} \cdot u_R + \frac{x}{t}\right] \\ p = p_R \left[\frac{2}{\gamma+1} - \frac{\gamma-1}{(\gamma+1)a_R} \cdot \left(u_R - \frac{x}{t}\right)\right]^{\frac{2\gamma}{\gamma-1}} \end{cases} : I = R \end{cases}$$

The next step is to define the vector W as a function of position and time, based on the region the point (x, t) belongs to. It's important to note that x is centered around the point of discontinuity:

$$x = x_i - x_0$$

The relation between $\frac{x}{t}$ and u_* determines whether the point lies to the left or right of the contact discontinuity and the relation between $\frac{x}{t}$ and the wave speeds (S_I, S_{TI}, S_{HI}) determines whether the point lies to the left or right of the wave.

If $\frac{x}{t} < u_*$ we are on the left side of the discontinuity and so:

We determine the type of wave by comparing the pressure (Equation (13))

If the wave created is a shock wave we compare the characteristic speed to the shock speed to see if we are before the wave (in the initial left state) or after the wave (in the star region):

$$(20)\ W(x,t) = \begin{cases} W_L & if\ \frac{x}{t} \le S_L \\ W_{*L}^{sho} & if\ S_L < \frac{x}{t} \end{cases}$$

If the wave created is a rarefaction wave, there will be a change from the initial left state to the fan of the wave and then to the star region as a function of the wave speeds:

$$(21)\ W(x,t) = \begin{cases} W_L & if\ \frac{x}{t} \le S_{HL} \\ W_{Lfan} & if\ S_{HL} < \frac{x}{t} \le S_{TL} \\ W_{*L}^{fan} & if\ S_{TL} < \frac{x}{t} \end{cases}$$

If $u_* < \frac{x}{t}$ we are on the right side of the discontinuity, the process is almost the same:

We determine the type of wave by comparing the pressure (again, Equation (13))

If the wave created is a shock wave we compare the characteristic speed to the shock speed to see if we are before the wave (in the star region) or after the wave (in the initial right state):

$$(22)\ W(x,t) = \begin{cases} W_{*R}^{sho} & if\ \frac{x}{t} < S_R \\ W_R & if\ S_R < \frac{x}{t} \end{cases}$$

If the wave created is a rarefaction wave, there will be a change from the star region state to the fan of the wave and then to the initial right state as a function of the wave speeds:

$$(23)\ W(x,t) = \begin{cases} W_{*R}^{fan} & if\ \frac{x}{t} \le S_{TR} \\ W_{Rfan} & if\ S_{TR} < \frac{x}{t} \le S_{HR} \\ W_R & if\ S_{HR} < \frac{x}{t} \end{cases}$$

Where:

$$(24)\ W_{*I}^{sho} = \begin{bmatrix} \rho_{sho*I} \\ u_* \\ p_* \end{bmatrix},\ W_{*I}^{fan} = \begin{bmatrix} \rho_{fan*I} \\ u_* \\ p_* \end{bmatrix}$$

And ρ_{sho*I} and ρ_{fan*I} correspond to how the density is calculated either by Equation (15) or Equation (17) respectively.

Finally, we have a solution for $W(x,t)$ for each point in space and time, for clarity we will sum up the steps taken:

1. Calculate the pressure in the star region using NRM. Equations: (9) (11) (12)
2. Calculate the velocity in the star region. Equations: (14)

3. Identify what type of wave is formed and calculate the density accordingly. Equations: (13) (15) (17)
4. Calculate the value of W according to the position in space and time and in relation to the wave speeds. Equations: (19) - (24)

Finding W is equivalent to finding U, and can be easily transformed using the definitions of the conserved variables (Equations (2))

It is important to note that the exact solution for the Riemann problem is only a function of space since we are calculating the state at a given end time

Godunov Method:

To numerically solve the Euler equations, we employ the Godunov method. This method uses a finite volume scheme that necessitates the discretization of the computational domain in both space and time.

In the one-dimensional case, the spatial domain is simply a line of length L. We divided the line into N uniform cells, each of width:

$$(25)\ \Delta x = \frac{L}{N}$$

Each cell is indexed by i, and the solution is advanced in discrete time steps indexed by n (see Figure 3). Time is discretized according to the Courant–Friedrichs–Lewy (CFL) condition to ensure numerical stability:

$$(26)\ \Delta t = CFL \cdot \frac{\Delta x}{max\ (|u|+a)}$$

where u is the velocity, a is the speed of sound, and the maximum is taken over all cells in the domain. At each time step, the Riemann problem, as described earlier, is solved at the interface between every two adjacent cells. The left and right states at each interface are used as initial conditions. By solving the Riemann problem at each interface, we can use the conserved values in U to obtain the primitive values and calculate the flux as seen in Equation (2). The numerical flux, which represents the flow of conserved quantities across that interface, is then used to update the conserved variables in each cell using the finite volume update formula:

$$(27)\ U_i^{n+1} = U_i^n + \frac{\Delta t}{\Delta x}\left[F_{i-\frac{1}{2}} - F_{i+\frac{1}{2}}\right]$$

The flux F is calculated at every interface between adjacent cells, and is therefore indexed at the half-points between cell centers i.e. $i \pm \frac{1}{2}$. As a result, there are no flux values defined at the outer boundaries of the domain (specifically at $i = 0\ , i = N$). To handle this, the boundary conditions are updated at each time step to preserve the initial values at the left and right edges of the domain. This ensures that the solution remains consistent with the physical setup throughout the simulation.

$$U_0^n = U_0^{n+1}, \qquad U_N^n = U_N^{n+1}$$

This process is repeated iteratively at each time step until the desired final time is reached.

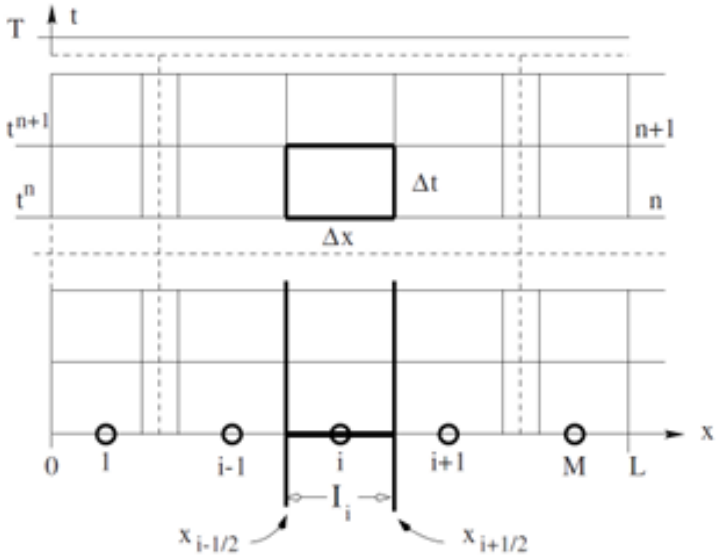

Figure 3 - grid representation for numerical scheme

Riemann Solvers:

Utilizing the Godunov method requires a means of calculating the flux to use in the update equation. In this paper, three methods are employed for this purpose: Exact, HLLC, and Roe-Pike solvers. These solvers are designed to handle the complex wave structures arising from the Euler equations. For instances involving shock waves, the fundamental principles governing these discontinuities are the Rankine-Hugoniot conditions. These are a set of algebraic equations that describe the conservation of mass, momentum, and energy across a shock wave in a fluid, relating the fluid properties (density, velocity, and pressure) before and after the shock. The Exact solver explicitly applies these conditions when determining fluxes across shocks, while the approximate solvers (HLLC and Roe-Pike) are formulated to accurately capture these jumps, implicitly or through linearization.

Exact Solver: The exact solver implements the method for the exact solution. However, instead of calculating it for the final time, as in a full solution to the Riemann problem, it is applied at each cell interface. In this context, the flux is calculated using the cell interface as the point of discontinuity, and the states within the adjacent cells provide the left and right initial conditions for the Riemann problem. It is important to note that, in this case, the Riemann problem is solved at each interface by evaluating the solution at $\frac{x}{t} = 0$. This corresponds to the assumption that a discontinuity exists at the cell interface.

HLLC solver: The HLLC method is an approximate Riemann solver that simplifies the complex wave structure. It still assumes a three-wave structure: a left-moving wave, a right-moving wave, and a central contact discontinuity (as depicted in Figure 1). However, instead of solving the detailed interactions of all possible waves, HLLC solver estimates the speeds of these three waves and uses these estimates to calculate the fluxes at cell interfaces.

The HLLC solver estimates wave speeds based on characteristic speeds, which represent the rates at which information propagates within the fluid. The solver approximates these speeds using the fluid properties of the left and right states at the cell interface to capture the dominant movement of information in the Riemann problem. To refine these estimates, HLLC uses the average density, velocity, and speed of sound, along with a pressure estimate from a Primitive Variable Riemann Solver (PVRS), for the star region (the region between the left and right waves).

$$(28)\ p_{pvrs} = p_{avg} - \frac{1}{2}(u_R - u_L) \cdot \rho_{avg} \cdot a_{avg}$$

$$(29)\ p_* = \max\left(0, p_{pvrs}\right)$$

Where the average values are simple arithmetic averages:

$$x_{avg} = \frac{x_L + x_R}{2}$$

The wave speed estimates are then assumed to be:

$$(30)\ S_L = u_L - a_L q_L\ ,\ S_R = u_R + a_R q_R$$

Where:

$$(31)\ q_I = \begin{cases} 1 \\ \left(1 + \frac{\gamma+1}{2\gamma} \cdot \frac{p_*}{p_I - 1}\right)^{\frac{1}{2}} \end{cases} \begin{matrix} if\ p_* \le p_I \\ if\ p_* \ge p_I \end{matrix}$$

The speed of the contact discontinuity S_* can be calculated using estimates for the left S_L and right S_R wave speeds:

$$(32)\ S_* = \frac{p_R - p_L + \rho_L u_L (S_L - u_L) - \rho_R u_R (S_R - u_R)}{\rho_L (S_L - u_L) - \rho_R (S_R - u_R)}$$

For the HLLC solver, the fluxes are defined as follows:

$$(33)\ F_{i+\frac{1}{2}} = \begin{cases} F_L\ if\ \ 0 \le S_L \\ F_{*L}\ if\ \ 0 \le S_* \\ F_{*R}\ if\ \ 0 \le S_R \\ F_R\ if\ \ S_R \le 0 \end{cases}$$

Where:

$$(34)\ F_{*I} = F_I + S_I(U_{*I} - U_I)$$

And:

$$(35)\ U_{*I} = \rho_I \frac{(S_I - u_I)}{S_I - S_*} \begin{bmatrix} 1 \\ S_* \\ \frac{E_I}{\rho_I} + (S_* - u_I) \cdot \left(S_* + \frac{p_I}{\rho_I \cdot (S_I - u_I)}\right) \end{bmatrix}$$

To summarize, the steps involved in using the HLLC solver are:

1. Calculate the pressure estimate. Equation: (28), (29)
2. Calculate wave speed estimates and wave speed of discontinuity. Equations: (30) - (32)
3. Calculate the flux according to the speed estimates. Equations: (33) - (35)

Roe-Pike Solver: The Roe-Pike method approximates the Riemann problem by linearizing it. This is achieved by assuming that the matrix A (Equation (6)) is a function of the data states U_L, U_R ($A \equiv \tilde{A}(U_L, U_R)$. This makes the matrix constant for every pair of cells, rather than the Jacobian matrix which varies with the flow variables. This simplification transforms the quasi-linear Euler equations into a linear system.

$$\begin{cases} U_t + \tilde{A} U_x = 0 \\ U(x,0) = \begin{cases} U_L, & x < 0 \\ U_R, & x > 0 \end{cases} \end{cases}$$

In this simplified linear form, the solution at the cell interface ($\frac{x}{t} = 0$) can be solved exactly using the following solution:

$$(36)\ F_{i+\frac{1}{2}} = \frac{1}{2}(F_L + F_R) - \frac{1}{2}\sum_{i=1}^{m} \tilde{\alpha}_i \cdot \left|\tilde{\lambda}_i\right| \cdot \tilde{K}_i$$

Where λ are the eigenvalues of the constant matrix A, K are the corresponding right eigenvectors and α are the wave strengths determined by the initial left and right states. In the original Roe method, an averaged Jacobian matrix $\tilde{A}$ is first calculated to determine the eigenvalues and right eigenvectors. However, the updated Roe-Pike approach avoids explicitly constructing the matrix. Instead, it uses averages of density, velocity, enthalpy, and speed of sound to directly evaluate the eigenvalues, right eigenvectors, and wave strengths needed in Equation (36). For the one-dimensional problem being considered, the number of equations (m) in Equation (36) equals 3.

