Writing a 1D finite volume solver

Introduction

In this short practical exercise, you will learn how to write a very basic, yet powerful 1D finite volume solver, that can be used to solve the Euler equations of hydrodynamics in the case of a polytropic ideal gas with an equation of state of the form

$$p = (\gamma - 1)\rho e, \tag{1}$$

where p is the pressure of the ideal gas, ρ is its density, e is its thermal energy, and γ is a constant *adiabatic index*, for which you will use the value $\gamma = 5/3$ throughout this exercise.

The aim of the exercise is to numerically solve the 1D Sod shock problem. The initial condition of the problem is defined as follows: in a 1D "box" of unit side length (x coordinate values in the range $[0, 1]$), the density (ρ), fluid velocity (u), and pressure (p) are given by:

$$(\rho(x), u(x), p(x)) = \begin{cases} (1, 0, 1) & x < 0.5, \\ (0.125, 0, 0.1) & 0.5 \leq x. \end{cases} \tag{2}$$

You goal is to obtain the solution of this system at a time $t = 0.2$, using the techniques and sample code introduced in the lectures about finite volume methods.

In principle, you could use any programming language to do this, but for this exercise you will use Python, as it is a much more flexible language than high-level languages like Fortran or C(++), and ships with a library that can be used to make plots. This flexibility comes with a significant loss in computational efficiency, but Python is still fast enough for this 1D exercise.

As discussed in the lectures, a key component of any finite volume method is the Riemann solver, which is used to obtain physically sensible fluxes at the boundaries between neighbouring cells. Since writing a Riemann solver is a very tedious process from which you would not really learn anything useful, you can use the Riemann solver provided on `https://github.com/bwvdnbro/python_finite_volume_solver`. This solver can be included in your own Python program as an external library that you can call wherever you need it. In the same online code repository you will also find another file that computes a reference solution for the 1D Sod shock problem, which you can use to see how accurate your own solver is.

The steps below will guide you through the development process of your own 1D finite volume solver.

Setting up the grid

The first step is to set up the grid that you will use for the spatial discretization of the fluid variables. You will need to subdivide the 1D "box" into a number of *cells*, which each contain values for the fluid variables. As discussed in the lectures, you need two sets of variables: *conserved variables* (mass m, fluid momentum q and total energy E), and

primitive variables (density ρ, fluid velocity u and pressure p). The former are, as their name suggests, conserved, i.e. the total sum of these variables across all cells is constant in time, and the only change in these variables happens through a *flux exchange* between cells. The primitive variables can be derived from the conserved variables if the volume of the cell is known. They are not conserved, but are the variables of interest to us (note that the initial condition of the Sod shock problem was defined in terms of these variables). They are also the variables that enter the Riemann solver and the expressions for the fluxes.

As an extra, you will also need to know which cells are neighbours. This can be easily realized by ordering the cells from left to right in a list or array. And if you want to be able to make plots of the fluid variables as a function of position, you will also need to keep track of the cell positions. The best way to do this is by storing the midpoint of each cell as an extra cell variable. A last important variable is the cell volume (which is actually rather a length in the 1D case). Although you will start by using the same volume for each cell, it is a good idea to make your program more general from the start, so you should store the volume of each cell as well.

Create a Python class (called `Cell`) that contains fields for the primitive and conserved variables, and the cell midpoint and volume, and initialize these variables to 0 in the class initialization function (which is called `__init__(self)`). Note that class variables are always preceded by the `self` keyword, and that it is a widely used convention to start names of class member variables with an underscore, to distinguish them from normal variables, e.g. `self._density`.

Next, you will need to initialize your cells using the initial condition. Note that you will need to initialize both the conserved variables and the primitive variables, while only the primitive variables are given. At the same time, you will also need to set the midpoints of the cells, and their volumes.

Create an empty list that will contain the cells. Write a `for` loop that creates 100 equally spaced cells in the $[0, 1]$ simulation box, and set the volumes and midpoints of these cells. Use the initial conditions above to set the primitive and conserved variables for the cells.

Before you continue, you need to make sure the set up routine worked properly. There are various ways to print out the contents of variables using Python's `print` routine, but it would also be good to do a visual inspection of your cells by making a plot. You will need to make a plot of your final result anyway, so it is good to already create the appropriate plotting routine.

Python ships with a very powerful plotting library called `matplotlib`. The code snippet in listing 1 uses this library to make a plot of density as a function of position (assuming you made a list called `cells` which contains `Cell` objects which have variables `_midpoint` and `_density`). You can save the plot as a PNG image file by replacing `plt.show()` with `plt.savefig("FILENAME.png")`.

Listing 1: Sample plot script.

```
# lines that start with a '#' are comment lines,
# which are ignored by the Python interpreter
# you can also use a '#' in the middle of a line,
# in this case everything after the '#' is ignored
# by the interpreter

# import the matplotlib library
import matplotlib.pyplot as plt

# the [] notation below does a loop over all cells
# and returns a new list that contains the _midpoint
# or _density of each cell
# the "k." tells matplotlib to use a black (k)
# dot (.) to plot each data point
plt.plot([cell._midpoint for cell in cells],
         [cell._density for cell in cells],
         "k.")
# show the plot: this creates a window and pauses
# the program
# program execution will resume when the window is
# closed by the user
plt.show()
```

Make a plot of the density, fluid velocity and pressure as a function of the cell position. Make sure these variables have the correct values.

