Numerical Fluid Dynamics

Question 1*: A functional fluid solver*

This exercise involved writing a C code that solves Euler's equations to reproduce exact shock tube test results. The figures below detail the results of the shock tube using the Lax-Friedrichs method (blue), HLL method (red) and exact solutions for comparison (black).

Problem A

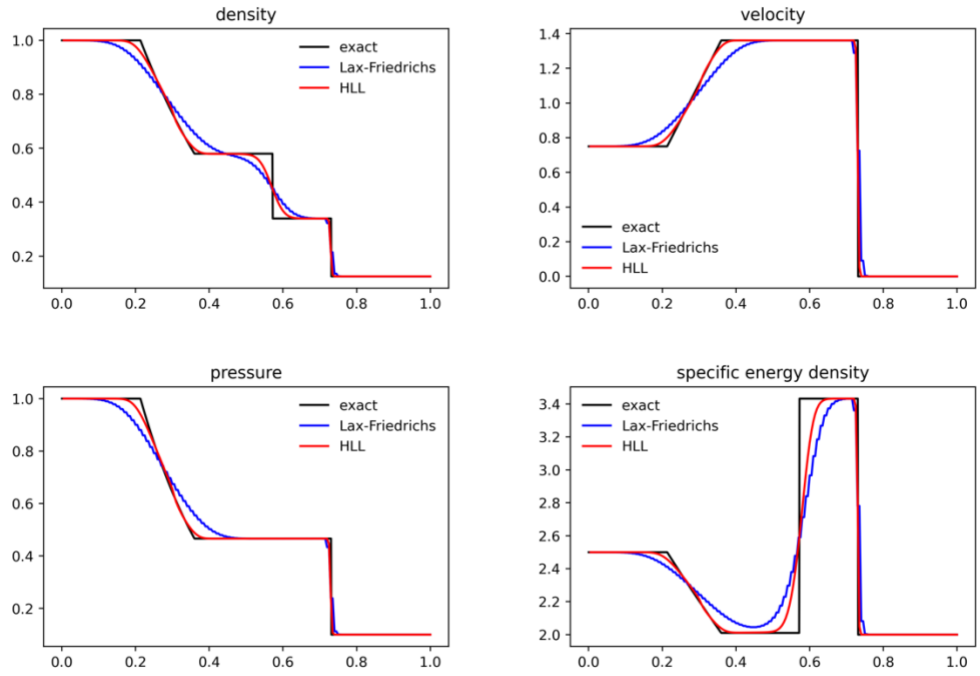

Figure 1: Standard shock tube test for Lax-Friedrichs and HLL methods. The initial conditions are that the left state ($x < 0.3$ at $t = 0$) has $\rho = 1$, $p = 1$ and $v = 0.75$, whereas the right state has $\rho = 0.125$, $p = 0.1$ and $v = 0$. Both solvers use 200 zones (not counting ghost zones) between $x = 0...1$. The snapshots are taken at $t = 0.2$.

Problem B

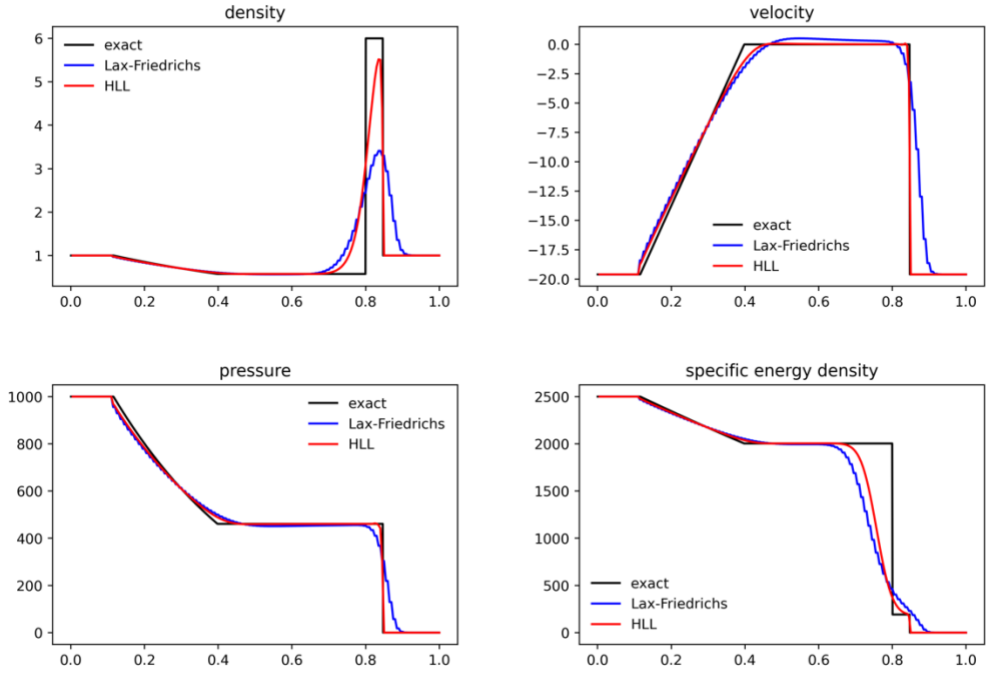

Figure 2: Standard shock tube test for HLL and LF methods. The initial conditions are that the left state (x < 0.8 at t = 0) has ρ = 1, p = 1000.0 and v = −19.59745, whereas the right state has ρ = 1., p = 0.01 and v = −19.59745. Both solvers use 200 zones (not counting ghost zones) between x = 0...1. The snapshots are taken at t = 0.012.

Question 2*: Colliding atomic gas clouds in the interstellar medium*

This exercise involved applying the chock tube model to an astrophysical problem modelling two clouds. Initial conditions are the left state (x <= 5 at t = 0) has rho = 10, p = 20631 and v = 50, whereas the right state has rho = 30, p = 618.93 and v = -50. Snapshots are taken when the impact of the collision are felt a box size of 5 away.

