Computational Exercise #2

Astrophysical Fluids — PHYS 643

Fall 2018

Bridget Andersen

October 23, 2018

Description of Files

- **hw2.pdf** : contains the write-up for this assignment
- **hw2_general.py** : the python file where the donor cell 1D hydrodynamics simulation code is kept

Note that the code file "hw2_general.py" can take command line arguments that specify the particular simulation to run. By default, the code runs the simulation described in the **Small-Amplitudes** section. To see options from the command line, type `python hw2_general.py --help`. Feel free to play with the parameters, but I can't promise that the plots will scale nicely by default (I have hardcoded nice scalings for the example simulations). Set `--example_scaling` equal to 0 if you want automatic scaling.

Problem Description

The goal of this computational exercise was to create a simple 1D hydrodynamic code and use it to demonstrate the steepening of a sound wave. To do this, we use donor cell advection to solve the fluid equations in their flux-conservative form:

$$\frac{\partial f_1}{\partial t} + \frac{\partial}{\partial x}\left(v f_1\right) = 0 \tag{1}$$

$$\frac{\partial f_2}{\partial t} + \frac{\partial}{\partial x}\left(v f_2\right) = -\frac{\partial P}{\partial x} \tag{2}$$

where the conserved quantities are the mass density $f_1 = \rho$ and the momentum density $f_2 = \rho v$. Here, the pressure gradient $\partial P/\partial x$ acts as a source term for the momentum density. In this computational exercise, we will assume that $P = c_s^2 \rho$ where c_s is the constant sound speed.

Algorithm

Donor cell advection is a finite-volume method that solves the fluid equations in their flux-conservative form. "finite-volume" means that a given volume is separated into cells such that the grid points x_j

are at the center of each cell. The donor cell scheme itself is based on the conservation of a given quantity, given by:

$$\frac{\partial f}{\partial t} = -\frac{\partial J}{\partial x} \tag{3}$$

In first-order discretized form, this becomes:

$$\frac{f_j^{n+1} - f_j^n}{\Delta t} = \frac{J_{j+\frac{1}{2}}^{n+\frac{1}{2}} - J_{j-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta x} \tag{4}$$

where $J_{j\pm\frac{1}{2}}^{n+\frac{1}{2}}$ is the flux of quantity f at the boundaries of each cell, averaged over a timestep. According to the donor cell advection scheme, this flux can be approximated as:

$$\begin{aligned}
J_{j+\frac{1}{2}} &= v_{j+\frac{1}{2}} f_j^n \quad v_{j+\frac{1}{2}} > 0 \\
J_{j+\frac{1}{2}} &= v_{j+\frac{1}{2}} f_{j+1}^n \quad v_{j+\frac{1}{2}} < 0 \\
J_{j-\frac{1}{2}} &= v_{j-\frac{1}{2}} f_{j-1}^n \quad v_{j-\frac{1}{2}} > 0 \\
J_{j-\frac{1}{2}} &= v_{j-\frac{1}{2}} f_j^n \quad v_{j-\frac{1}{2}} < 0
\end{aligned}$$

Depending on the sign of the velocity at the boundaries, the contents are either advected out cell j or into cell j from the lower $(j-1)$ or upper $(j+1)$ neighbour. We can approximate the velocities at the boundaries by averaging over the central velocities in the neighboring cells:

$$\begin{aligned}
v_{j+\frac{1}{2}} &= \frac{1}{2}\left(v_j + v_{j+1}\right) \\
v_{j-\frac{1}{2}} &= \frac{1}{2}\left(v_j + v_{j-1}\right)
\end{aligned}$$

We can now apply this scheme to solve for equations 1 and 2. Each timestep will be completed in a series of stages. Note that, in our system of equations, the velocity at the center of the grid v_j is just given by f_2/f_1.

1) First, we update the value of f_1 using the same form as equation 4:

$$\frac{{}^1f_j^{n+1} - {}^1f_j^n}{\Delta t} = -\frac{{}^1J_{j+\frac{1}{2}}^{n+\frac{1}{2}} - {}^1J_{j-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta x} \rightarrow \boxed{{}^1f_j^{n+1} = {}^1f_j^n - \frac{\Delta t}{\Delta x}\left({}^1J_{j+\frac{1}{2}}^{n+\frac{1}{2}} - {}^1J_{j-\frac{1}{2}}^{n+\frac{1}{2}}\right)} \tag{5}$$

Note that we have written $f_1 \rightarrow {}^1f$ to reduce confusion between indices.

