Solving the Fluid Equations in 1D with a First Order Upwind Roe Scheme

Andrew Emerick

May 8, 2014

I (try to) evolve the fluid equations in one dimension using a first order Roe type solver, which is a linearized, approximate Riemann solver. Sod's tests are used to determine the effectiveness/accuracy of the solver.

As you know, this endeavour did not work out, but I lay out below what I tried to do. The formulas and methods are taken from a series of lecutures from Cornelis Dullemond at Heidelberg. I had e-mailed him to ask and see if he had a working version of the code. His response was more or less "no", but he pointed me to a text that would contain a means to properly implement the solution. Maybe I'll revist this for fun over the summer...

1 The Fluid Equations

The fluid equations can be defined using a state vector, q, and associated flux, f as

$$\partial_t q + \partial_x f_x(q) = 0, \tag{1}$$

where q and f_x are

$$q = \begin{pmatrix} \rho \\ \rho u \\ \rho e_{tot} \end{pmatrix} \qquad and \qquad f_x = \begin{pmatrix} \rho u \\ \rho u^2 \\ \rho h_{tot} u \end{pmatrix}. \tag{2}$$

ρ is the fluid density, u is the velocity in the x-direction (again, 1D), e_{tot} is the total energy, $h_{tot} = e_{tot} + P/\rho$ is the total enthalpy, and P is pressure. The sound speed is defined as the adiabatic sound speed as $C_s^2 = \gamma P/\rho$. I will use 0, 1, and 2 to denote the first, second, and third components of the state vector q.

Eq. (1) can be recast with a Jacobian matrix (not included here), which has the following eigenvalues and eigenvectors:

$$\lambda_1 = u - C_s \tag{3}$$
$$\lambda_2 = u \tag{4}$$
$$\lambda_3 = u + C_s \tag{5}$$

and

$$e_1 = \begin{pmatrix} 1 \\ u - C_s \\ h_{tot} - C_s u \end{pmatrix} \qquad e_2 = \begin{pmatrix} 1 \\ u \\ \frac{1}{2}u^2 \end{pmatrix} \qquad e_3 = \begin{pmatrix} 1 \\ u + C_s \\ h_{tot} + C_s u \end{pmatrix} \tag{6}$$

The 1st and 3rd eigenvalues denote backwards and forwards traveling sound waves, while the 2nd eigenvalue gives the entropy wave.

2 The Method

The above equations can be evolved using the Roe type solver with the following steps. I begin with a zeroth step which is setting up the grid (amounting to choosing a grid spacing Δx since I am operating on a fixed grid), defining the initial conditions (i.e. state vector q at t=0), and the boundary conditions. Once this is chosen, the following steps are iterated over.

Note: 'i' subscripts refer to the grid cell, defined at grid center. i ranges from 1 to N (N being the number of grid cells). Thus cell interfaces are denoted by i-1/2. At interfaces, 'L' and 'R' subscripts refer to the relevant value in the cell to the left and right of the interface, respectively. Superscripts (usually involving a lower-case 'n') refer to the time step. Carats denote primitive variables.

For the computation, I implement this scheme in Python. Since this was a fun programming excercise for me, I decided to play around with the inherent object-oriented-ness of Python. I think this generally turned out pretty well. I've commented the code fairly well, but please let me know if you end up looking through it and need some explanation for things. In short, I have two classes that describe the simulation grid (the 'simGrid' class) and the state of the simulation (i.e. the state and flux vectors at all grid points) (the 'simState' class). The simState class has several member functions used to set the state and caclculate primitive and interface values. The simulation state itself is evolved via the 'roe_evolve' function (not a member function), and the initial conditions are set via the 'set_initial_conditions' function. Some of the code itself is a little rough, but I've cleaned up most of it. The slope limiters are calculated in the 'slope_limiter' function. I've written this so the 'simulation' can be restarted from any write out of the variables.

2.1 Step 1: Setting the CFL

The first step is short. The size of the current time step (Δt) is chosen by setting the CFL condition at 0.5. This means

$$\Delta t = \frac{C\Delta x}{\max(|u| + |C_s|)}, \tag{7}$$

taking the denominator to be the maximum fluid velocity in the system. In practice, however, I fix the timestep to some initial (small) value and retain this so long as it meets the CFL condition. It is adjusted accordingly if this is not the case.

