Chapter 3

Numerical hydrodynamics of astrophysical gas clouds

3.1 Gas dynamics in astronomy

The space between the stars is not empty. It contains large quantities of hydrogen and helium gas, albeit at very low densities. This *interstellar medium* is not homogeneous: it consists of cold ($\sim$ 10 K) turbulent *molecular clouds* and hot tenuous atomic gas in between. In the cold molecular clouds new stars are formed. An example is the Orion molecular cloud complex, of which the Orion nebula can be seen with the naked eye.

The gas is always in motion, like waves on the sea. These motions, and the laws that govern them, are called *gas dynamics*, or equivalently *hydrodynamics*. When dense clumps of gas form, the self-gravity of the gas can cause it to collapse and form a star. Since no clump will have exactly zero rotation, the angular momentum conservation typically leads to the formation of a *protostellar disk* aroud the growing star. Once most of the disk gas has accreted onto the star, the remaining disk is called a *protoplanetary disk* because it is believed that planetary systems form in them. Also the gas in these disks moves around and features waves.

The gas inside stars and planets also obeys the laws of gas dynamics. The simplest example is the weather on Earth. In other words: astronomy is full of objects that are made of gas, and thus behave according to the laws of gas dynamics.

The dynamics of gas, for historical reasons often called *hydrodynamics* even though it has nothing to do with water, is complex. Again just think of the weather patterns on Earth and you see why. Hydrodynamics displays extremely *non-linear* behavior. Analytical solutions are rare and/or approximate. Usually it is necessary to employ *numerical hydrodynamics codes* to model the behavior of the gas. In this chapter we will give a *very brief* introduction to this topic.

3.2 Equations of hydrodynamics

The motion of gas in most astrophysical settings follows the equation of *compressible hydrodynamics*. In 1-D these equations are:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho v}{\partial x} = 0 \tag{3.1}$$

$$\frac{\partial \rho v}{\partial t} + \frac{\partial \rho v^2 + P}{\partial x} = \rho f \tag{3.2}$$

where the pressure $P = \rho c_s^2$, with c_s the *isothermal sound speed* given by

$$c_s = \sqrt{\frac{k_B T}{\mu m_p}} \tag{3.3}$$

with k_B the Boltzmann constant, m_p the proton mass and $\mu = 2.3$ the mean molecular weight of a H_2,He mixture. The right-hand side of Eq. (3.2) contain a possible external force, such as the gravitational force (either by an external body or by the gas itself).

For simplicity we will *assume* from now on that the temperature T is everywhere the same, both in space and time. Therefore c_s^2 is a global constant, specified by the scientist. This means that we do not have to solve the energy equation, which simplifies the problem.

3.3 Advection on a grid: Numerical diffusion/viscosity

The two equations (Eqs. 3.1, 3.2) are the conservation equations of mass and momentum, respectively. The both take the form of an *advection equation*:

$$\frac{\partial q(x,t)}{\partial t} + \frac{\partial q(x,t)\, v(x,t)}{\partial x} = s(x,t,q) \tag{3.4}$$

where $q = \rho$ and $s = 0$ for the first equation and $q = \rho v$ and $s = -\partial p/\partial x + \rho f$ for the second equation.

An advection equation is conceptually very simple: it is the equation that moves around stuff. The velocity of the movement is $v(x,t)$ and the "stuff" is $q(x,t)$.

One would intuitively think that "moving stuff around" is a simple task, even for a computer. It turns out, however, that for a computer this task is very hard. Consider the following simple problem:

$$\frac{\partial q(x,t)}{\partial t} + v_0 \frac{\partial q(x,t)}{\partial x} = 0 \tag{3.5}$$

with $v_0 > 0$ a positive constant. This is the problem of shifting a function to the right with a velocity v_0. Suppose our function at $t = 0$ is the following step function:

$$q(x,0) = \begin{cases} 1 & \text{for} \quad x < 0 \\ 0 & \text{for} \quad x \geq 0 \end{cases} \tag{3.6}$$

Clearly the solution to this advection equation (Eq. 3.5), with initial condition given by Eq. (3.6), for $t > 0$ is:

$$q(x,t) = \begin{cases} 1 & \text{for} \quad x < v_0 t \\ 0 & \text{for} \quad x \geq v_0 t \end{cases} \tag{3.7}$$

Again: analytically this is trivial.