The average values are calculated as follows:

$$(37)\ \tilde{\rho} = \sqrt{\rho_L \cdot \rho_R}$$

$$(38)\ \tilde{u} = \frac{\sqrt{\rho_L} \cdot u_L + \sqrt{\rho_R} \cdot u_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}$$

$$(39)\ \tilde{H} = \frac{\sqrt{\rho_L} \cdot H_L + \sqrt{\rho_R} \cdot H_R}{\sqrt{\rho_L} + \sqrt{\rho_R}}$$

$$(40)\ \tilde{a} = \left((\gamma - 1) \cdot \left(\tilde{H} - \frac{1}{2}\tilde{u}^2\right)\right)^{\frac{1}{2}}$$

The eigenvalues are then calculated:

$$(41)\ \tilde{\lambda}_1 = \tilde{u} - \tilde{a}$$

$$(42)\ \tilde{\lambda}_2 = \tilde{u}$$

$$(43)\ \tilde{\lambda}_3 = \tilde{u} + \tilde{a}$$

And the right eigenvectors are calculated:

$$(44)\ \tilde{K}_1 = \begin{bmatrix} 1 \\ \tilde{u} - \tilde{a} \\ \tilde{H} - \tilde{u}\tilde{a} \end{bmatrix}$$

$$(45)\ \tilde{K}_2 = \begin{bmatrix} 1 \\ \tilde{u} \\ \frac{1}{2}\tilde{u}^2 \end{bmatrix}$$

$$(46)\ \tilde{K}_3 = \begin{bmatrix} 1 \\ \tilde{u} + \tilde{a} \\ \tilde{H} + \tilde{u}\tilde{a} \end{bmatrix}$$

Lastly the wave strengths are calculated:

$$(47)\ \tilde{\alpha}_1 = \frac{1}{2\tilde{a}^2}[(p_R - p_L) - \tilde{\rho}\tilde{a}(u_R - u_L)]$$

$$(48)\ \tilde{\alpha}_2 = (\rho_R - \rho_L) - \frac{p_R - p_L}{\tilde{a}^2}$$

$$(49)\ \tilde{\alpha}_3 = \frac{1}{2\tilde{a}^2}[(p_R - p_L) + \tilde{\rho}\tilde{a}(u_R - u_L)]$$

The calculated eigenvalues, eigenvectors, and wave strengths are then used to compute the flux according to Equation (36).

Although this solver is the simplest to follow, to maintain consistency with the presentation of other solvers and to ensure clarity, a summary of the steps is presented below:

1. Calculate average values. Equations: (37) - (40)
2. Calculate eigen values, eigenvectors and wave strength. Equations: (41) - (49)
3. Calculate the flux. Equations: (36)

Results and Discussion

The following table represents all relevant values for the compared cases:

Table 1 - initial conditions for test cases

Test case	ρ_L	u_L	p_L	ρ_R	u_L	p_L	x_0	t_{end}
1	1.0	0.75	1.0	0.125	0.0	0.1	0.3	0.2
4	5.99924	19.5975	460.894	5.99242	-6.19633	46.0950	0.4	0.035

The value for Ratio of specific heats is constant for both cases: $\gamma = 1.4$

The following graphs show the values at $t = t_{end}$ of density, velocity, pressure, and internal energy as functions of space for the test cases. Internal energy as represented in the graphs is calculated using the primitive values as follows:

$$(50)\ e = \frac{p}{(\gamma-1)\cdot\rho}$$

The data points for the results from Toro were extracted from photos of the graphs using the online tool at WebPlotDigitizer **[2]**

First, the results obtained from the custom-built code will be compared against Toro's reference solutions to validate the code's legitimacy and demonstrate its accuracy. Subsequently, the performance of the three different numerical solvers (Exact, HLLC, and Roe-Pike) will be assessed by comparing their solutions against the exact analytical solution for the chosen test cases. Finally, a comparative analysis of the computational cost for each solver will be conducted, including an investigation into how increased spatial resolution affects their respective running times.

For test case 1 the following results are obtained:

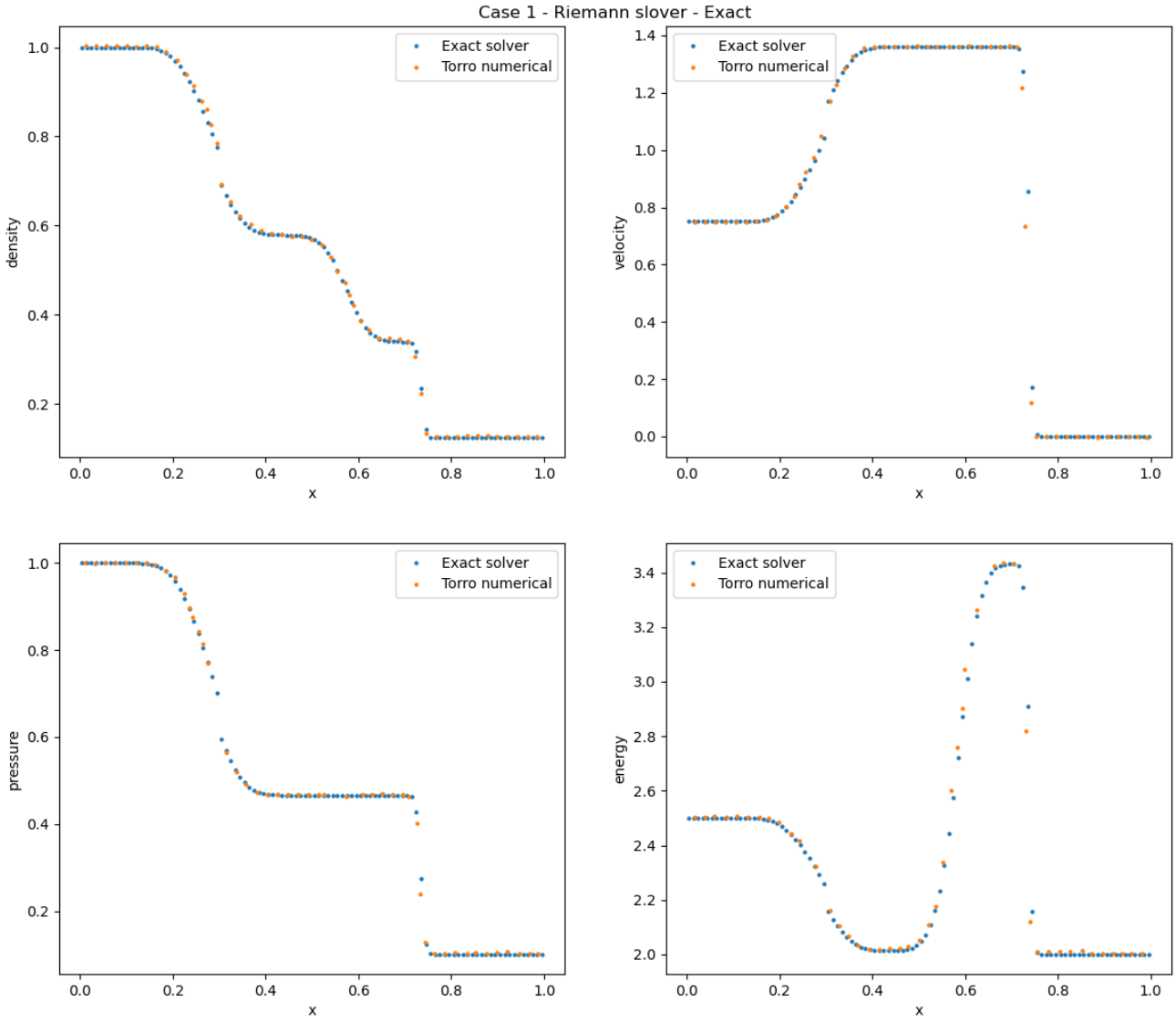

Figure 4 - Case 1: Exact solver

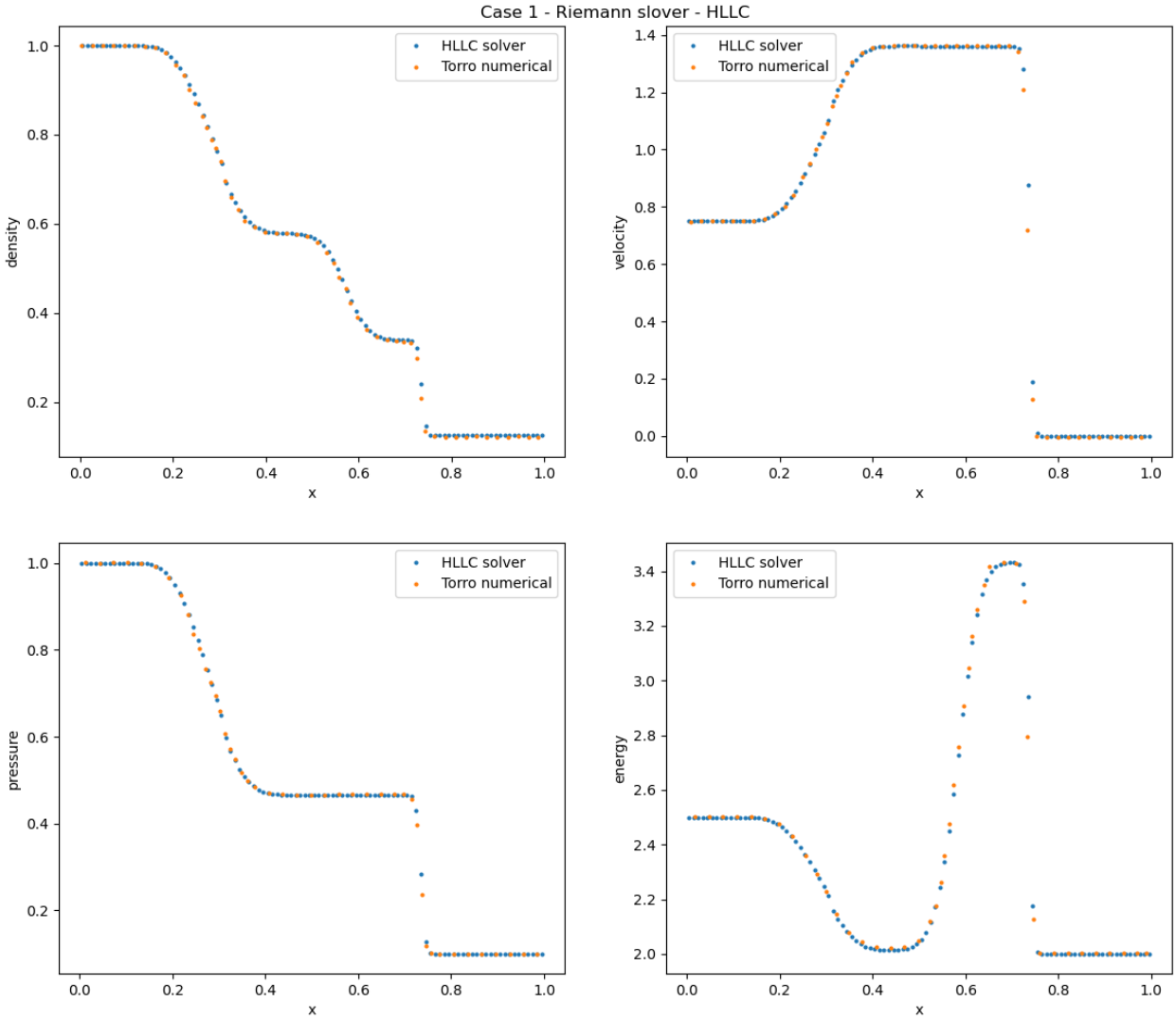

Figure 5 - Case 1: HLLC solver

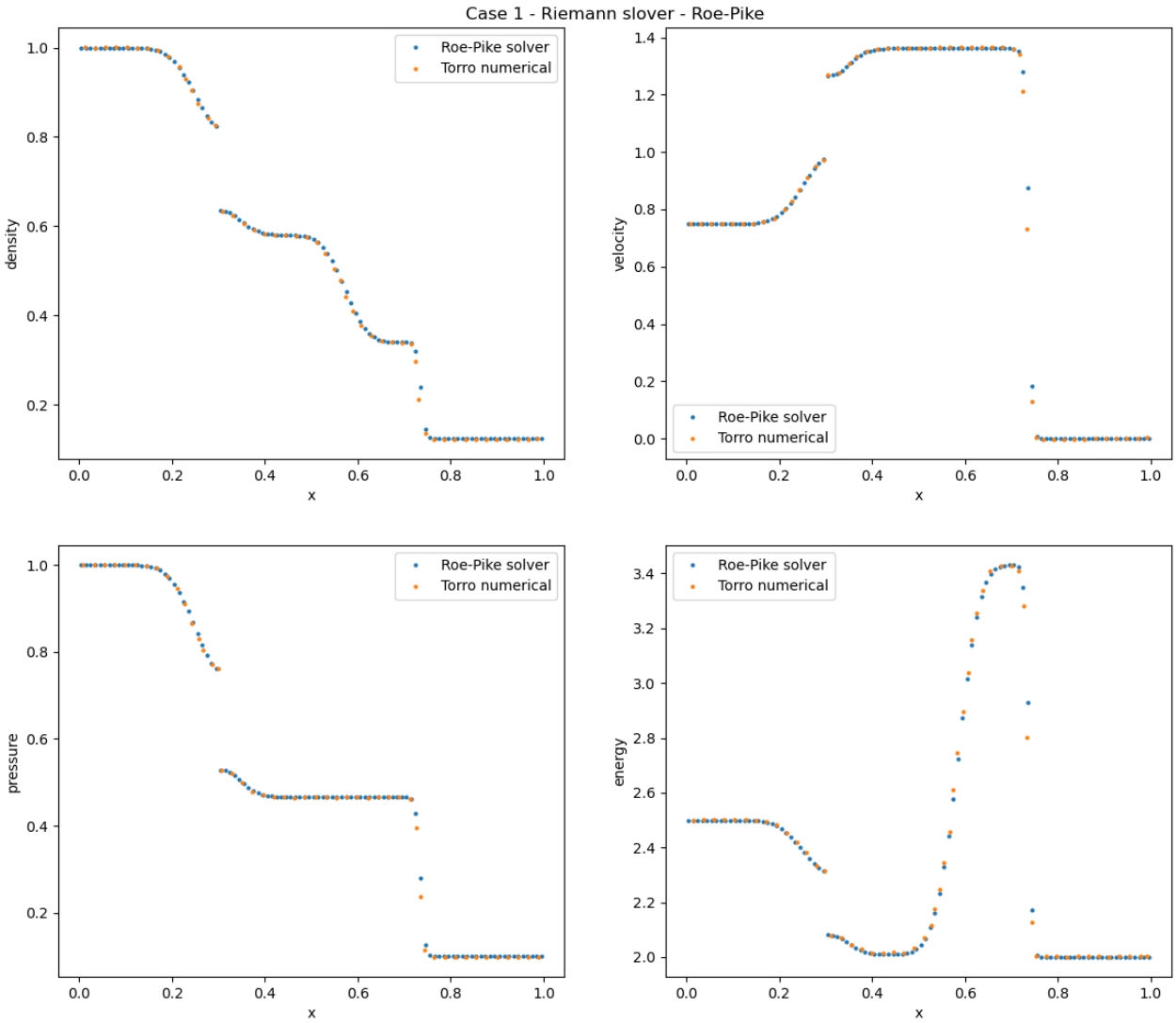

Figure 6 - Case 1: Roe-Pike solver

In Figures 4-6, the blue data points represent the results from the custom-built code, while the orange data points are extracted from Toro's reference solutions. It is clear from these figures that the results for Test Case 1 demonstrate excellent agreement with the numerical solutions presented by Toro across all three solvers.

For test case 4 the following results are obtained:

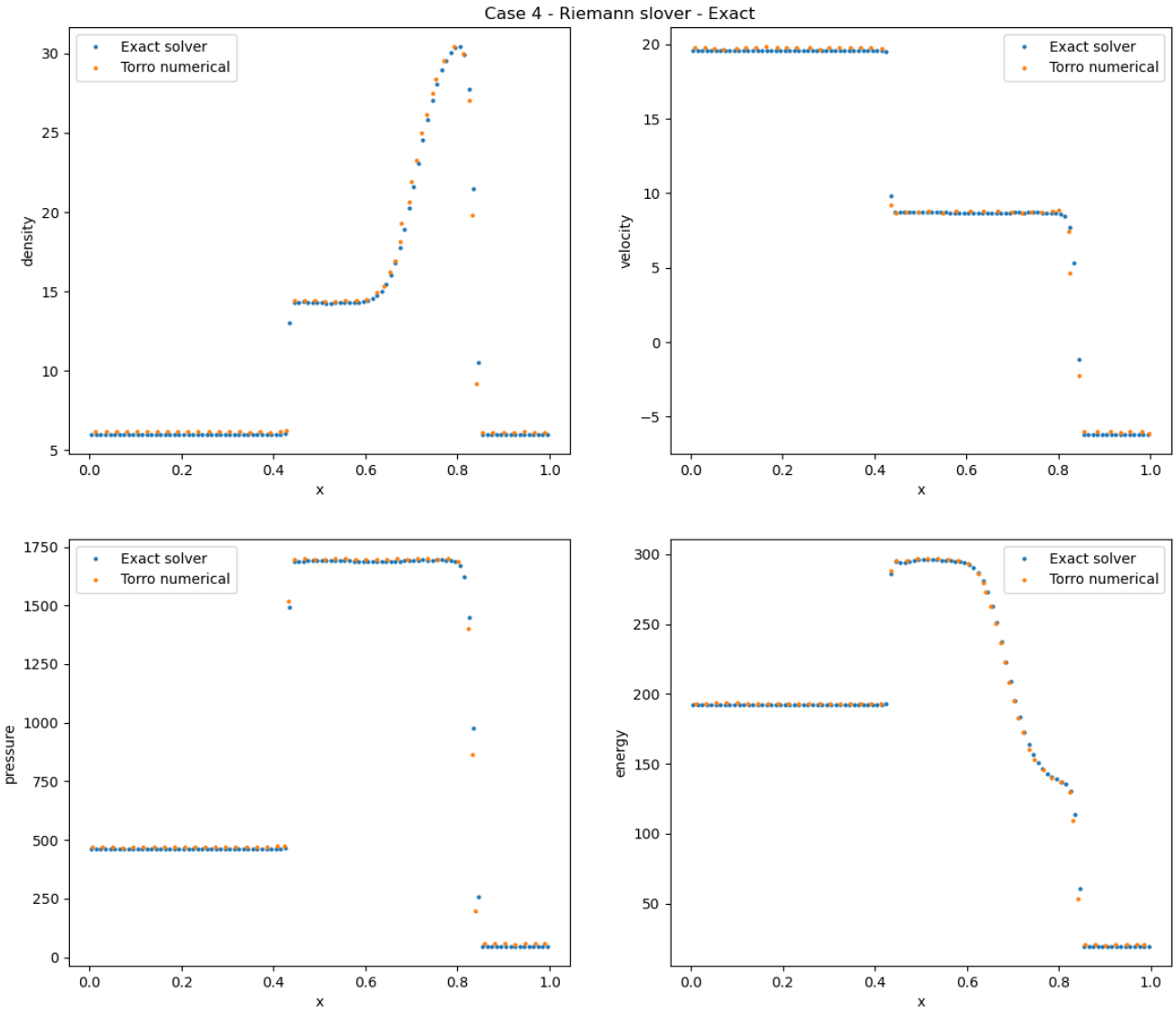

Figure 7 - Case 4: Exact solver

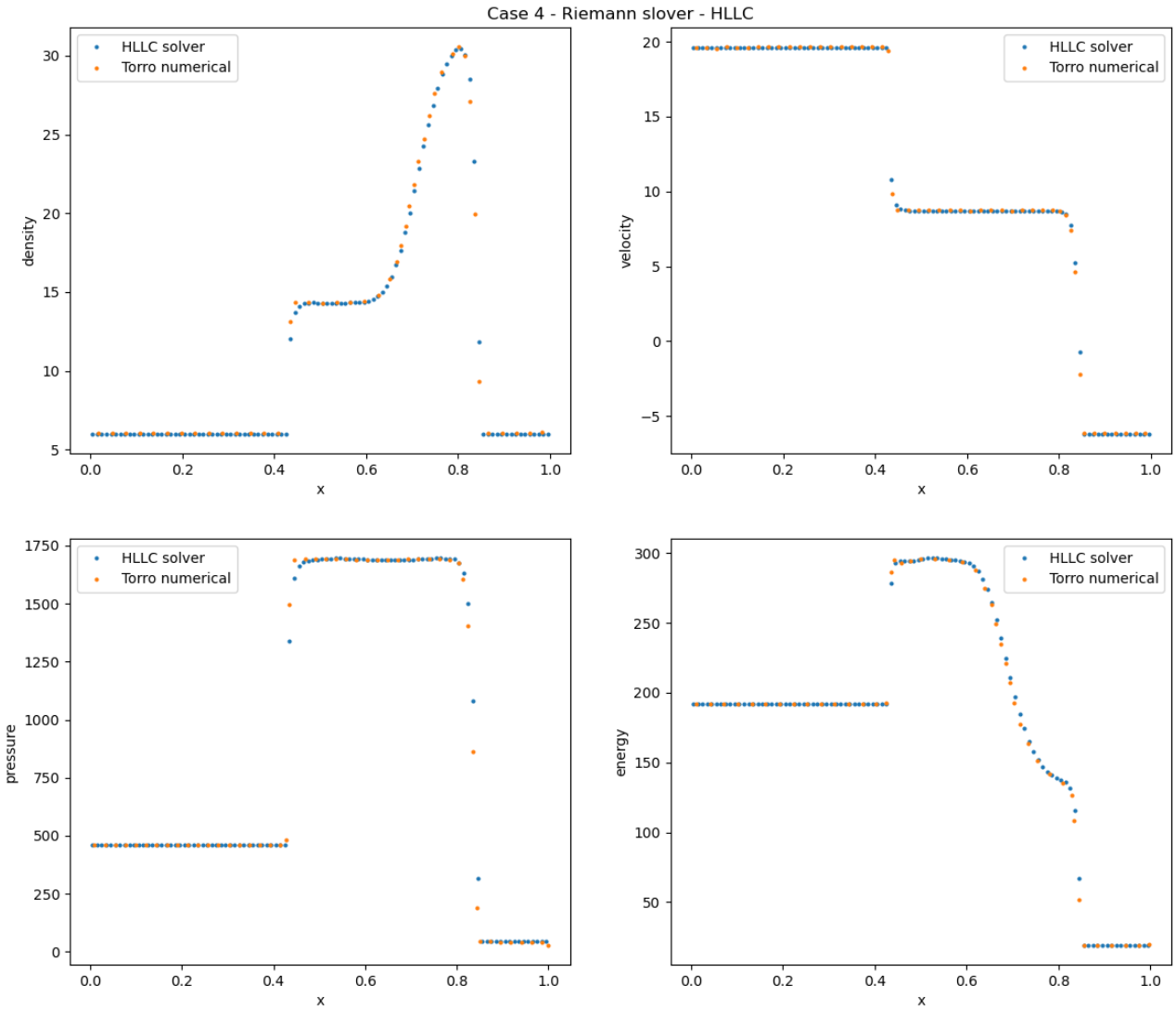

Figure 8 - Case 4: HLLC solver

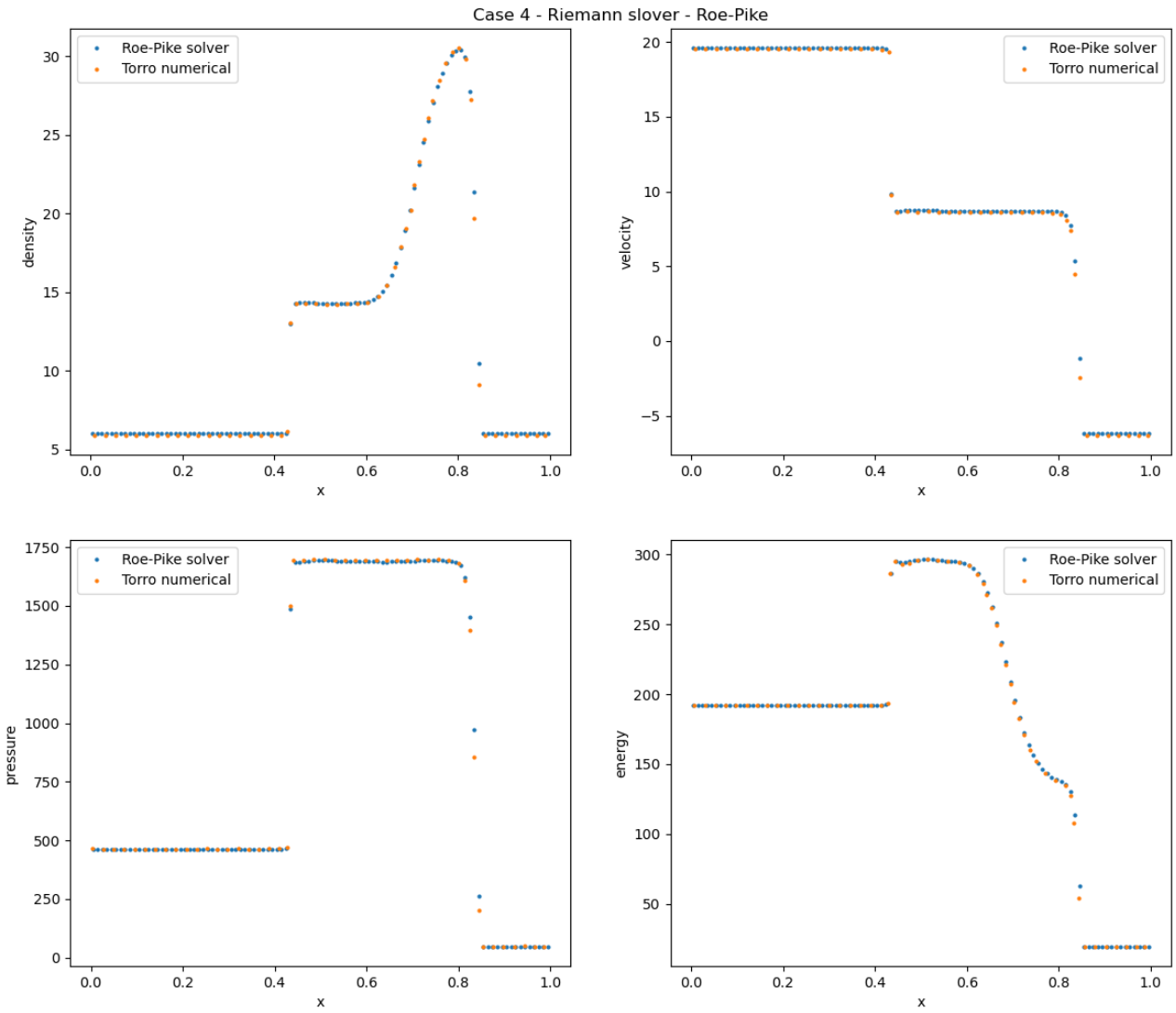

Figure 9 - Case 4: Roe-Pike solver

Consistent with the previous figures, Figures 7-9 display the results for Test Case 4, with blue data points representing the custom-built code and orange points corresponding to Toro's reference data (2013). In Figure 8, which illustrates the HLLC solver's performance, slight inconsistencies are observable. Specifically, in the pressure and density profiles around x=0.45, within the transition zone of the shock wave, Toro's results show a slightly sharper transition compared to those of the custom-built code. However, this difference is minor and is likely within an acceptable error margin. Therefore, we conclude that there is overall good agreement for Test Case 4 across all implemented solvers.

In conclusion, the custom-built code effectively reconstructs the numerical solvers for the one-dimensional Euler equations. This verification against established benchmarks demonstrates its ability to correctly solve the Euler equations using the Godunov method and the specified Riemann solvers. Further verification is provided by a direct comparison of the numerical results generated by the custom-built code against the exact analytical solutions, as seen in Figure 10 and Figure 11.