2 A

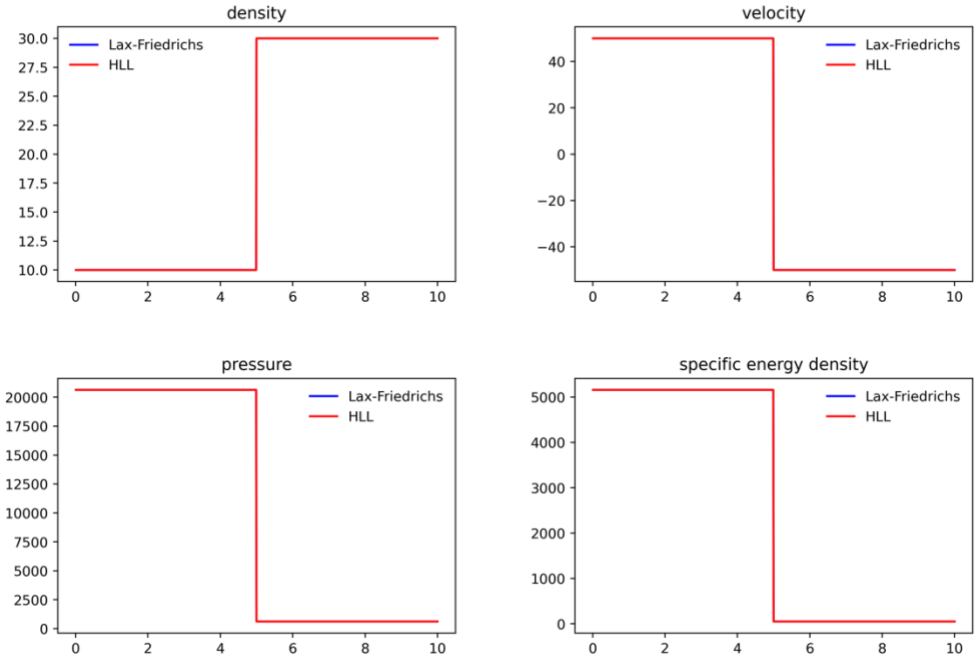

Figure 3: Shock tube test for HLL and LF methods to model colliding clouds. Initial conditions are the left state (x <= 5 at t = 0) has rho = 10, p = 20631 and v = 50, whereas the right state has rho = 30, p = 618.93 and v = -50. Snapshots are taken when the impact of the collision are felt a box size of 5 away. Both solvers use 1000 zones (not counting ghost zones) between x = 0...1, scaled to 0…10. The snapshots are taken at t = 0.

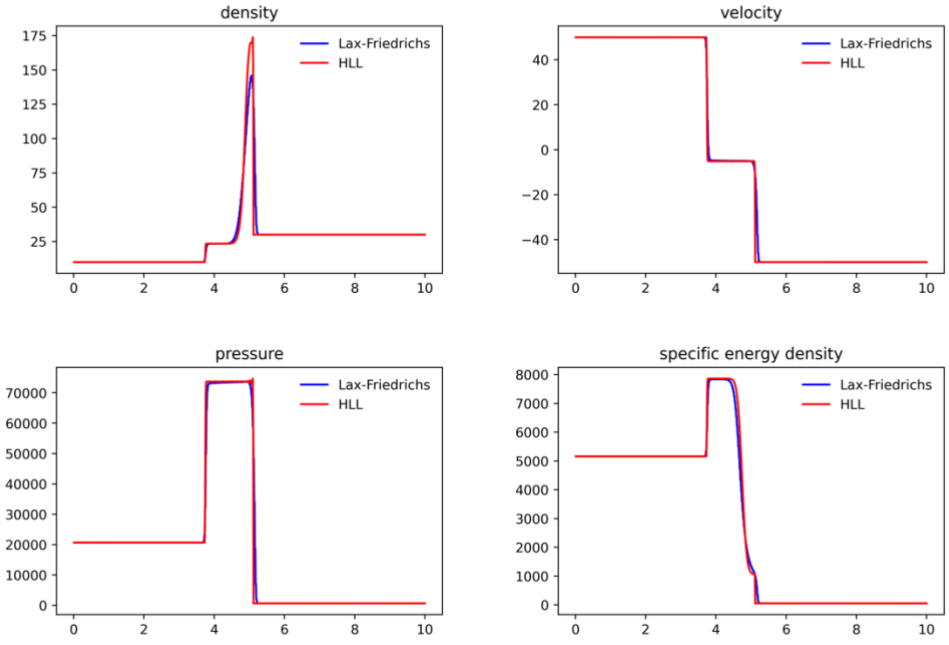

Figure 4: Standard shock tube from problem 2 A. Both solvers use 1000 zones (not counting ghost zones) between x = 0...1, scaled to 0…10. The snapshots are taken at t = .010767/4 s.