2.2 Step 2: q at Cell Center

The state vector q is set at cell centers. This is done such that $q_{i-1/2,R} = q_i$ and $q_{i-1/2,L} = q_{i-1}$ (by definition). At this point, one can also specify the flux vector at cell center. This can be computed directly from the state vector (not reproduced here).

2.3 Step 3: Construct Roe's Averages

The 'Roe' in the Roe type scheme comes in constructing the average values across the interfaces. A simple average, 0.5*(L + R), breaks down in the formation of contact discontinuities or shocks. The Roe solver attempts to properly take the average across the interface in order to provide an accurate representation of the discontinuities. This method solves the Riemann problem linearly, and is an approximate Riemann solver.

The Roe averages are of the form

$$\hat{u} = \frac{\sqrt{\rho_L}u_L + \sqrt{\rho_R}u_R}{\sqrt{\rho_L} + \sqrt{\rho_R}} \tag{8}$$

$$\hat{h}_{tot} = \frac{\sqrt{\rho_L}h_{tot,L} + \sqrt{\rho_R}h_{tot,R}}{\sqrt{\rho_L} + \sqrt{\rho_R}} \tag{9}$$

$$\hat{\rho} = \sqrt{\rho_L \rho_R} \tag{10}$$

$$\hat{C}_s = \sqrt{(\gamma - 1)\left(\hat{h}_{tot} - \frac{1}{2}\hat{u}^2\right)} \tag{11}$$

2.4 Step 4: Calculate λ_k and e_k

Using the Roe averages defined in Step 3, I then calculate the eigenvalues and eigenvectors as given in (3) and (6).

2.5 Step 5: Compute the Flux Jump Across the Interface

The fluid equations are evolved by calculating the flux at each interface, and adjusting the state vector accordingly. For this scheme, the time centered flux (n + 1/2) at each cell interface is given by (12).

$$f^{n+1/2}_{k,i-1/2} = \frac{1}{2}\left(f^n_{k,i-1/2,R} + f^n_{k,i-1/2,L}\right)$$
$$-\frac{1}{2}\sum_{m=1,2,3} \lambda_{m,i-1/2}\tilde{\Delta q}_{k,i-1/2}e_{m,k,i-1/2}\left[\theta_{m,i-1/2} + \hat{\phi}_{m,i-1/2}\left(\epsilon_{m,i-1/2} - \theta_{m,i-1/2}\right)\right] \tag{12}$$

There is quite a bit going on here, and it is calculated in a few steps:

2.5.1 Calculating $\tilde{\Delta}q_{k,i-1/2}$

The flux is calculated for each of the state vectors (i.e. k = 0,1,2). This involves a sum over $\tilde{\Delta}q_{k,i-1/2}$ multiplied by the three eigenvalues (m=1,2,3), and other terms (see below). The $\tilde{\Delta}q_{k,i-1/2}$ term is the jump of the state vectors across the Jacobian, and is given in terms of the difference of the state vector across the interface:

$$\tilde{\Delta}q_1 = \frac{\gamma-1}{2\hat{C}_s^2}\left[\hat{e}_{kin}\Delta q_0 - \xi\right] - \frac{\Delta q_1 - \hat{u}\Delta q_0}{2\hat{C}_s} \tag{13}$$

$$\tilde{\Delta}q_2 = \frac{\gamma-1}{2\hat{C}_s^2}\left[\left(\hat{h}_{tot} - 2\hat{e}_{kin}\right)\Delta q_0 + \xi\right] \tag{14}$$

$$\tilde{\Delta}q_3 = \frac{\gamma-1}{2\hat{C}_s^2}\left[\hat{e}_{kin}\Delta q_0 - \xi\right] + \frac{\Delta q_1 - \hat{u}\Delta q_0}{2\hat{C}_s} \tag{15}$$

where $\hat{e}_{kin} = \frac{1}{2}\hat{u}^2$ and $\xi = \hat{u}\Delta q_1 - \Delta q_2$.