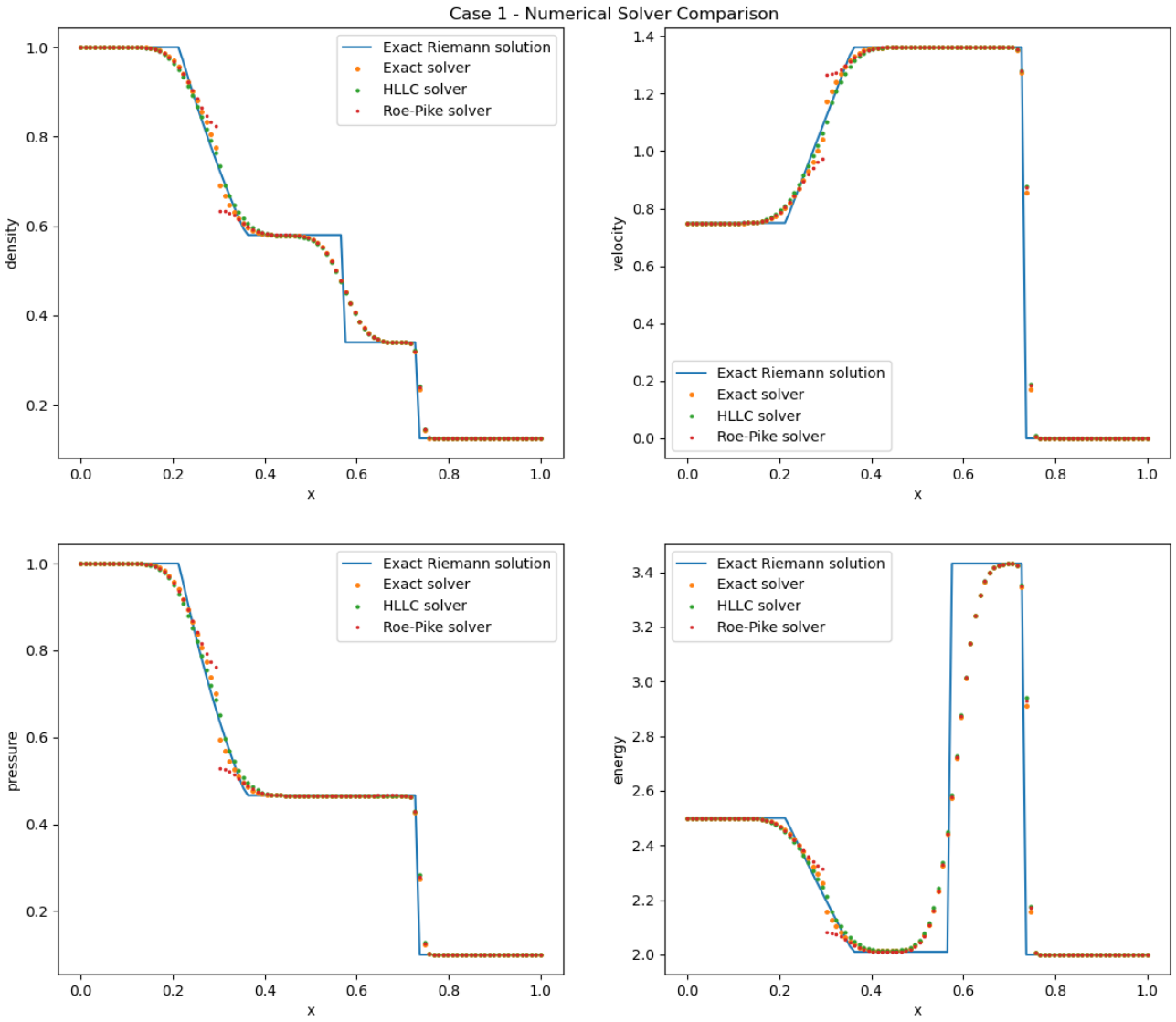

Figure 10 - Case 1: comparison of numerical solvers

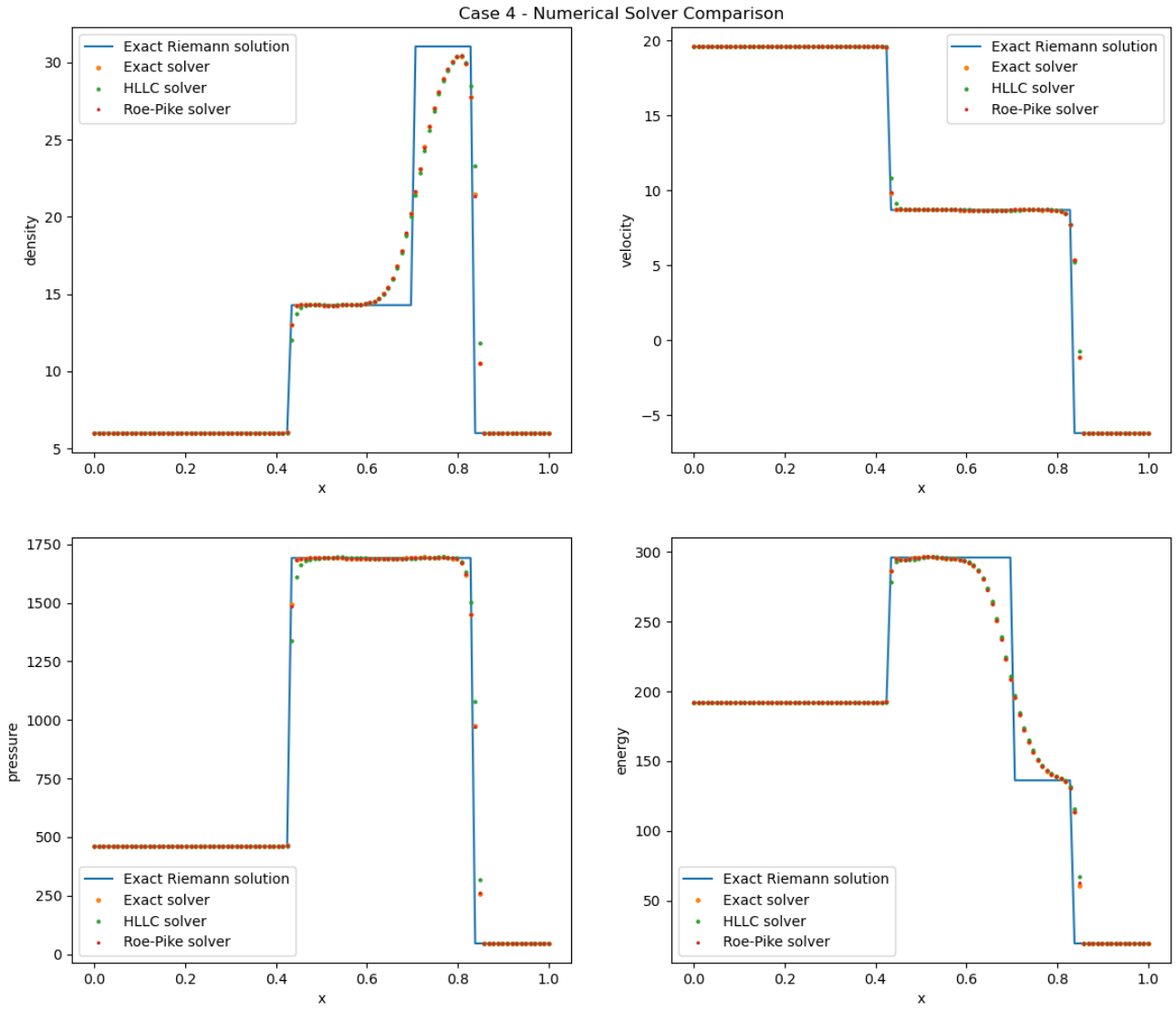

Figure 11- Case 4: comparison of numerical solvers

In Figure 10 and Figure 11, the solid line represents the exact analytical solution, while the discrete points indicate the results from the various numerical solvers, each distinguished by a different color as indicated in the legend. Figure 10 presents the comparative results for Case 1, and Figure 11 displays those for Case 4.

From these figures, three main observations can be drawn:

1. **Transition at the Discontinuity:** A closer examination of the density and internal energy graphs, specifically around x=0.58 for Case 1 and x=0.7 for Case 4, reveals that the transition across the contact discontinuity wave is smoother and less instantaneous than in the analytical result. This characteristic demonstrates that, for all numerical solvers—including the one implementing the exact Riemann solution—the Godunov method inherently struggles to resolve instantaneous discontinuities perfectly.

2. **Transition at the rarefaction wave**

   For Case 1, where the left-moving wave is a rarefaction, the numerical results notably diverge from the analytical solution across all parameters, particularly around x=0.3. In this scenario, the exact solver consistently provides the most accurate results, with the HLLC solver as a close second. The Roe-Pike solver, however, exhibits a distinct skewness in its representation of the rarefaction wave, causing its results to veer noticeably away from the analytical profile.

3. **Transition at the shock wave**

   In Case 1, the right-moving wave is a shock wave, and in Case 4, both the left and right waves are shock waves (x=0.72 in Case 1 and at x=0.42, x=0.82 in Case 4). For both cases and across all parameters, good agreement is observed between the numerical and analytical results. A subtle advantage of the approximate solvers (HLLC and Roe-Pike) over the exact solver can be noted for the left wave in Case 4 (around x=0.42). Here, a slightly sharper and more accurate result is provided by the HLLC and Roe-Pike solvers for all parameters, most visibly in the pressure profiles, however the difference is minor.

This difference in numerical behavior between the transition over contact discontinuities and shock waves is attributed to the artificial diffusion introduced by the Godunov method. The method fundamentally represents flow variables as cell-averaged quantities, and the process of calculating these averages at each time step naturally smooths out sharp gradients over the cell width. This averaging implicitly acts as a form of numerical diffusion.

For contact discontinuities, which are characterized primarily by jumps in density (and internal energy) while pressure and velocity remain continuous, this numerical diffusion leads to smearing because there is no mechanism to counteract it. Conversely, in the non-linear physics of a shock wave there is a simultaneous and strong jumps across all primitive variables (density, velocity, and pressure). the Riemann problem is designed to accurately predict the specific, discontinuous changes (Rankine-Hugoniot jump conditions) that occur, enabling the numerical method to maintain a much sharper profile for shock waves, effectively counteracting the spreading effect of numerical diffusion.

The results for the transition in rarefaction waves present a noticeable skewing behavior. Unlike shocks or contact discontinuities, rarefaction waves are continuous, smooth expansions. While numerical methods inherently introduce some diffusion by discretizing these continuous gradients into cell-averaged values, leading to a degree of spreading in all solvers, the specific deviations observed, particularly with the Roe-Pike method, require further explanation.

For the Exact and HLLC solvers, their results closely follow the analytical line but exhibit a slight skewing. This minor deviation is primarily due to the inherent numerical diffusion of the finite volume method and the discrete nature of the computational grid, which subtly spreads out even smooth features.

The Roe-Pike solver, however, displayed a more distinct and problematic skewing pattern for rarefaction waves. This behavior is rooted in its linearization approach. Roe's linearization fundamentally struggles with rarefaction waves because it attempts to represent a continuous, smooth expansion (where fluid properties change smoothly) using a series of constant-speed waves derived from a single averaged state. This mismatch leads to an "overly compressive" behavior; that is to say, the scheme does not allow the fluid to expand as much as it physically should. Instead, the solver forces the continuous expansion into a structure resembling a discontinuity, creating a numerical approximation that is too steep. For example, in a sonic rarefaction wave (like that in Test 1), the fluid velocity crosses the speed of sound (u=a), the characteristic speed $\lambda_1 = u - a$ changes sign. In such cases, this overly compressive behavior can result in Roe's method producing an unphysical discontinuity within the rarefaction fan. This unphysical jump violates the entropy condition, a fundamental physical principle ensuring characteristics spread out in rarefactions. This phenomenon is a known limitation of the original Roe solver, which typically requires an entropy fix, a topic beyond the scope of this paper.

Lastly, we analyze the computational cost associated with each solver. The following table presents the execution time (in seconds) for each solver across different test cases and with varying number of Grid Cells (N) which corresponds to spatial resolutions.

Table 2 computational time as a function of grid cells and solvers

Solver / (N)	Exact	HLLC	Roe-Pike
100	0.43603	0.09131	0.12098
500	8.04628	2.18608	2.90358
800	19.57577	5.47947	7.32855
1000	29.66863	8.74637	11.68999

Solver / (N)	Exact	HLLC	Roe-Pike
100	0.69799	0.16174	0.23391
500	14.79478	3.96755	5.5892
800	36.48576	10.11533	14.28297
1000	56.45312	15.74325	22.25641

To visualize the change in computational cost as resolution increases, the results are also presented graphically and fitted with a power law trend line:

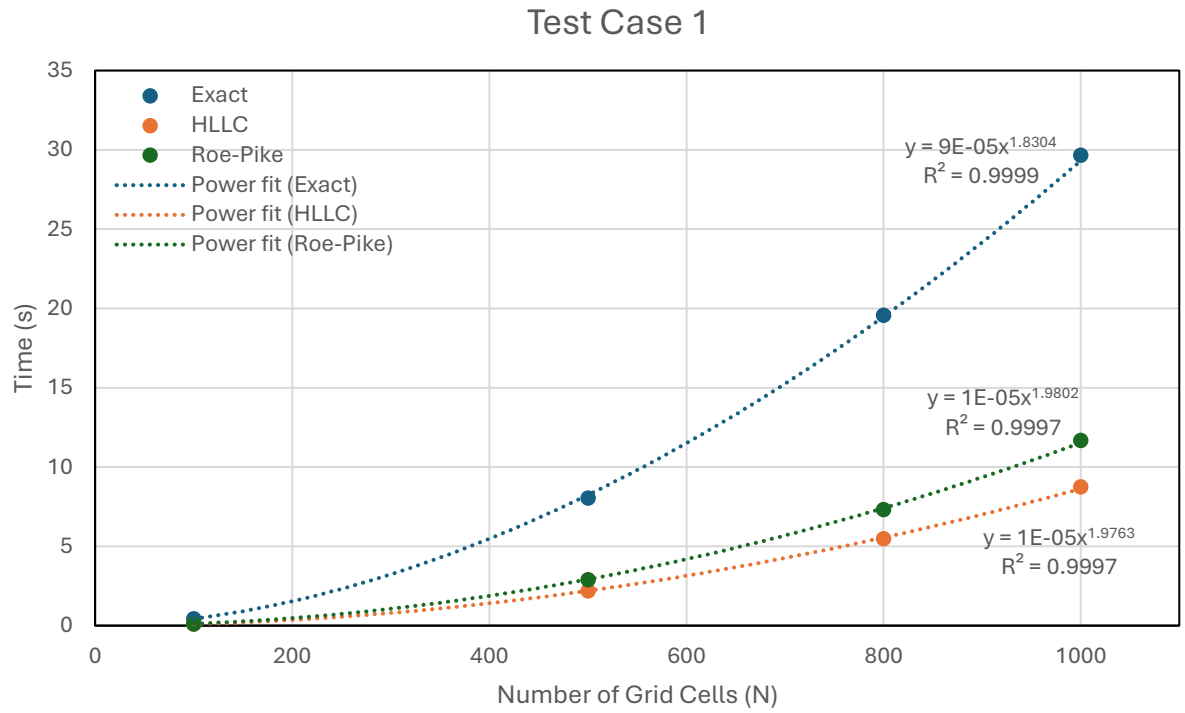

Figure 12 - Case 1: comparison of computational cost as a function of grid cells

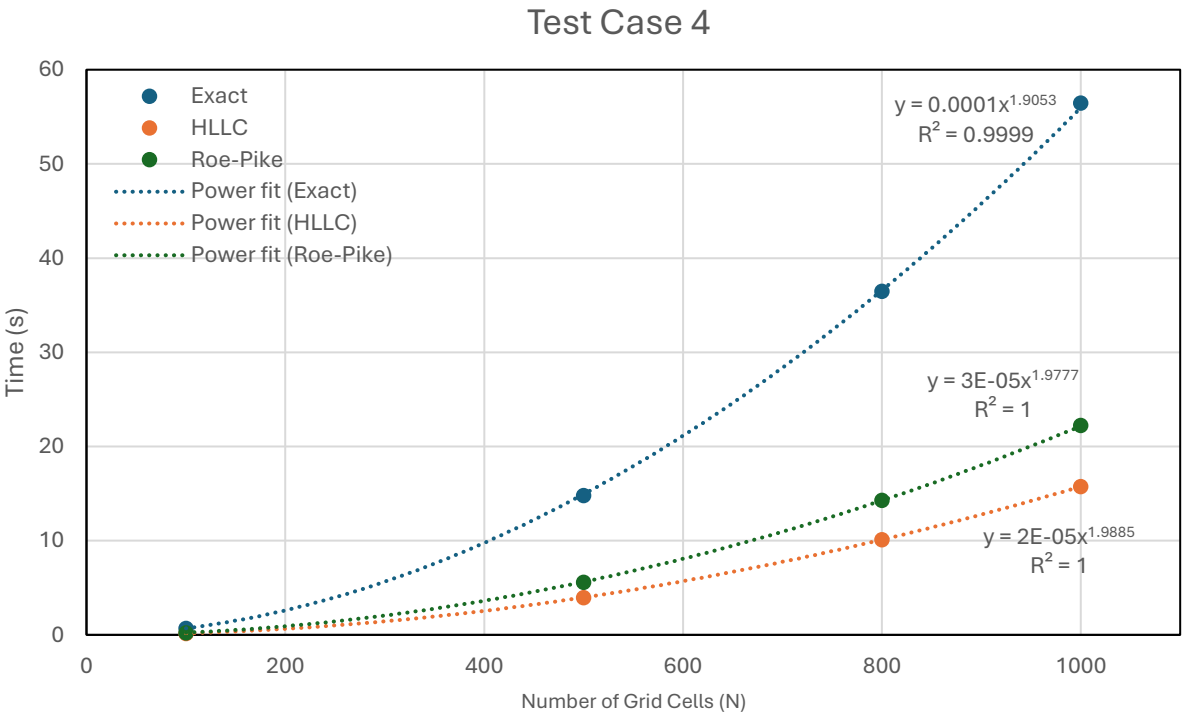

Figure 13 - Case 4: comparison of computational cost as a function of grid cells

The graphs in Figure 12 and Figure 13 clearly show that the computational cost of the Exact solver increases significantly with rising resolution. The HLLC and Roe-Pike solvers exhibit comparable performance, though the HLLC solver holds a distinct advantage. At the highest resolution, the Exact solver takes over three times longer to run compared to the HLLC solver. This outcome is expected, as the exact method rigorously solves the Riemann problem at each cell interface, whereas the approximate solvers employ various simplifications or "shortcuts."

The HLLC solver is consistently faster than the Roe-Pike solver. This difference in computational efficiency likely stems from the number of calculations required per cell. While the HLLC solver primarily needs to compute a pressure estimate, three wave estimates, and then the intermediate $U_*$and flux F_* states, the Roe-Pike solver must calculate three eigenvalues, three eigenvectors, and three wave strengths per cell. Even accounting for the various averaging steps, the Roe-Pike method generally involves more complex computations per cell, leading to its slightly higher computational cost.

Conclusions

This study successfully implemented and analyzed three different Riemann solvers—Exact, HLLC, and Roe-Pike—within a custom-built Godunov-type finite volume code for solving the one-dimensional Euler equations.

The verification of the custom-built code against established benchmark solutions from Toro demonstrated excellent agreement, confirming the code's legitimacy and its ability to correctly implement the underlying numerical methods.

In terms of accuracy when compared to the exact analytical solution, several key observations were made regarding the solvers' ability to resolve different wave types:

1. **Contact Discontinuities:** All numerical solvers, including the one using the exact Riemann solution, inherently struggled to perfectly resolve instantaneous contact discontinuities. This is attributed to the numerical diffusion introduced by the Godunov method's cell-averaging approach, which smooths out sharp gradients.
2. **Rarefaction Waves:**
    - For rarefaction waves, the Exact and HLLC solvers exhibited good agreement with the analytical solution, showing only minor skewing due to inherent numerical diffusion.
    - The Roe-Pike solver, however, displayed a more pronounced and problematic skewing. This behavior stems from its linearization approach, which, while effective for shock discontinuities, struggles to accurately represent continuous expansions, even leading to unphysical discontinuities that violate the entropy condition, as observed in the analysis.
3. **Shock Waves:** All three solvers demonstrated strong agreement with the exact solution for shock waves.

Regarding computational cost, a clear hierarchy was observed: the HLLC solver proved to be the most efficient, with the Roe-Pike solver a close second, and the Exact solver being significantly less efficient. The Exact solver's computational burden increased steeply with higher spatial resolution, taking over three times longer than the HLLC solver at the highest resolution. This is a direct consequence of its rigorous and iterative approach to solving the Riemann problem at each cell interface. While both HLLC and Roe-Pike solvers showed comparable efficiency and were considerably faster than the Exact solver, the HLLC solver consistently maintained a slight speed

advantage. This difference is likely due to the Roe-Pike method requiring more complex computations per cell, specifically the calculation of eigenvalues, eigenvectors, and wave strengths, compared to the HLLC solver's more streamlined computations of pressure and wave estimates.

In summary, the custom-built code accurately implements the chosen numerical solvers. While the Exact solver provides the most accurate results, its high computational cost makes it less practical for high-resolution simulations. The HLLC solver stands out as the most suitable solver for this simulation, delivering solid accuracy while maintaining excellent computational efficiency. The Roe-Pike solver also provides good efficiency and sharp shock capturing, but its limitations in accurately resolving rarefaction waves must be considered. The choice of solver thus involves a trade-off between the level of detail required for different wave types and the available computational resources.

Appendix

References:

[1] Toro, E. F. (2013). *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer.

[2] automeris.io/wpd/

The code used to create the simulations in this report:

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import newton
import csv
import os
import time

################## Global functions and variables ##################

# Constants
gamma = 1.4  # Ratio of specific heats for air
N=100
x_min=0
x_max=1

# Define the initial conditions for the Riemann problem
def inital_conditions(case):
    if case==1:
        rho_L = 1.0  # Density on the left
        p_L = 1.0  # Pressure on the left
        u_L = 0.75  # Velocity on the left

        rho_R = 0.125  # Density on the right
        p_R = 0.1  # Pressure on the right
        u_R = 0.0  # Velocity on the right

        t_max=0.2
        x0=0.3 # Point of discontinuity

    else: #case==4
        rho_L = 5.99924  # Density on the left
        p_L = 460.894  # Pressure on the left
        u_L = 19.5975  # Velocity on the left

        rho_R = 5.99242  # Density on the right
        p_R = 46.0950  # Pressure on the right
        u_R = -6.19633 # Velocity on the right

        t_max=0.035
        x0=0.4 # Point of discontinuity

    return rho_L, p_L, u_L, rho_R, p_R, u_R, t_max, x0

# Compute conserved variables U = [rho, rho*u, E]
def primitive_to_conserved(rho, u, p):
    E = p / (gamma - 1) + 0.5 * rho * u**2
    return np.array([rho, rho * u, E])

# Compute primitive variables rho, u, p
def conserved_to_primitive(U):
    rho = U[0]
    u = U[1] / rho
    p = (gamma - 1) * (U[2] - 0.5 * rho * u**2)
    return rho, u, p
```

```
# Compute the sound speed a
def compute_a_sound_speed(U):
    rho, _, p = conserved_to_primitive(U)
    a= np.sqrt(gamma * p / rho)
    return a

# Estimate the maximum speed of the wave
def Max_Speed(U,i):
    a = compute_a_sound_speed(U)
    _, u, _=conserved_to_primitive(U)
    return np.abs(u) + a

# Compute the flux F(U)
def flux_from_U(U):
    rho, u, p = conserved_to_primitive(U)
    F = np.array([
        U[1],
        (rho * u**2) + p,
        u * (U[2] + p)
    ])
    return F

################## Functions for the exact solution ##################
# Compute a helpul parameter W
def compute_W(x_t, a_L, a_R, I, S, SH, ST,rho_L, p_L, u_L, rho_R, p_R, u_R,
state):

    p_star=compute_p_star(a_L, a_R, p_L, u_L, rho_L, p_R, u_R, rho_R)
    u_star=compute_u_star(p_star, a_L, a_R, rho_L, p_L, u_L, rho_R, p_R, u_R)
    rho_star=compute_rho_star(p_star, rho_L, p_L, rho_R, p_R, I)

    if x_t < u_star:
        if state=="shock":
            if x_t < S:
                W= np.array([rho_L, u_L, p_L])

            else: # x_t > S
                W=np.array([rho_star, u_star, p_star])

        else: # state=="rarefaction"
            if x_t < SH:
                W= np.array([rho_L, u_L, p_L])
            elif SH < x_t < ST:
                W=compute_W_fan(x_t, a_L, rho_L, p_L, u_L, rho_R, p_R, u_R, "L")
            else: # ST < x_t < u_star
                W= np.array([rho_star, u_star, p_star])

    else: # x_t > u_star
        if state=="shock":
            if x_t > S:
                W= np.array([rho_R, u_R, p_R])

            else: # x_t < S
                W=np.array([rho_star, u_star, p_star])

        else: # state=="rarefaction"
            if x_t > SH:
                W= np.array([rho_R, u_R, p_R])
            elif ST < x_t < SH:
                W=compute_W_fan(x_t, a_R, rho_L, p_L, u_L, rho_R, p_R, u_R, "R")
            else: # u_star < x_t < ST
                W= np.array([rho_star, u_star, p_star])
```

```
    return W

# Compute W_fan
def compute_W_fan(x_t, a, rho_L, p_L, u_L, rho_R, p_R, u_R, I):
    if I=="L":
        a_L=a
        rho= rho_L*((2/(gamma+1)) + ((gamma-1)/((gamma+1)*a_L))*(u_L-
x_t))**(2/(gamma-1))
        u= 2/(gamma+1) * (a_L + (gamma-1)/2 * u_L + x_t)
        p= p_L * (2/(gamma+1) + ((gamma-1)/((gamma+1)*a_L))*(u_L-
x_t))**(2*gamma/(gamma-1))

    else: # I=="R"
        a_R=a
        rho= rho_R*((2/(gamma+1)) - ((gamma-1)/((gamma+1)*a_R))*(u_R-
x_t))**(2/(gamma-1))
        u= 2/(gamma+1) * (-a_R + (gamma-1)/2 * u_R + x_t)
        p= p_R * (2/(gamma+1) - ((gamma-1)/((gamma+1)*a_R))*(u_R-
x_t))**(2*gamma/(gamma-1))

    return np.array([rho, u, p])