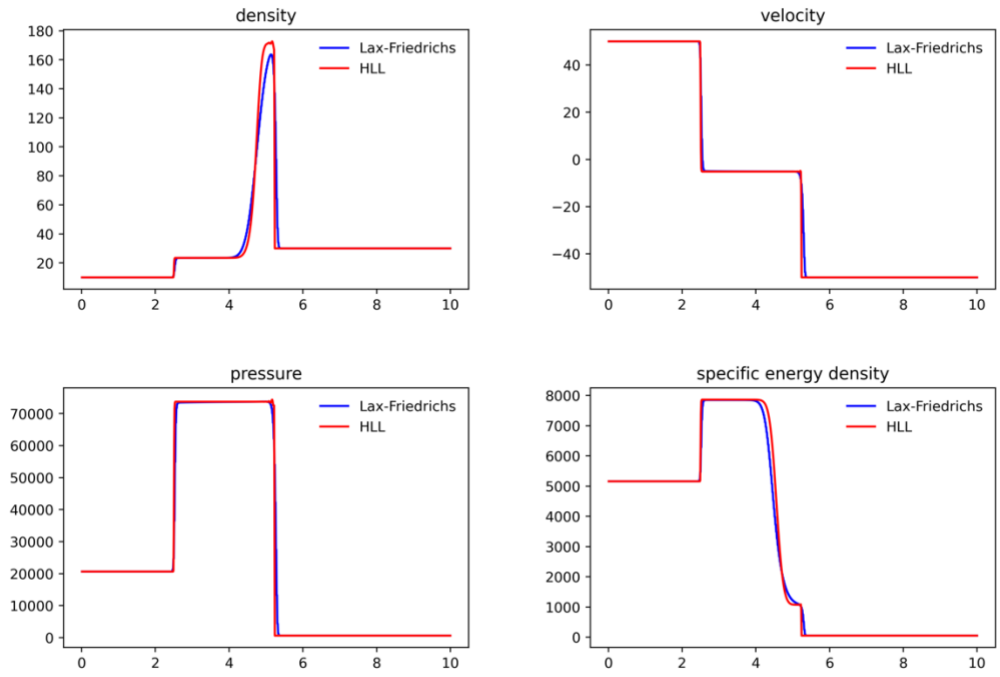

Figure 5: Standard shock tube from problem 2 A. Both solvers use 1000 zones (not counting ghost zones) between x = 0...1, scaled to 0…10. The snapshots are taken at t = 0.010767/2.

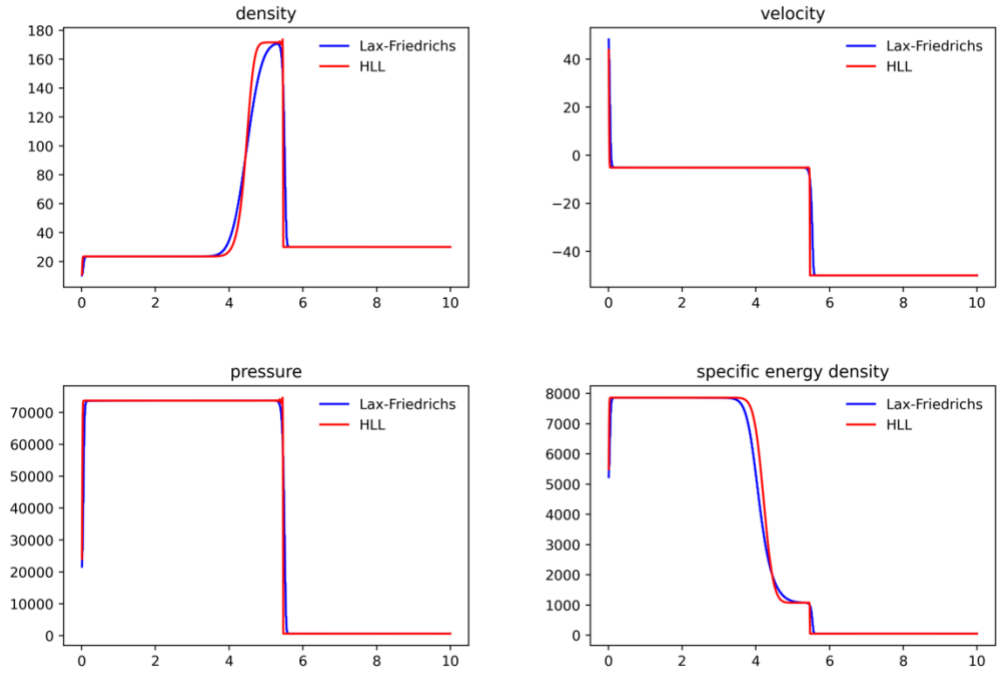

Figure 6: Standard shock tube from problem 2 A. Both solvers use 1000 zones (not counting ghost zones) between x = 0...1, scaled to 0…10. The snapshots are taken at t = 0.010767.

2 B

Figures 3 – 6 depict the evolution of the clouds over time. When the clouds collide, the shock propagates to the left. The initial contact discontinuity also shifts to the left. The rarefaction wave travels to the right where there is a low area of pressure.

3

Approach to the problem

- The problem was to solve Euler's equations in one dimension using numerical methods. The numerical methods I looked at were finite volume methods, more specifically the Lax-Friedrich (LF) scheme and the HLL scheme.

Choice of solver

- The LF scheme is a numerical scheme based on finite differences. The LF scheme is first order accurate, so it is only stable if the following condition is met,

$$\left| a \frac{\Delta t}{\Delta x} \right| \leq 1.$$

- At orders higher than this the LF becomes dissipative and not as accurate, for instance Figure 2. The FL struggled to model a more extreme shock tube. At large spatial resolution the LF moves towards the desired solutions, Figure 12, although a better scheme to use for such conditions is the HLL scheme. The HLL scheme is uses the intercell (grid) flux . The HLL scheme proved to be a more accurate scheme than the FL, Figure 2.

Time step

- Time step was calculated using the system's largest eigenvalue in the following equation:

$$dt = CFL * dx \, / max(|v0| + c\_s).$$

- Where v0 is the velocity and c_s is the sound speed. The timestep is dx over the maximum wave speed on the grid, which adjusts for fluctuations in the system. The CFL constant can be used to reduce timestep in case of a supersonic shock.

Choice of solver

- The LF scheme and the HLL scheme.