But for modeling numerical hydrodynamics we wish to advect stuff on a grid in x, i.e. we describe $q(x,t)$ as

$$q_i^n = q(x_i, t_n) \tag{3.8}$$

We have no more information about $q(x,t)$ than the discrete set of numbers q_i^n. *If and only if* v_0 is constant, the spatial grid spacing $\Delta x = x_{i+1} - x_i$ is constant, and the time spacing $\Delta t = t_{n+1} - t_n$ is exactly equal to $\Delta t = k\Delta x/v_0$ with k a positive integer, then the numerical solution can be expressed exactly as $q_i^{n+1} = q_{i-k}^n$.

But in practice these conditions are rarely met, because in most cases of practical interest, v is not a constant in space or because other constraints make it impossible to choose Δt as that precise value. Suppose, for instance, that $\Delta t = \frac{1}{3}\Delta x/v_0$ (still keeping v_0 constant). Then the value of q_i^{n+1} (i.e. at the next time step) should become the value of $q_{i-1/3}^n$ (at the current time step). However, $q_{i-1/3}^n$ does not exist: it is a value of q between two grid points. One has no other choice than to perform an interpolation of some kind. And interpolations smear things out. This means that *advection on a grid leads to diffusion*. This kind of diffusion is called *numerical diffusion*. In the case of the momentum equation (Eq. 3.2) this is numerical momentum-diffusion, which we usually call *numerical viscosity*.

Note: It is important to understand that numerical diffusion/viscosity is unavoidable when modeling hydrodynamics on a grid! The better a hydrodynamic algorithm, the more such numerical diffusion/viscosity can be suppressed, but it can never be eliminated. And the suppression of numerical diffusion/viscosity comes at a price: it can make the code unstable or can lead to spurious (unphysical) features.

3.4 Advection on a grid: Naive approach which fails spectacularly

Let us try to come up with a super-simple advection method. Again, let us focus on the simple test problem of Eq. (3.5). If we employ the same logic of discretization as we used in Section 1.2, we arrive at:

$$\frac{q_i^{n+1} - q_i^n}{t_{n+1} - t_n} + v_0 \frac{q_{i+1}^n - q_{i-1}^n}{x_{i+1} - x_{i-1}} = 0 \tag{3.9}$$

Solving for q_i^{n+1} yields:

$$q_i^{n+1} = q_i^n - v_0(t_{n+1} - t_n)\frac{q_{i+1}^n - q_{i-1}^n}{x_{i+1} - x_{i-1}} \tag{3.10}$$

We still need to deal with the boundary conditions. We do this by defining the first and last grid point as dummy points: we do not update these points, instead we keep the values constant: $q_0^{n+1} = q_0^n$ and $q_{I-1}^{n+1} = q_{I-1}^n$ where I is the number of grid points. For $i > 0$ up to $i < I - 1$ we employ Eq. (3.10). Now let us see what happens.

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

nx   = 100
nt   = 100
x    = np.linspace(0.,10.,nx)
t    = np.linspace(0.,10.,nt)
q    = np.zeros((nt,nx))
q[0,x<3] = 1.0     # Step function initial condition
v0   = 0.4         # Constant advection velocity
for n in range(0,nt-1):
    dt          = t[n+1] - t[n]
    q[n+1,1:-1] = q[n,1:-1] - dt * v0 * ( q[n,:-2] - q[n,2:] ) / ( x[:-2] - x[2:] )
    q[n+1,0]    = q[n,0]
    q[n+1,-1]   = q[n,-1]

n = 0
def animupdate(frameNum, a0):
    global n,q,x,nt
    y = q[n,:]
    a0.set_data(x, y)
    n = (n + 1) % nt