2.5.2 Calculating the Flux Limiter

The choice of the flux limiter is an essential element to this scheme. $\theta_{k,i-1/2}$ in (12) is the sign of the kth eigenvalue, defined as

$$\theta_{k,i-1/2} = \frac{|\lambda_{k,i-1/2}|}{\lambda_{k,i-1/2}} = \pm 1. \tag{16}$$

$\epsilon_{k,i-1/2}$ is given as

$$\epsilon_{k,i-1/2} = \frac{\lambda_{k,i-1/2}\Delta t}{\Delta x}. \tag{17}$$

Finally, the decision between deifferent flux limiting schemes is a choice on $\tilde{\phi}_{k,i-1/2}$. $\tilde{\phi}_{k,i-1/2}$ is a function of a parameter r, defined as

$$r^n_{k,i-1/2} = \begin{cases} \frac{\tilde{q}_{k,i-1} - \tilde{q}_{k,i-1/2}}{\tilde{q}_{k,i} - \tilde{q}_{k,i-1}} & \lambda_{k,i-1/2} > 0 \\ \\ \frac{\tilde{q}_{k,i+1} - \tilde{q}_{k,i}}{\tilde{q}_{k,i} - \tilde{q}_{k,i-1}} & \lambda_{k,i-1/2} < 0. \end{cases}$$

$\tilde{\phi}_{k,i-1/2}$ is given below for a few slope limiter cases:

$$\tilde{\phi}_{k,i-1/2}\left(r^n_{k,i-1/2}\right) = \begin{cases} 0 & \text{Donor-cell} \\ 1 & \text{Lax-Wendroff} \\ r & \text{Beam-Warming} \\ \frac{1}{2}(1+r) & \text{Fromm} \\ \text{minmod}(1,r) & \text{minmod} \\ \max(0, \min(1,2r), \min(2,r)) & \text{superbee} \end{cases}$$

In the final implementation of the Roe solver, I utilize the non-linear superbee slope limiter, which alternates between a second and first order slope limiter depending upon whether or not a shock is present. This limiter is TVD. All of these slope limiters are made available as a switch in my code.

2.5.3 Calculation of the flux difference

The final step to calculate the flux across the interface is to calculate the flux average $1/2\Big(f^n_{k,i-1/2,R} + f^n_{k,i-1/2,L}\Big)$.

2.6 Final Step: Update the State Vector

Now that the flux across the interface is determined, we can update the state vector q accordingly:

$$q_i^{n+1} = q_i^n - \frac{\Delta t}{\Delta x}\left(f_{i+1/2}^{n+1/2} - f_{i-1/2}^{n+1/2}\right) \tag{18}$$

Then, we repeat all steps, beginning with calculating the next Δt using the CFL condition, until the run reaches the specified t_{final}.

3 Boundary Conditions

I choose simple boundary conditions where the state vector is fixed to the initial starting (t=0) value at the left and right edges. This is done easily by suppling ghost cells at the boundary. I include the ability to turn on periodic bounary conditions instead, but have not spent much time testing this to ensure it is functioning properly. It would be relatively straightfoward to also implement reflective boundary conditions.

4 Numerical Viscosity

As an attempt to improve the performance of my code, I included the ability to turn numberical viscosity on/off via a switch. Although the linearized Roe solver should not require numerical viscosity, the effects of removing / adding it are noticeable.

I add von Neumann-Richtmyer artificial viscosity, which acts as an additional pressure term that serves to smooth out a shock (if present) over a few grid cells. This smoothing is tunable by the parameter ξ, which is on the order of a few (3 in my case). The additional pressure term Π, is given as

$$\Pi_i = \begin{cases} \frac{1}{4}\xi^2 \left(u_{i+1} - u_{i-1}\right)^2 \rho_i & \text{if} u_{i+1} \leq u_{i-1} \\ 0 & \text{if} u_{i+1} > u_{i-1} \end{cases}$$

5 Sod's Test Cases

Well, as you know, my final implementation fails. I've made an attempt to hunt down the source of the error, but have been unable to do so thus far. In short, the algorithm as just described is very numerically unstable, and is prone to breaking within a few time steps with either negative / infinite pressures or densities. If the simulation does manage to survive blowing up immediately, it develops some hideous ringing that propogates throughout the 1D box, blowing up onces the two ringing "waves" turn back around at the boundary and collide.

5.1 Test Case 1

The first test case is fairly simple. The grid is divided in half, setting the density, pressure, and velocity at t=0 to constant on the left and right half of the grid. This establishes a shock at the center of the grid. For the left (L) and right (R) halves: $\rho_L = 1.0\times10^5$, $\rho_R = 1.25\times10^4$, $P_L = 1.0$, $P_R = 0.1$, and u_L=u_R=0.