Workings of code

- The code is split up into several functions. The first step is setting up the shock tube grid into two regions, with pressure, density and velocity values in each region.
- Next is setting up boundary conditions using ghost cells. Ghost cells make up the first cell and the last cell, these cells decide what happens at the boundary of the system where there is only one contribution of flux. Boundary conditions that conserve quantities are periodic or reflective boundary conditions. My code uses constant boundary conditions as there were initially most convenient.
- Update the timestep dynamically (Time step), total time += dt.
- Next is to update the state vector flux values, initially the shock tube grid is assigned to these values. When the shock tube grid values are updated the state vectors are updated again using a while loop. This determines the in-cell flux.
- Next is applying one of the schemes mentioned, FL or HLL. These schemes take the in-cell flux and produce boundary flux values, between the centre of the cells (i+1/2 or i-1/2). The following equation is used to update these half flux values.

$$Q_i^{n+1} = Q_i^n - \frac{\Delta t}{\Delta x}\left(F_{i+\frac{1}{2}}^{avg} - F_{i-\frac{1}{2}}^{avg}\right).$$

- This equation gives a new state vector value (n+1). This is used to update the grid variables mentioned at the start (pressure, density and velocity). The time step increases as this marks the end of the algorithm. The loop starts again by defining boundary conditions.
- When then time condition is satisfied and the total time is met the programme exits the loop and the initial grid variables (pressure, density and velocity) are saved. to then be plotted

4

The source code is in the appendix 3.

Appendix

Question 1

Problem A

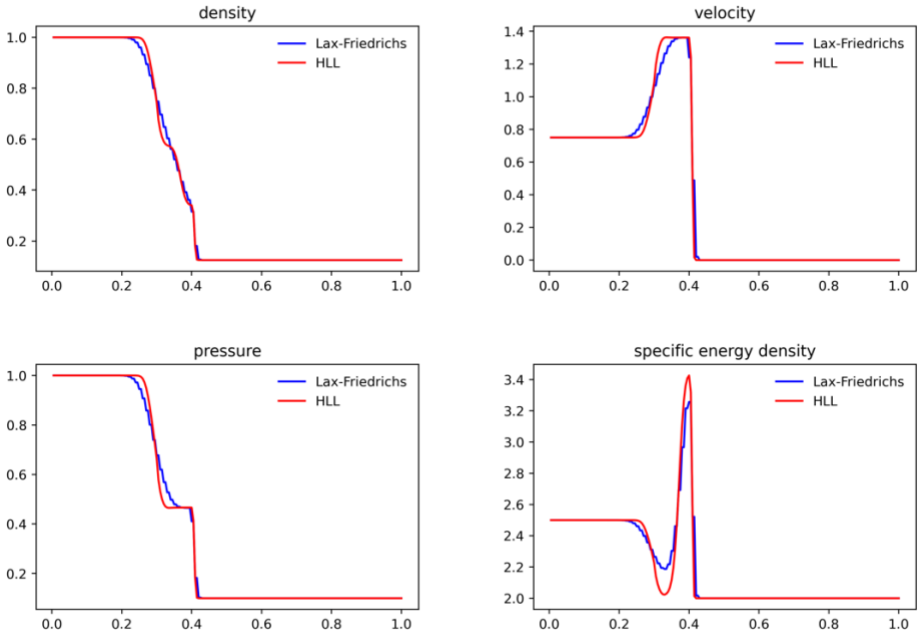

Figure 7: Standard shock tube from problem A. The snapshots are taken at t = 0.05.

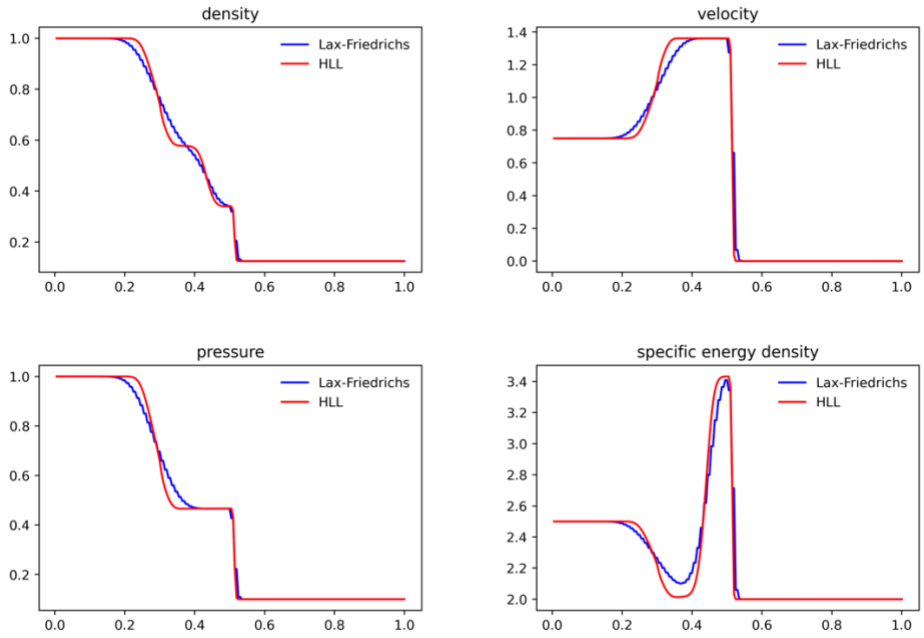

Figure 8: Standard shock tube from problem A. The snapshots are taken at t = 0.1.