# Is the wave a shock or a rarefaction wave?
def is_shock_or_rarefaction_wave(p, p_I, I):
    if I=="L":
        p_L=p_I
        if p>p_L:
            return "shock"
        else: # p<p_L
            return "rarefaction"

    else: # I=="R"
        p_R=p_I
        if  p>p_R:
            return "shock"
        else: # p<p_R
            return "rarefaction"

# Compute a helpul parameter A
def compute_A(rho_i):
    A=2/((gamma+1)*rho_i)
    return A

# Compute a helpul parameter B
def compute_B(p_i):
    B=((gamma-1)/(gamma+1))*p_i
    return B

# Compute the funcfion f whos root is the solution for p_star
def compute_f_and_df(p, a, rho_L, p_L, rho_R, p_R, I):
    if I=="L":
        a_L=a
        A_L=compute_A(rho_L)
        B_L=compute_B(p_L)
        state=is_shock_or_rarefaction_wave(p, p_L, "L")
        if state=="shock": # shock wave
            f=(p-p_L)*(A_L/(p+B_L))**0.5
            df=(A_L/(p+B_L))**0.5 * (1 - (p-p_L)/(2*(p+B_L)))

        else: #  rarefaction wave
            f=(2*a_L/(gamma-1))*(((p/p_L)**((gamma-1)/2/gamma)) -1)
            df=1/(rho_L*a_L) * (p/p_L)**(-(gamma+1)/(2*gamma))

    else: # I=="R"
        a_R=a
```

```
        A_R=compute_A(rho_R)
        B_R=compute_B(p_R)
        state=is_shock_or_rarefaction_wave(p, p_R, "R")

        if state=="shock": # shock wave
            f=(p-p_R)*(A_R/(p+B_R))**0.5
            df=(A_R/(p+B_R))**0.5 * (1 - (p-p_R)/(2*(p+B_R)))

        else: # rarefaction wave
            f=(2*a_R/(gamma-1))*((p/p_R)**(((gamma-1)/2/gamma)) -1)
            df=1/(rho_R*a_R) * (p/p_R)**(-(gamma+1)/(2*gamma))

    return f, df

# Compute u_star - the velocity of the contact discontinuity
def compute_u_star(p_star, a_L, a_R, rho_L, p_L, u_L, rho_R, p_R, u_R):
    f_R,_=compute_f_and_df(p_star, a_R, rho_L, p_L, rho_R, p_R, "R")
    f_L,_=compute_f_and_df(p_star, a_L, rho_L, p_L, rho_R, p_R, "L")
    u_star=0.5*(u_L + u_R) + 0.5*(f_R - f_L)
    return u_star

# Compute p_star exact - the pressure at the contact discontinuity
def compute_p_star(a_L, a_R, p_L, u_L, rho_L, p_R, u_R, rho_R):
    func = lambda p: compute_f_and_df(p, a_L, rho_L, p_L, rho_R, p_R, "L")[0] +
compute_f_and_df(p, a_R, rho_L, p_L, rho_R, p_R, "R")[0]  + u_R - u_L # Extracts
f
    dfunc = lambda p: compute_f_and_df(p, a_L, rho_L, p_L, rho_R, p_R, "L")[1] +
compute_f_and_df(p, a_R, rho_L, p_L, rho_R, p_R, "R")[1] # Extracts df
    p_guess = (p_L + p_R) / 2
    p_star = newton(func, x0=p_guess, fprime=dfunc, tol=1e-6, maxiter=1000)
    #print(f"p_star: {p_star:.6f}")
    return p_star

# Compute rho_star - the density at the contact discontinuity
def compute_rho_star(p_star,rho_L, p_L, rho_R, p_R, I):
    if I=="L":
        state= is_shock_or_rarefaction_wave(p_star, p_L, "L")
        if state=="shock":
            rho_star=rho_L*(((p_star/p_L) + ((gamma-
1)/(gamma+1)))/((p_star/p_L)*((gamma-1)/(gamma+1)) + 1))
        else: # rarefaction wave
            rho_star=rho_L*((p_star/p_L)**(1/gamma))

    else: # I=="R"
        state= is_shock_or_rarefaction_wave(p_star, p_R, "R")
        if state=="shock":
            rho_star=rho_R*(((p_star/p_R) + ((gamma-
1)/(gamma+1)))/((p_star/p_R)*((gamma-1)/(gamma+1)) + 1))
        else: #rarefaction wave
            rho_star=rho_R*((p_star/p_R)**(1/gamma))

    return rho_star

# Compute speed of wave front S
def compute_S(p_star, a, u_star, p_L, u_L, p_R, u_R, I):
    if I=="L":
        a_L=a
        state= is_shock_or_rarefaction_wave(p_star, p_L, "L")
        if state=="shock":
            S=u_L - a_L*(((gamma+1)/(2*gamma))*(p_star/p_L) + ((gamma-
1)/(2*gamma)))**0.5
            SH, ST= None, None

        else: #rarefaction wave
            a_star=a_L*((p_star/p_L)**((gamma-1)/(2*gamma)))
```

```
            SH=u_L - a_L
            ST=u_star - a_star
            S= None

    else: # I=="R"
        a_R=a
        state= is_shock_or_rarefaction_wave(p_star, p_R, "R")
        if state=="shock":
            S=u_R + a_R*(((gamma+1)/(2*gamma))*(p_star/p_R) + ((gamma-
1)/(2*gamma)))**0.5
            SH, ST= None, None

        else: #rarefaction wave
            a_star=a_R*((p_star/p_L)**((gamma-1)/(2*gamma)))
            SH=u_R + a_R
            ST=u_star + a_star
            S= None

    return S, SH, ST, state

################## Functions for HLLC solver ##################

# Compute the pressure estimate at the contact discontinuity for HLLC
def p_star_estimate(rho_L, p_L, u_L, a_L, rho_R, p_R, u_R, a_R):
    rho_avg=(rho_L + rho_R)/2
    a_avg=(a_L + a_R)/2
    p_avg=(p_L + p_R)/2
    p_pvrs=p_avg - 0.5*(u_R - u_L)*rho_avg*a_avg
    p_star=max(0, p_pvrs)
    return p_star

# Compute q for HLLC
def compute_q(p_star, p):

    if p_star<=p:
        q=1
    else: # p_star>p
        q=(1 + (((gamma+1)/(2*gamma))*(p_star/p -1)))**0.5

    return q

# Compute the wave speeds estimates for HLLC
def wave_speed_estimates(p_L, u_L, a_L, q_L, p_R, u_R, a_R, q_R):

    S_L = u_L - a_L*q_L
    S_R = u_R + a_R*q_R
    S_star=(p_R-p_L+rho_L*u_L*(S_L-u_L)- rho_R*u_R*(S_R-u_R))/(rho_L*(S_L-u_L)-
rho_R*(S_R-u_R))

    return S_L, S_star, S_R

# Compute the flux in star riegon for HLLC
def F_star(U, U_star, S):
    F_star=flux_from_U(U) + S*(U_star - U)
    return F_star

# Compute the flux for HLLC
def compute_flux_for_HLLC(U_L, U_star_L, U_R, U_star_R, S_L, S_star, S_R):

    if S_L >= 0:
        F_L=flux_from_U(U_L)
        F_HLLC=F_L
    elif S_star >= 0:
        F_L_star=F_star(U_L, U_star_L, S_L)
        F_HLLC=F_L_star
```

```
    elif S_R >= 0:
        F_R_star=F_star(U_R, U_star_R, S_R)
        F_HLLC=F_R_star
    else: # S_R < 0:
        F_R=flux_from_U(U_R)
        F_HLLC=F_R

    return F_HLLC

# Compute U in star riegon for HLLC
def compute_U_star(rho, p, u, S, S_star, U_I):
    E=U_I[2]
    scalar=rho*((S-u)/(S-S_star))
    U_star=scalar * np.array([
        1,
        S_star,
        E/rho + (S_star-u)*(S_star + (p/(rho*(S-u))))
    ])
    return U_star

################## Functions for Roe-pike solver ##################

# Compute enthalpy H for Roe-Pike
def compute_H(rho, p, u):
    U= primitive_to_conserved(rho, u, p)
    H = (U[2] + p) / rho
    return H

# Compute avrage values for Roe-Pike
def compute_avg_values(rho_L, p_L, u_L, rho_R, p_R, u_R):
    rho_avg = (rho_L*rho_R)**0.5
    u_avg = ((rho_L**0.5 * u_L) + (rho_R**0.5 * u_R)) / (rho_L**0.5 +
rho_R**0.5)
    H_L = compute_H(rho_L, p_L, u_L)
    H_R = compute_H(rho_R, p_R, u_R)
    H_avg = ((rho_L**0.5 * H_L) + (rho_R**0.5 * H_R)) / (rho_L**0.5 +
rho_R**0.5)
    a_avg = ((gamma-1)*(H_avg - 0.5*(u_avg**2)))**0.5
    return rho_avg, u_avg, H_avg, a_avg

# Compute the eigenvalues for Roe-Pike
def compute_lambda(u_avg, a_avg):
    lambda_vector = np.zeros(3)
    lambda_vector[0] = u_avg - a_avg
    lambda_vector[1] = u_avg
    lambda_vector[2] = u_avg + a_avg
    return lambda_vector

# Compute the right eigenvectors for Roe-Pike
def compute_K(u_avg, H_avg, a_avg):
    K_vector_1 = np.zeros(3)
    K_vector_1[0] = 1
    K_vector_1[1] = u_avg - a_avg
    K_vector_1[2] = H_avg - (u_avg * a_avg)

    K_vector_2 = np.zeros(3)
    K_vector_2[0] = 1
    K_vector_2[1] = u_avg
    K_vector_2[2] = 0.5 * (u_avg**2)

    K_vector_3 = np.zeros(3)
    K_vector_3[0] = 1
    K_vector_3[1] = u_avg + a_avg
    K_vector_3[2] = H_avg + (u_avg * a_avg)
```

```
    K_matrix = np.array([K_vector_1, K_vector_2, K_vector_3])

    return K_matrix

# Compute alpha for Roe-Pike
def compute_alpha(rho_L, p_L, u_L, rho_R, p_R, u_R, rho_avg, a_avg):
    alpha_1 = ((p_R - p_L) - (rho_avg * a_avg * (u_R - u_L))) / (2 * a_avg**2)
    alpha_2 = (rho_R - rho_L) - ((p_R - p_L) / a_avg**2)
    alpha_3 = ((p_R - p_L) + (rho_avg * a_avg * (u_R - u_L))) / (2 * a_avg**2)

    alpha_vector = np.array([alpha_1, alpha_2, alpha_3])

    return alpha_vector

################## Solvers for the Riemann problem ##################

# Exact solver for the Riemann problem
def solve_exact(rho_L, p_L, u_L, rho_R, p_R, u_R, t, x0):

    x = np.linspace(x_min, x_max, N)  # Spatial grid
    U_exact = np.zeros((N, 3))  # Initialize the array for conserved variables
    for i in range(N):
        x_i = x[i] - x0  # Centered at point of discontinuity
        if t==0:
            x_t = 0
        else:
            x_t = x_i / t_max  # Characteristic speed x/t
        U_exact[i] = Solve_Riemann_exact(rho_L, p_L, u_L, rho_R, p_R, u_R, x_t)

    return x, U_exact

# HLLC solver for the Riemann problem
def HLLC(rho_L, p_L, u_L, rho_R, p_R, u_R):
    # Compute L and R state parameters
    U_L = primitive_to_conserved(rho_L, u_L, p_L)
    U_R = primitive_to_conserved(rho_R, u_R, p_R)
    a_L = compute_a_sound_speed(U_L)
    a_R = compute_a_sound_speed(U_R)
    #print(f"U_L: {U_L} U_R: {U_R} F_L: {F_L} F_R: {F_R} a_L: {a_L} a_R: {a_R}")
    # Compute the pressure estimate at the contact discontinuity
    p_star = p_star_estimate(rho_L, p_L, u_L, a_L, rho_R, p_R, u_R, a_R)