fig = plt.figure()
ax  = plt.axes(xlim=(x.min(),x.max()),ylim=(0.,1.5))
a0, = ax.plot([], [])
anim = animation.FuncAnimation(fig,animupdate,fargs=(a0,),interval=40)
plt.show()
```

You can find this snippet in `snippets/snippet_naive_advection.py`.

You see that exponentially growing oscillations appear, which make the solution entirely useless. Apparently this naive method fails.

3.5 Advection on a grid: The upwind method

There is, in fact, a logical explanation for why the naive approach fails. The spatial derivative is symmetric: it takes the difference between q_{i+1}^n and q_{i-1}^n, i.e. both adjacent values of q_i^n. However, in our test problem, the flow velocity $v_0 > 0$ (moving to the right), meaning that in order to know what q_i^{n+1} is, it should be sufficient to only use information about q for grid points to the left. In other words: the value of q_{i+1}^n should not be involved at all, because that information flows further to the right, and should never affect q_i^{n+1}. A method that takes this into account is called an *upwind method*, because it only uses information from the direction from "where the wind comes from".

Here is how the simplest upwind method is implemented: instead of using the symmetric spatial derivative, we should only use, for $v_0 > 0$, the left derivative (involving q_i^n and q_{i-1}^n). For the opposite case where $v_0 < 0$ we should use the right derivative (involving q_{i+1}^n and q_i^n).

Exercise 23: Adapt the naive method to an upwind method. You can still assume that $v_0 > 0$ is constant everywhere. Try it out and demonstrate that this method remains stable. Describe what you see in terms of numerical diffusion.

3.6 Advection on a grid: Finite Volume methods

In general the v_0 is not a constant, neither in time nor in space. It may, in fact, change sign from one part of the grid to another. It is therefore not possible to decide globally whether to use a left- or a right-sided derivative. Also the x-grid may not be constant-spaced as in the example.

Under these adverse conditions, how do we still create a stable algorithm, and how do we ensure that a conserved quantity q is indeed conserved?

The trick is to think in terms of *grid cells* instead of grid points. Between every two adjacent grid points i and $i+1$ there is a *cell wall* that separates the cell domains belonging to grid points i and $i+1$ respectively. *For notational convenience* we write the indices of these cell walls as $i+1/2$ (for the wall between cells i and $i+1$). The cell wall location is thus written as $x_{i+1/2}$. The $1/2$ is just a notational convention. A computer does not understand half-integer indices of arrays. So for the implementation in the computer we need to decide how to index these cell walls. *Our convention* is that the cell wall between cells i and $i+1$ has index i, and that for I cells we have $I-1$ cell walls.

Now that all grid points (except the boundary grid points, in our convention) become *grid cells*, we can now formulate the flow of quantity $q(x,t)$ as flow through cell walls. Any "stuff" passing through a cell wall leaves one cell, and automatically enter the adjacent cell. There is no danger of numerically accidently losing stuff.

The class of methods that employ this cell-structure is called *finite volume methods*. Each cell has a volume V_i (which in 1-D is simply the size of the cell $V_i = x_{i+1/2} - x_{i-1/2}$). Each cell wall has a surface area $A_{i+1/2}$ (which in 1-D is simply $A_{i+1/2} = 1$).

Updating the q_i^n to the new q_i^{n+1} now goes in two steps:

1. Compute, using some algorithm (to be discussed later), the interface fluxes at half-time $f_{i+1/2}^{n+1/2}$.
2. Compute q_i^{n+1} according to bookkeeping how much flux enters and leaves the cells:

$$V_i q_i^{n+1} = V_i q_i^n + (t_{n+1} - t_n)\left(f_{i-1/2}^{n+1/2} - f_{i+1/2}^{n+1/2}\right) \tag{3.11}$$

In this scheme we can only have "leakage" of q at the boundaries. For the rest this scheme is perfectly *numerically conservative*.

The big question remains step 1: what algorithm can we use to compute $f_{i+1/2}^{n+1/2}$? This is the topic of the next section.

3.7 Advection on a grid: The donor-cell algorithm

The simplest finite volume scheme is the donor-cell scheme. In this scheme the "average interface state" is simply:

$$\tilde{q}_{i+1/2}^{n+1/2} = \begin{cases} q_i^n & \text{for } v_{i+1/2} > 0 \\ q_{i+1}^n & \text{for } v_{i+1/2} < 0 \end{cases} \tag{3.12}$$

This means that the donor-cell interface flux is:

$$f_{i+1/2}^{n+1/2} = \begin{cases} v_{i+1/2}\, q_i^n & \text{for } v_{i+1/2} > 0 \\ v_{i+1/2}\, q_{i+1}^n & \text{for } v_{i+1/2} < 0 \end{cases} \tag{3.13}$$

The physical interpretation of this method is the following. One assumes that the density is constant within each cell. We then let the material flow through the cell interfaces, from left to right for $v_{i+1/2} > 0$. Since the density to the left of the cell interface is constant, and as long as we choose the time step small enough (see Section 3.8 for the CFL condition), we know that for the whole time between time t_n and t_{n+1} the flux through the cell interface (which is $\tilde{q}_{i+1/2}\, v_{i+1/2}$) is constant, and is equal to Eq. (3.13).

Note that the velocity $v_{i+1/2}$ is defined at the cell walls, and may vary from cell wall to cell wall (and even change sign).

Exercise 24: Develop a function that performs a single advection step with the donor-cell method. You can start from the snippet that follows. Test this function on the simple advection problem we already solved.

The starting snippet for the above exercise is:

```
def advect_timestep_donorcell(x,q,vi,dt):
    """
    Simple donor-cell advection scheme. First and last cell
    are ghost cells and are not affected.

    ARGUMENTS:
    x      The cell center locations (array of nx elements)
    q      The cell center q values (array of nx elements)
    vi     The velocities at the cell walls (array of nx-1 elements)
    dt     The time step

    RETURNS:
    qnew   The updated values of q at the next time step

    """
    xi   = 0.5*(x[1:]+x[:-1])     # Location of cell walls
    dx   = xi[1:]-xi[:-1]         # Array of cell widths
    dx   = np.hstack((0,dx,0))    # Get dx indices in line with cell indices
    fi   = np.zeros_like(vi)      # Prepare array of interface fluxes
    ipos = np.where(vi>=0.)[0]    # Indices of interfaces with positive vi
    ineg = np.where(vi<0.)[0]     # Indices of interfaces with negative vi

    <<< COMPLETE THE ALGORITHM HERE >>>
```

You can find this snippet in `snippets/snippet_donor_cell_template.py`.

Since the results for this simple test problem are identical to the simple upwind scheme, let us now try a more complex problem:

$$v(x) = v_0 \sin(2\pi x/(x_{\max} - x_{\min})) \tag{3.14}$$

Exercise 25: Apply your advection method to this more complicated problem and show that you get regions where q decreases (why?) and regions where q increases (why?).

3.8 Limitation on the time step (the CFL condition)

Finite volume methods assume that within each time step the flow of stuff through one cell wall does not pass through the next cell wall too. The reason is that we assume that each cell only receives stuff from its neighbor, not from the neighbor's neighbor. This automatically sets a rigorous limit on the time step $\Delta t = t_{n+1} - t_n$:

$$\Delta t < \frac{\Delta x}{|v|} \tag{3.15}$$

Typically it is even better to impose an even slightly more restrictive limit: $\Delta t < 0.5\Delta x/|v|$, but that may depend on the algorithm.

3.9 Making a hydro code from the advection function

Now that we have an advection function, we can use this to solve the hydrodynamics equations, which we repeat here for convenience:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho v}{\partial x} = 0 \tag{3.16}$$

$$\frac{\partial \rho v}{\partial t} + \frac{\partial \rho v^2}{\partial x} = -\frac{\partial P}{\partial x} + \rho f \tag{3.17}$$

For the advection equations we need to find the interface velocities. We can obtain the *cell center velocities* from:

$$v_i = \frac{(\rho v)_i}{\rho_i} \tag{3.18}$$

Next we find the interface velocities by interpolation:

$$v_{i+1/2} = \frac{1}{2}(v_i + v_{i+1}) \tag{3.19}$$

Next we call the advection function for ρ and for (ρv). We update the boundary values according to our boundary condition model. We then calculate the pressure from the *new* ρ and the known c_s^2. Then we add the centered difference of the pressure times the time step

$$-\Delta t \left.\frac{\partial p}{\partial x}\right|_i \simeq -\Delta t \frac{p_{i+1} - p_{i-1}}{x_{i+1} - x_{i-1}} \tag{3.20}$$

to the momentum density $(\rho v)_i$. Then we are done for this time step.

Exercise 26: Build a function that performs this task.

Let us set up a test hydrodynamics problem: the domain is $0 < x < 2\pi$ with 100 spatial grid cells, the isothermal sound-speed-squared is $c_s^2 = 1$, the initial density profile is

$$\rho(x, t = 0) = 1 + 0.3\cos(x) \tag{3.21}$$

the time step is $\Delta t = 0.025$, we take 1000 time steps.

Exercise 27: Now model this problem with your code, and make an animation of it.