Problem B

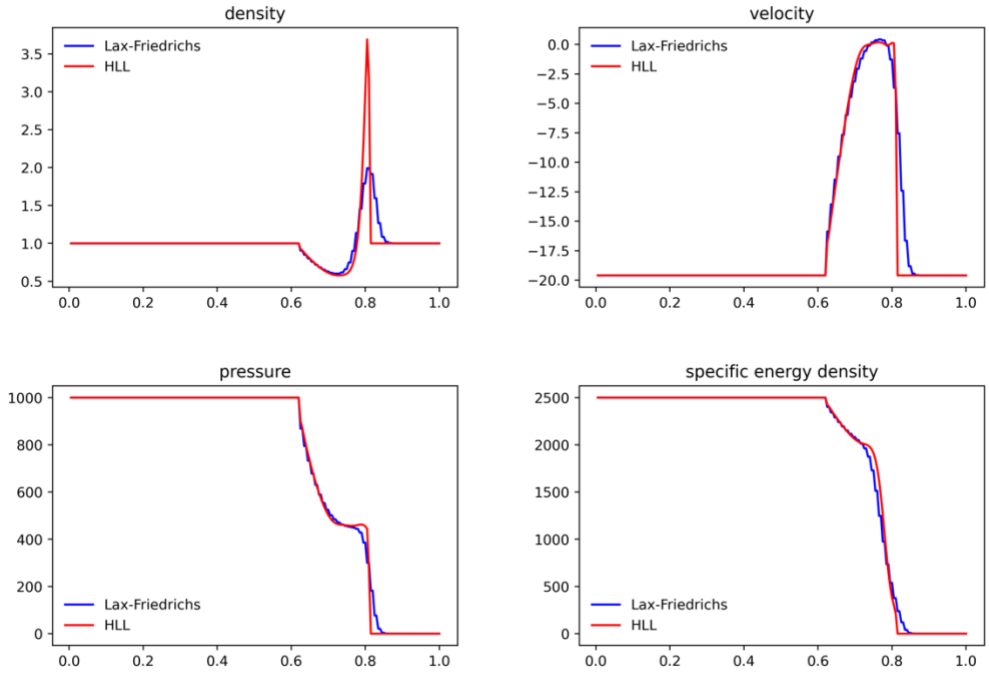

Figure 9: Standard shock tube from problem B. The snapshots are taken at t = 0.003.

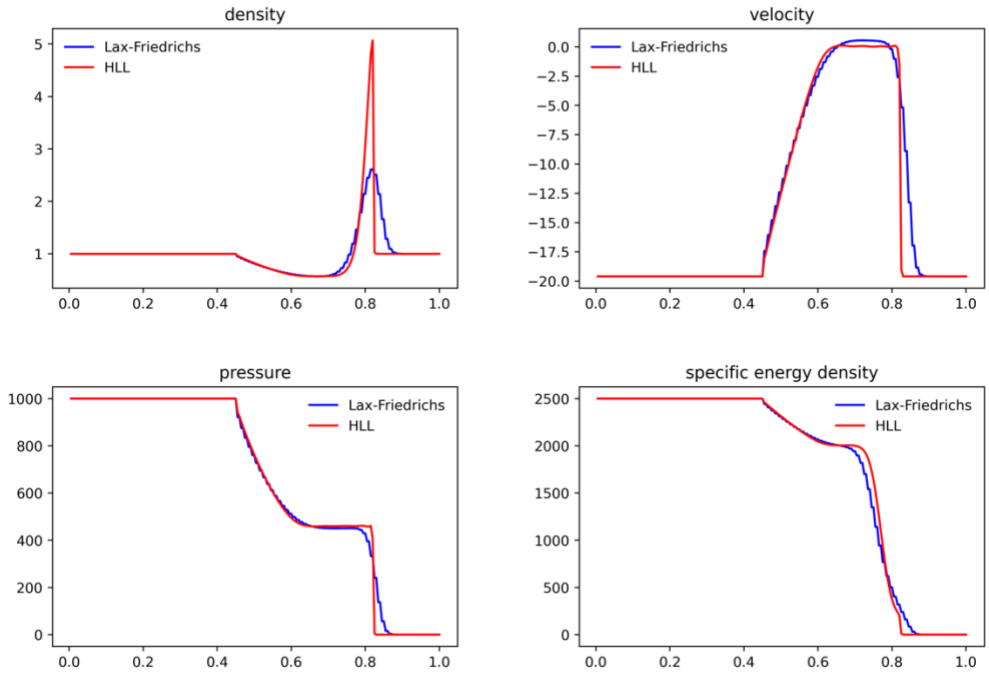

Figure 10: Standard shock tube from problem B. The snapshots are taken at t = 0.06.

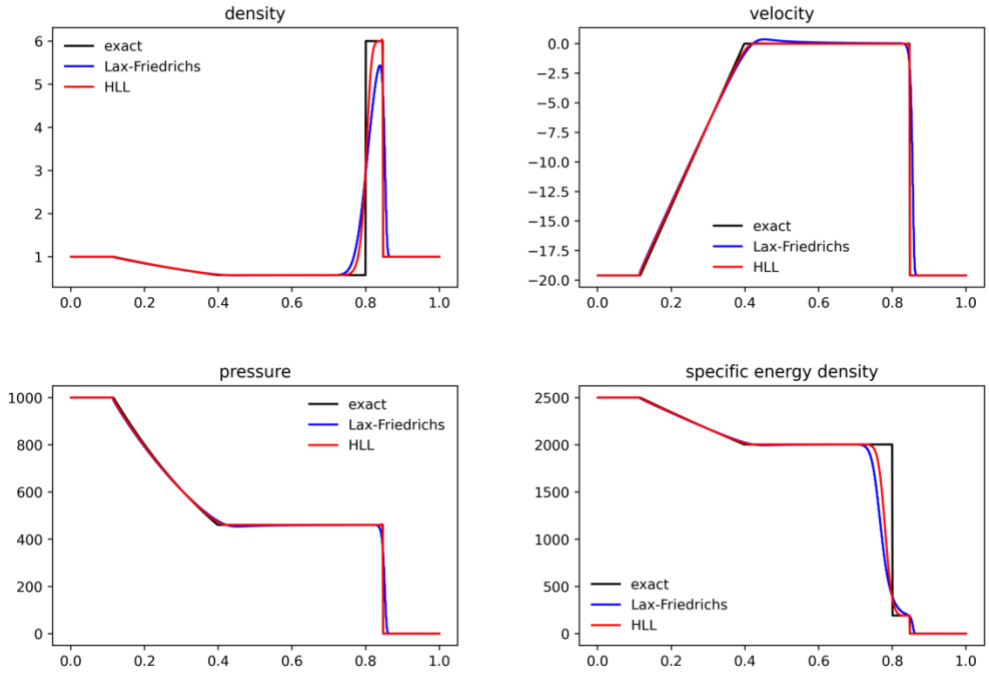

Figure 11: Standard shock tube from problem B. Both solvers use 1000 zones here, to prove the LF algorithm improves at a higher order, The snapshots are taken at t = 0.012