    #print(f"p_star: {p_star}")
    # Compute paramter q
    q_L = compute_q(p_star, p_L)
    q_R = compute_q(p_star, p_R)
    #print(f"q_L: {q_L} q_R: {q_R}")
    # Compute the wave speeds estimates
    S_L, S_star, S_R = wave_speed_estimates(p_L, u_L, a_L, q_L, p_R, u_R, a_R,
q_R)
    #print(f"S_L: {S_L} S_star: {S_star} S_R: {S_R}")
    # Compute U_star
    U_star_L = compute_U_star(rho_L, p_L, u_L, S_L, S_star, U_L)
    U_star_R = compute_U_star(rho_R, p_R, u_R, S_R, S_star, U_R)
    #print(f"U_star_L: {U_star_L} U_star_R: {U_star_R}")
    # Compute the fluxes
    F_HLLC = compute_flux_for_HLLC(U_L, U_star_L, U_R, U_star_R, S_L, S_star,
S_R)
    #print(f"F_HLLC: {F_HLLC}")
    return F_HLLC

# Roe-Pike solver for the Riemann problem
def Roe_Pike(rho_L, p_L, u_L, rho_R, p_R, u_R):
    U_L= primitive_to_conserved(rho_L, u_L, p_L)
    U_R= primitive_to_conserved(rho_R, u_R, p_R)
```

```
    rho_avg, u_avg, H_avg, a_avg = compute_avg_values(rho_L, p_L, u_L, rho_R,
p_R, u_R)
    lambda_vector = compute_lambda(u_avg, a_avg)
    K_matrix = compute_K(u_avg, H_avg, a_avg)
    alpha_vector = compute_alpha(rho_L, p_L, u_L, rho_R, p_R, u_R, rho_avg,
a_avg)

    F_avg= 0.5*(flux_from_U(U_L) + flux_from_U(U_R))
    for i in range(3):
        F_avg -= 0.5 * alpha_vector[i] * abs(lambda_vector[i]) * K_matrix[i]

    return F_avg

################## Main simulation function ##################

# Exact solution for the Riemann problem
def Solve_Riemann_exact(rho_L, p_L, u_L, rho_R, p_R, u_R, x_t):

    U_L = primitive_to_conserved(rho_L, u_L, p_L)
    U_R = primitive_to_conserved(rho_R, u_R, p_R)
    a_L = compute_a_sound_speed(U_L)
    a_R = compute_a_sound_speed(U_R)

    # Solve for p_star, u_star, and rho_star
    p_star = compute_p_star(a_L, a_R, p_L, u_L, rho_L, p_R, u_R, rho_R)
    u_star = compute_u_star(p_star, a_L, a_R, rho_L, p_L, u_L, rho_R, p_R, u_R)

    S_L, SH_L, ST_L, state_L = compute_S(p_star, a_L, u_star, p_L, u_L, p_R,
u_R, "L")
    S_R, SH_R, ST_R, state_R = compute_S(p_star, a_R, u_star, p_L, u_L, p_R,
u_R, "R")


    if x_t < u_star:  # Left region
        W = compute_W(x_t, a_L, a_R, "L", S_L, SH_L, ST_L, rho_L, p_L, u_L,
rho_R, p_R, u_R, state_L)
    else:  # Right region
        W = compute_W(x_t, a_L, a_R, "R", S_R, SH_R, ST_R, rho_L, p_L, u_L,
rho_R, p_R, u_R, state_R)

    rho, u, p = W

    # Convert to conserved variables
    U = primitive_to_conserved(rho, u, p)

    return U

#Simulate using Godunov method
def Gondunov_method(case, solver):
    start_time = time.time()

    # Define starting time and counter
    t = 0
    counter = 0

    # Define grid step size
    dx=(x_max-x_min)/N
    x_half= np.linspace(x_min+dx/2, x_max-dx/2, N)
    x= np.linspace(x_min, x_max, N)
    #initalize conditions
    rho_L, p_L, u_L, rho_R, p_R, u_R, t_max, x0 = inital_conditions(case)

    # Initalize U_array
    U_L = primitive_to_conserved(rho_L, u_L, p_L)
    U_R = primitive_to_conserved(rho_R, u_R, p_R)
```

```
    U_array = np.zeros((N, 3))  # Initialize the array for conserved variables
    F_array = np.zeros((N+1, 3))
    for i in range(N):
        x_i = x[i]
        if x_i < x0:
            U_array[i] = U_L
        else:
            U_array[i] = U_R
    #print(f"U_array: {U_array}")

    # Main loop
    while t <= t_max:
        if counter <5:
            CFL=0.9*0.2
        else:
            CFL=0.9

        max_speed_i = np.max([Max_Speed(U_array[i],i) for i in range(N)])

        dt = CFL* dx / max_speed_i  # Time step size

        U_new= np.zeros_like(U_array) #create a new array to store the updated
values

        for i in range(1, N):

            rho_L, u_L, p_L = conserved_to_primitive(U_array[i-1])
            rho_R, u_R, p_R = conserved_to_primitive(U_array[i])

            if solver=="Exact":
                U = Solve_Riemann_exact(rho_L, p_L, u_L, rho_R, p_R, u_R, 0)
#x_t=0
                F_array[i] = flux_from_U(U)

            elif solver=="HLLC":
                F_array[i] = HLLC(rho_L, p_L, u_L, rho_R, p_R, u_R)

            elif solver=="Roe-Pike":
                F_array[i] = Roe_Pike(rho_L, p_L, u_L, rho_R, p_R, u_R)

        U_new = U_array - (dt/dx) * (F_array[1:] - F_array[:-1])
        U_new[0] = U_array[0]
        U_new[-1] = U_array[-1]

        # Update the solution
        U_array=U_new.copy()

        # Update time
        t += dt
        if t + dt > t_max:
            dt = t_max - t

        counter += 1

        # print(f"Time step {counter}: t = {t:.6f}")
        # print(f"F_array: {F_array}")
    end_time = time.time()
    print(f"Time taken for case {case} with {solver} solver: {end_time -
start_time:.5f} seconds")
    return x_half, U_array

################## Visualization functions ##################

# Extract data from torro files
def Extract_torro_data(case, variables, solver):
```

```
    torro_data = {}

    if solver=="Exact Riemann solution":
        for var in variables:
            # Read exact data
            exact_file =
os.path.join('torro_exact_solution',f'torro_case_{case}_{var}_exact.csv')
            x_values, y_values = [], []
            with open(exact_file, 'r', newline='') as csvfile:
                reader = csv.reader(csvfile)
                for row in reader:
                    if row:
                        x_values.append(float(row[0]))
                        y_values.append(float(row[1]))
            torro_data[var] = (x_values, y_values)


    else:
        directory_path=f'torro_{solver}_solver'
        for var in variables:
            # Read numerical data
            num_file =
os.path.join(directory_path,f'torro_case_{case}_{var}_numerical.csv')
            x_values, y_values = [], []
            with open(num_file, 'r', newline='') as csvfile:
                reader = csv.reader(csvfile)
                for row in reader:
                    if row:
                        x_values.append(float(row[0]))
                        y_values.append(float(row[1]))

            torro_data[var] = (x_values, y_values)

    return torro_data

# Calculate internal energy and set up data dictionary
def calculate_e_and_set_up_data_dictionary(U_array):
    # Convert to primitives in vectorized fashion
    rho, u, p = np.array([conserved_to_primitive(U) for U in U_array]).T

    # Compute internal energy
    e = p / ((gamma - 1) * rho)

    # Package into dictionaries
    data = {
        "density": rho,
        "velocity": u,
        "pressure": p,
        "energy": e
    }

    return data

# Plot results for each case and solver
def plot_results_per_case_and_solver(x, x_half, torro_data_exact_solution,
all_data_dic, case, variables, solver):

    plt.figure(figsize=(14, 12))

    torro_data=Extract_torro_data(case, variables, solver)
    plt.title(f"Case {case} - Riemann slover - {solver}")
    plt.axis("off")

    for i, var in enumerate(variables):
        plt.subplot(2, 2, i+1)
```

```
        plt.plot(x_half, all_data_dic[case][solver][var], label=f"{solver}
solver", marker='o', markersize=2, linestyle='')
        plt.plot(x, all_data_dic[case]["Exact Riemann solution"][var],
label="Exact Riemann solution")
        plt.plot(torro_data[var][0], torro_data[var][1], label="Torro
numerical", marker='o', markersize=2, linestyle='')
        plt.plot(torro_data_exact_solution[var][0],
torro_data_exact_solution[var][1], label="Torro exact", linestyle='--')
        plt.xlabel("x")
        plt.ylabel(f"{var}")
        plt.legend()

    #plt.show()
    plt.savefig(f"case_{case}_{solver}.png")

# Plot results for all solvers
def plot_results_per_case_all_solvers(x, all_data_dic, case, variables):
    plt.figure(figsize=(14, 12))
    plt.title(f"Case {case} - Numerical Solver Comparison")
    plt.axis("off")

    for i, var in enumerate(variables):
        plt.subplot(2, 2, i+1)

        # Plot all solvers
        for j, solver in enumerate(all_data_dic[case]):
            if solver != "Exact Riemann solution":
                plt.plot(x, all_data_dic[case][solver][var], label=f"{solver}
solver", marker='o', markersize=3 - 0.5*j, linestyle='')
            else:
                plt.plot(x, all_data_dic[case][solver][var], label="Exact
Riemann solution")

        plt.xlabel("x")
        plt.ylabel(f"{var}")
        plt.legend()

    #plt.show()
    plt.savefig(f"case_{case}_solver_comparison.png")

################## Run the simulation ##################

cases=[1,4]
variables = ["density", "velocity", "pressure", "energy"]
solvers=["Exact", "HLLC", "Roe-Pike"]
all_data_dic={}

for case in cases:
    all_data_dic[case] = {}
    rho_L, p_L, u_L, rho_R, p_R, u_R, t_max, x0 = inital_conditions(case)
    print(f"intialized case {case}")
    x, U_exact = solve_exact(rho_L, p_L, u_L, rho_R, p_R, u_R, t_max, x0)
    all_data_dic[case]["Exact Riemann solution"] =
calculate_e_and_set_up_data_dictionary( U_exact)
    print("computed exact solution")
    torro_Exact_Riemann_solution=Extract_torro_data(case, variables,"Exact
Riemann solution")
    print("extracted torro exact solution")

    for solver in solvers:
        x_half, U_numerical = Gondunov_method(case,solver)
        print(f"computed numerical solution using {solver} solver")
        all_data_dic[case][solver] =
calculate_e_and_set_up_data_dictionary(U_numerical)
        plot_results_per_case_and_solver(x, x_half,
```

```
torro_Exact_Riemann_solution, all_data_dic, case, variables, solver)
        print(f"plotted case {case} with {solver} solver")

for case in cases:
    plot_results_per_case_all_solvers(x, all_data_dic, case, variables)
    print(f"plotted case {case} with all solvers")
```