2) Next, we update the value of f_2. The update for f_2 is the same as for f_1, except that we need to take into account the pressure gradient source term. We can do this by discretizing the $\partial P/\partial x$ derivative in the second order as follows:

$$\frac{\partial P}{\partial x} = c_s^2 \frac{\partial \rho}{\partial x} \implies \frac{\partial P}{\partial x} = c_s^2\left(\frac{\rho_{j+1} - \rho_{j-1}}{2\Delta x}\right)$$

Since $f_1 = \rho$, then we can just substitute in to the discrete pressure gradient and then use equation 5 to find the total timestep for f_2:

$$\frac{{}^2f_j^{n+1} - {}^2f_j^n}{\Delta t} = -\frac{{}^2J_{j+\frac{1}{2}}^{n+\frac{1}{2}} - {}^2J_{j-\frac{1}{2}}^{n+\frac{1}{2}}}{\Delta x} - c_s^2\left(\frac{{}^1f_{j+1}^n - {}^1f_{j-1}^n}{2\Delta x}\right) \tag{6}$$

$$\rightarrow \boxed{{}^2f_j^{n+1} = {}^2f_j^n - \frac{\Delta t}{\Delta x}\left[{}^2J_{j+\frac{1}{2}}^{n+\frac{1}{2}} - {}^2J_{j-\frac{1}{2}}^{n+\frac{1}{2}} - c_s^2\left(\frac{{}^1f_{j+1}^n - {}^1f_{j-1}^n}{2}\right)\right]} \tag{7}$$

3) In the simulation, we implement ghost cells on either end of the 1D grid to help us control the boundary conditions of the simulation. Thus, the final step of the simulation is to update the boundary conditions in the ghost cells. We implement periodic boundary conditions by using the `numpy.roll()` function to shift our data arrays when making the calculations shown above, so the updating of our boundary conditions is intrinsic to calculating each timestep.

Results

With the framework described above, we can simulate a number of simple 1D sound waves. We have created a code that is capable of simulating simple Gaussian and sinusoidal density profiles and simple constant, zero, Gaussian, and sinusoidal velocity profiles. Our code also takes in many parameters controlling the amplitudes of the density/velocity profiles, the size of the timestep, the number of grid positions to sample, the sound speed, and the number of iterations to complete (for a full list of parameters, type `python hw2_general.py --help` in the command line). In this section, we discuss a few qualitative effects seen when tuning the parameters to different values.

1 Small-Amplitudes

A good way to test if our simulation works as expected for small amplitudes (small velocities and small density perturbations) is to start with a low-amplitude gaussian density profile and a zero initial velocity profile. To view this simulation, run the following command:

```
python hw2_general.py --p_amp 0.1 --v_type zeros --p_type gaussian --n_iter 5000
```

We can see that the initial density Gaussian collapses and forms two Gaussian waves propagating in opposite directions without any noticeable steepening.

2 Steepening

At larger density amplitudes, we expect the wave to propagate and experience steepening. To see this effect, we can run a simulation with initial in-phase sinusoidal density and velocity profiles. We also increase the... Note that the sound speed is, by default, $c_s = 1$. To see this simulation, run the following command:

```
python hw2_general.py --p_amp 2.  --v_amp 1.  --v_type sinusoid --p_type sinusoid --n_iter 5000
```

We start out with sinusoidal density waves advecting to the right. Gradually, the sinusoids start to steepen and appear more triangle-like. Eventually we see shock-like structures manifested as large discontinuities between the tops and bottoms of the velocity waves. Then stresses from numerical viscosity cause the system to relax. The shock fronts disappear and both the density and velocity return to moderately steepened wavefronts with reduced amplitudes.

3 Numerical Stability Dependent on the Timestep

To investigate the numerical stability of our algorithm, we can run several simulations iterating over different values for the timestep until we reach an unstable solution. The following commands run the same sinusoidal density/velocity simulation for timesteps of `dt=0.1, 0.3, 0.5`:

```
python hw2_general.py --p_amp 2.  --v_amp 1.  --v_type sinusoid --p_type sinusoid --dt 0.1 --n_iter 5000
python hw2_general.py --p_amp 2.  --v_amp 1.  --v_type sinusoid --p_type sinusoid --dt 0.3 --n_iter 5000
python hw2_general.py --p_amp 2.  --v_amp 1.  --v_type sinusoid --p_type sinusoid --dt 0.5 --n_iter 5000
```

We can see that the first simulation (`dt=0.1`) is marginally stable, but is starting to show instability at high velocity values. In the `dt=0.3` simulation, we get significant instability but the density and

velocity values remain finite. In the `dt=0.5` simulation, the density and velocity values explode and our code has become completely unstable.

Therefore, our simulation starts to become unstable at a timestep of `dt=0.1`.

4 Forming Shocks

As discussed in the notes, the width of the shocks that we can form are controlled by the viscous term in the momentum equation. Viscous stresses act to smooth out the velocity gradient, balance the wave steepening, and increase the width of the shocks that we can form. Although we have not explicitly written a viscous term in our momentum equation, the numerical precision of our simulation introduces a term that acts like a viscosity. To demonstrate this, we can run two different simulations with different grid resolutions:

```
python hw2_general.py --p_amp 2.  --v_amp 3.  --v_type sinusoid --p_type sinusoid --dt 0.01   //
    --n_iter 5000 --num_x 100
python hw2_general.py --p_amp 2.  --v_amp 3.  --v_type sinusoid --p_type sinusoid --dt 0.01   //
    --n_iter 5000 --num_x 1000
```

We can visualize the "thickness" of the shock by looking at the slope of the formed discontinuity. We can see that, as the grid resolution increases from `num_x=100` to `num_x=1000`, the shock becomes steeper.