2.
2a

Logic
Assuming box size 10 = 2 x 10^5 cm

- [cm] = 5e-5
- rho1’ = 10

Assuming 1 dimension

- V = [cm^3] = [cm]
- rho1’ = rho1
- [cm] = [mH] = 1.67e-24 [g]
- [g] = 2.99e19

Therefore

- rho1’ = 10
- rho2’ = 30

Not need to rescale seconds

- [s] = 1
- [cm]/[s] = [cm]

Therefore

- v1’ = 50
- v2’ = -50

From ideal gas law
Assuming the whole cloud is Hydrogen

- p = n Kb T
- n = number density = N/V
- N = m/u mH

Assume u is 1

- m/u mH = rho [cm] / [cm] = rho
- N = rho

Therefore

- p1 = 10 Kb T/V
- p2 = 30 Kb T /V

- Kb = 1.38e-16 [cm] [cm] [g] / [s] [s] [K]
- T1 = 10e4 [K]
- T2 = 10e2 [K]

Therefore

- p1 = 20631
- p2 = 618.93

2b

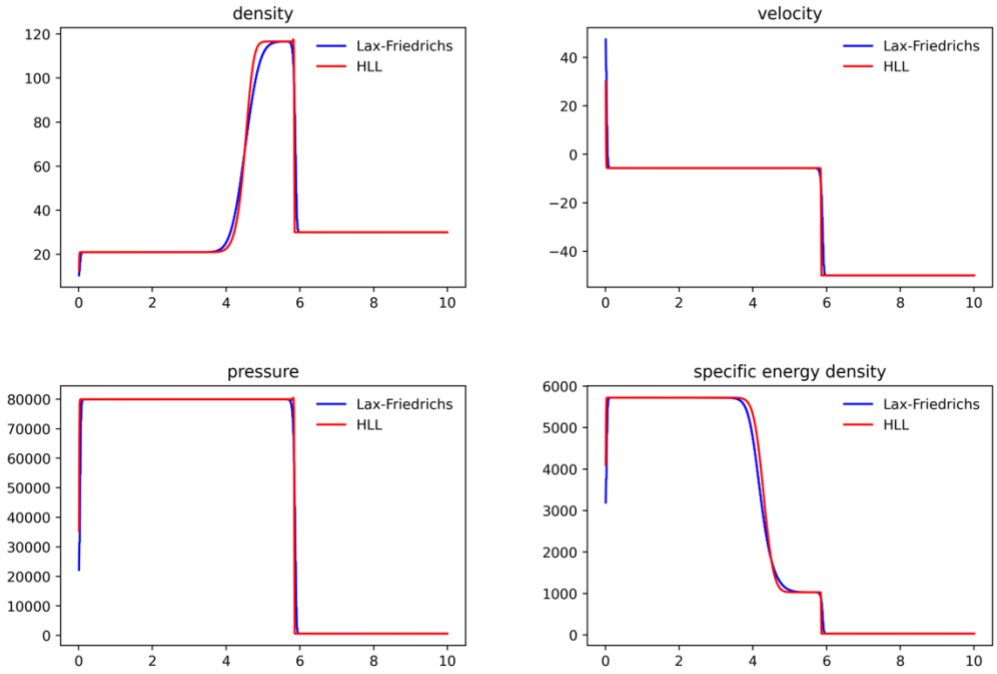

Figure 12: Standard shock tube from problem 2 A. Both solvers use 1000 zones (not counting ghost zones) between x = 0...1, scaled to 0…10. The snapshots are taken at t = 0.009.

3. Source code

```
#include <stdio.h> // pre-compiler statement
#include <stdlib.h> // free
#include <math.h> // pre-compiler statement, remember to LINK
EXPLICITLY using "-lm"
#include <time.h> // clock

/// introduce parameters
#define n_cells 200              // number of cells // Questions 2:
n_cells 1000 //
#define n_x (n_cells + 2)          // number of cells + ghost cells
#define x_initial 0          // length of tube
#define x_final 1

/// introduce variables
int i, j;
double t_final;     // final time changes with shocktube
double dx = (double) (x_final - x_initial) / n_cells;
double current_time, dt;    // timestep and current (total) time in
loop
double X[n_x], S[n_x][3];   // array for x count from 0 to 1 and L/R
wavespeed values
double F[n_x][3], Q[n_x][3], Qn[n_x][3]; // state cell vector arrays
double F_plus[n_x-1][3], F_minus[n_x-1][3]; // boundary vector arrays
```

```
                                    // defining dynamic variables
double *r0, *p0, *v0;         // r0 = rho, p0 = pressure, v0 = velocity
double *E, *c_s, *a;         // E = energy density, c_s = speed of
sound, a = max speed
double r0L, p0L, v0L, EL, c_sL; // defining variables for the HLL
method
double r0R, p0R, v0R, ER, c_sR;
double sL, sR, p0_star;

double gama = 1.4;  // gamma value for shocktube A and B, changes for C
double CFL = 1;      // to reduce stepsize in case of supersonic shock
int A = 0;          // variables and constants to allow smooth
operation
int B = 1;
int C = 2;
int LF = 3;
int HLL = 4;
int tube;


/// generate the maximum value of any array of any size int size
double maximum(double const array[], int size)
{
    double max;
    max = array[0];
    for(i=1 ;i < size - 1; i++)
    {
        if(array[i] > max)
            max = array[i];
    }
    return max;
}

/// function to allocate dynamic memory
void Allocate_Memory() {
    size_t memory_size = n_x * sizeof(double);
    r0 = malloc(memory_size);
    p0 = (double *) malloc(memory_size);
    v0 = (double *) malloc(memory_size);
    E = (double *) malloc(memory_size);
    c_s = (double *) malloc(memory_size);
    a = (double *) malloc(memory_size);
}

/// generate the initial conditions of the shock tube grid
/// dynamic input for shocktube A, B or C
void Initial_Conditions(int shocktube) {
    /// initialise cell arrays
    for(j=0; j < 3; j++) {
        for (i = 0; i < n_x ; i++)
        {
            Q[i][j] = 0;
            Qn[i][j] = 0;
            F[i][j] = 0;
            S[i][j] = 0;
            r0[i] = 0;
            p0[i] = 0;
            v0[i] = 0;
```

```
            E[i] = 0;
            c_s[i] = 0;
            a[i] = 0;
        }
        /// initialise boundary arrays
        for (i = 0; i < n_x-1; i++)
        {
            F_minus[i][j] = 0;
            F_plus[i][j] = 0;
        }
    }

    /// initial conditions of Shocktube A
    if(shocktube == A) {
        tube = A;
        t_final = 0.2;
        for (i = 0; i < n_x; i++) {
            X[i] = i * dx;
            if (X[i] < 0.3) {
                r0[i] = 1;
                p0[i] = 1;
                v0[i] = 0.75;
            } else {
                r0[i] = 0.125;
                p0[i] = 0.1;
                v0[i] = 0;
            }
        }
    }
    /// initial conditions of Shocktube B
    else if(shocktube == B) {
        tube = B;
        t_final = 0.012;
        for (i = 0; i < n_x; i++) {
            X[i] = i * dx;
            if (X[i] < 0.8) {
                r0[i] = 1;
                p0[i] = 1000;
                v0[i] = -19.59745;
            } else {
                r0[i] = 1;
                p0[i] = 0.01;
                v0[i] = -19.59745;
            }
        }
    }
    /// initial conditions of Shocktube C
    else if(shocktube == C) {
        tube = C;
        gama = 1.667;       // gamma value is 5/3
        t_final = 0.1;     // max t value calculated in Update_Timestep
function
                        // t_final stops at 0.008799 using the if loop
in Update_Timestep
                        // multiple values were used in the report
        for (i = 0; i < n_x; i++) {
            X[i] = (10 * i * dx);   // box size of 10
            if (X[i] < 5) {         // dimensionless values
```

```
                r0[i] = 10;         // calculated in report appendix
                p0[i] = 20631;
                v0[i] = 50;
            } else {
                r0[i] = 30;
                p0[i] = 618.93;
                v0[i] = -50;
            }
        }
    }
}

/// set boundary conditions/ ghost cell states
/// constant boundary conditions used for simplicity
void Boundary_Conditions() {
    /// Boundary conditions A
    if(tube == A) {
        /// boundary conditions x = 0
        r0[0] = 1;
        p0[0] = 1;
        v0[0] = 0.75;
        E[0] = p0[0] / ((gama - 1)) + r0[0] * v0[0] * v0[0] * 0.5;
        c_s[0] = sqrt((gama * p0[0]) / r0[0]);

        /// boundary conditions at x = 1
        r0[n_x - 1] = 0.125;
        p0[n_x - 1] = 0.1;
        v0[n_x - 1] = 0;
        E[n_x - 1] = p0[n_x - 1] / ((gama - 1)) + r0[n_x - 1] * v0[n_x
- 1] * v0[n_x - 1] * 0.5;
        c_s[n_x - 1] = sqrt((gama * p0[n_x - 1]) / r0[n_x - 1]);
    }
    /// Boundary conditions B
    else if(tube == B) {
        /// boundary conditions x = 0
        r0[0] = 1;
        p0[0] = 1000;
        v0[0] = -19.59745;
        E[0] = p0[0] / ((gama - 1)) + r0[0] * v0[0] * v0[0] * 0.5;
        c_s[0] = sqrt((gama * p0[0]) / r0[0]);

        /// boundary conditions at x = 1
        r0[n_x - 1] = 1;
        p0[n_x - 1] = 0.01;
        v0[n_x - 1] = -19.59745;
        E[n_x - 1] = p0[n_x - 1] / ((gama - 1)) + r0[n_x - 1] * v0[n_x
- 1] * v0[n_x - 1] * 0.5;
        c_s[n_x - 1] = sqrt((gama * p0[n_x - 1]) / r0[n_x - 1]);
    }
        /// Boundary conditions C
    else if(tube == C) {
        /// boundary conditions x = 0
        r0[0] = 10;
        p0[0] = 20631;
        v0[0] = 50;
        E[0] = p0[0] / ((gama - 1)) + r0[0] * v0[0] * v0[0] * 0.5;
        c_s[0] = sqrt((gama * p0[0]) / r0[0]);
```

```
        /// boundary conditions at x = 10
        r0[n_x - 1] = 30;
        p0[n_x - 1] = 618.93;
        v0[n_x - 1] = -50;
        E[n_x - 1] = p0[n_x - 1] / ((gama - 1)) + r0[n_x - 1] * v0[n_x
- 1] * v0[n_x - 1] * 0.5;
        c_s[n_x - 1] = sqrt((gama * p0[n_x - 1]) / r0[n_x - 1]);
    }
}

/// determine the stepsize dt based on speed of sounf and wavespeed
void Update_Timestep(){

    /// speed of sound and max possible wave speed
    for(i=1; i < n_x - 1; i++) {
        c_s[i] = sqrt((gama * p0[i]) / r0[i]);
        a[i] = fabs(v0[i]) + c_s[i];    // maximum wavespeed on grid
    }

    /// determine step size using the systems largest eigenvalue
    dt = (double) CFL * dx / maximum(a, n_x);

    /// condition for C to end script when impact parameters
    /// are felt at the end of the zone
    if (tube == C){
        if(round(r0[1]) != r0[0])
        {
            t_final = 0;
        }
    }
}

/// update cell values of state vectors
void Update_Matrix(){
    /// update values for Q, F, E and c_s
    for(i=0; i < n_x; i++)
    {
        /// specific internal energy
        E[i] = p0[i] / ((gama-1) * r0[i]) + v0[i] * v0[i] * 0.5;

        /// update state cell vector Q values
        Q[i][0] = r0[i];
        Q[i][1] = r0[i] * v0[i];
        Q[i][2] = E[i] * r0[i];

        /// update state cell vector F values
        F[i][0] = r0[i] * v0[i];
        F[i][1] = r0[i] * v0[i] * v0[i] + p0[i];
        F[i][2] = (E[i] * r0[i] + p0[i]) * v0[i];
    }
}

/// determine flux between cells using different methods
/// Lax-Friedrich or the HLL method
void Determine_Flux(int method) {
    /// implementation of the HLL method
    if(method == HLL){
        /// calculate initial cell properties Left and Right
```

```
        for (i = 0; i < n_x; i++) {
            /// initial conditions
            /// left i-1
            r0L = Q[i - 1][0];
            v0L = Q[i - 1][1] / r0L;
            EL = Q[i - 1][2];
            p0L = (gama - 1) * (EL - (r0L * v0L * v0L * 0.5));
            c_sL = sqrt((gama * p0L) / r0L);

            /// right i
            r0R = Q[i][0];
            v0R = Q[i][1] / r0R;
            ER = Q[i][2];
            p0R = (gama - 1) * (ER - (r0R * v0R * v0R * 0.5));
            c_sR = sqrt((gama * p0R) / r0R);

            /// estimate pressure between waves
            p0_star = fmax(0, 0.5 * (p0L + p0R) - 0.5 * (v0R - v0L) *
0.5 * (r0L + r0R) * 0.5 * (c_sL + c_sR));

            /// calculate the wave speed values L and R
            /// left
            if (p0_star <= p0L) {
                sL = v0L - c_sL;
            } else {
                sL = v0L - c_sL * sqrt(1 + ((gama + 1) / (2 * gama) *
((p0_star / p0L) - 1)));
            }

            /// right
            if (p0_star <= p0R) {
                sR = v0R + c_sR;
            } else {
                sR = v0R + c_sR * sqrt(1 + ((gama + 1) / (2 * gama) *
((p0_star / p0R) - 1)));
            }

            /// store values
            S[i][0] = sL;
            S[i][1] = sR;

        }

        /// 1/2 flux values
        /// calculating flux boundary values
        for (j = 0; j < 3; j++) {
            for (i = 1; i < n_x; i++) {
                sL = S[i][0];
                sR = S[i][1];
                if ((sL >= 0) && (sR >= 0)) {
                    F_minus[i][j] = F[i - 1][j];
                } else if ((sL <= 0) && (sR >= 0)) {
                    F_minus[i][j] = (sR * F[i - 1][j] - sL * F[i][j] +
                                     sR * sL * (Q[i][j] - Q[i - 1][j]))
/ (sR - sL);
                } else if ((sR <= 0) && (sL <= 0)) {
                    F_minus[i][j] = F[i][j];
                }
```

```
            }
        }
        for (j = 0; j < 3; j++) {
            for (i = 1; i < n_x - 1; i++) {
                F_plus[i][j] = F_minus[i + 1][j];
            }
        }
    }
    /// implementation of the Lax-Friedrichs algorithm
    else {
        for(j=0; j < 3; j++) {
            for (i = 1; i < n_x - 1 ; i++)
            {
            /// F + 1/2
            F_plus[i][j] = ((F[i][j] + F[i+1][j])*0.5) +
((dx/dt)*(Q[i][j] - Q[i+1][j])*0.5);
            /// F - 1/2
            F_minus[i][j] = ((F[i-1][j] + F[i][j])*0.5) -
((dx/dt)*(Q[i][j] - Q[i-1][j])*0.5);
            }
        }
    }
}

/// function to update the flux values
/// after one of either the LF or HLL method has been applied
/// to find flux values between cells
void Update_Flux() {
    for (j = 0; j < 3; j++) {
        for (i = 1; i < n_x-1 ; i++) {
            /// Q n+1 the new Q value
            Qn[i][j] = Q[i][j] - ((dt/dx) * (F_plus[i][j] -
F_minus[i][j]));
        }
    }
}

/// update initial variables with the updated state vector
void Update_Values(){
    /// assign output to a new array
    for(j=0; j < 3; j++) {
        for (i = 0; i < n_x ; i++) {
            Q[i][j] = Qn[i][j];
        }
    }
    /// update variables
    for(i=1; i< n_x-1; i++)
    {
        r0[i] = Qn[i][0];
        v0[i] = Qn[i][1] / r0[i];
        E[i] = Qn[i][2];
        p0[i] = (gama - 1) * ((E[i]) - r0[i] * v0[i] * v0[i] *0.5);
        c_s[i] = sqrt((gama * p0[i])/r0[i]);
    }
}

/// algorithm undergos a while loop
///  until the final time has been acheived by the system
```

```
void Algorithm(method){
    while(current_time < t_final)
    {
        Boundary_Conditions();

        Update_Timestep();

        Update_Matrix();

        Determine_Flux(method);

        Update_Flux();

        Update_Values();

        current_time += dt;     // update the current time
    }
}

/// save final arrays to then be printed in python
void Save_Values(){
    /// store output as a txt file to plot in python
    FILE *fp=NULL;
    fp=fopen("24511PH30110.text", "w");

    for(i=1; i < n_x-1; i++)
    {
        fprintf(fp, "%1.6f\t%1.6f\t%1.6f\t%1.6f\t%1.6f\n", r0[i],
p0[i], v0[i], p0[i]/((gama-1)*r0[i]), X[i]);
    }
}

/// free allocated memory
void Free_Memory(){
    free(r0);
    free(p0);
    free(v0);
    free(E);
    free(c_s);
    free(a);
}

/// run programme
int main() {

    Allocate_Memory();

    Initial_Conditions(A);  // assign shocktube - A, B or C

    Algorithm(HLL);     // assign method - LF or HLL

    Save_Values();

    Free_Memory();

    return 0;
}
```