Time integration

Now that you have set up the basic grid structure and have initialized its variables, it is time to move on to the actual time integration. As discussed in the lectures, this is basically a four step process:

1. Convert the conserved variables for each cell into primitive variables. This overwrites the current values of the primitive variables.

2. For each pair of neighbouring cells, use their primitive variables as input for the Riemann solver. This gives you primitive variables you can use to compute the fluxes between these neighbouring cells.

3. Still for the same pairs of neighbours: compute the fluxes.

4. Still for the same pairs of neighbours: update the conserved variables of each of the two cells using the fluxes. Note that the change in conserved variables does not affect the current step of the integration scheme, as it only uses primitive variables. The primitive variables will only be updated at the start of the next step.

These steps are detailed below.

Step 1: variable conversion

If the cell volume V is known, the primitive variables can be derived from the conserved variables using the following relations:

$$\rho = \frac{m}{V}, \tag{3}$$

$$v = \frac{q}{m}, \tag{4}$$

$$p = (\gamma - 1)\left(\frac{E}{V} - \frac{1}{2}\rho u^2\right). \tag{5}$$

This needs to be done in a separate loop over all cells, which is done before the second step. You will need the updated primitive variables during the next step, and you need all cells to be updated before you can start the next step.

Note that this conversion step is not really necessary if a fixed volume V is used; you could reformulate the fluxes to directly update the primitive variables and use primitive variables throughout. However, it is better to use a more general approach, which will still work should you decide to use your code in a Lagrangian mode in the future, since then the volumes would change and only the conserved variables stay constant between successive steps. Furthermore, not using conserved variables does not save memory, as you still need a second set of variables to keep track of the change in variables during the step. Nor is it more efficient, since the reformulated flux expressions will be more elaborate.

Step 2: Riemann solver

This step and the next two steps can all be done in a single loop over all cells, whereby each iteration of the loop deals with two neighbouring cells. One cell will act as the left state of a Riemann problem, the other as the right state. It does not matter which cell takes up which role, as long as you keep in mind that the fluxes flux out of the left state and into the right state, along a vector pointing from left to right. So your choice of left and right will affect the choice of sign in the flux update in step 4.

Listing 2 shows how you can use the provided Riemann solver in your program. Note that `RiemannSolver.solve()` returns four values: the density, velocity and pressure solution, plus an extra integer that tells you if the left state (-1), right state (+1) or a vacuum state (0) was sampled. This value is useful for solvers in 2D or 3D, but you can ignore it here (assigning the fourth return value to the variable `_` does exactly that).

Listing 2: Sample Riemann solver script.

```
# import the Riemann solver library (this only
# works if the file 'riemannsolver.py' is in the
# same folder as your own program)
import riemannsolver

# create a RiemannSolver instance for an
# adiabatic index of 5/3
solver = riemannsolver.RiemannSolver(5./3.)

# set the left and right state variables
densityL = 1.
velocityL = 0.
pressureL = 1.
densityR = 0.125
velocityR = 0.
pressureR = 0.1

# now get the Riemann problem solution
# (_ ignores the last return variable,
#  \ tells the interpreter to continue
#  reading on the next line)
densitysol, velocitysol, pressuresol, _ = \
  solver.solve(densityL, velocityL, pressureL,
               densityR, velocityR, pressureR)
```

Step 3: flux computation

The mass, momentum and total energy flux are given by

$$F_m = \rho u, \tag{6}$$

$$F_q = \rho u^2 + p, \tag{7}$$

$$F_E = \left(\frac{\gamma}{\gamma - 1} p + \frac{1}{2} \rho u^2 \right) u. \tag{8}$$

Note that to get the flux between the pair of cells in the previous step, you need to use the primitive variables that were returned by the Riemann solver in these expressions.

Step 4: conserved variables update

The conserved variables are updates using

$$m' = m - A\Delta t F_m, \tag{9}$$
$$q' = q - A\Delta t F_q, \tag{10}$$
$$E' = E - A\Delta t F_E, \tag{11}$$

where A is the direction of the flux from the point of view of the left state: $A = 1$ if the left state has a lower x position than the right state (the interface vector points in the positive direction), $A = -1$ if the left state has a higher x position. Δt is the time step, you can use $\Delta t = 0.001$ for now.

Note that every cell should receive two flux updates: one from its left neighbour, and one from its right neighbour. Be careful with the terms "left" and "right" in this case, because they can mean a different thing than the "left" and "right" state in the Riemann problem!

Repeat

Once step 4 has completed, you can start over again from step 1. A single iteration of steps 1 to 4 is called a *time step* of the integration scheme, and by performing many successive time steps, you evolve your grid forwards in time.

Implement a loop that executes steps 1 to 4. How many steps do you need to evolve your solution forward in time to $t = 0.2$?

Boundary conditions

While implementing steps 1 to 4, you will need to think about what happens at the boundaries of you simulation box: the first and last cell in your box only have one neighbouring cell, and hence only one flux contribution. You need to do something special to compute the flux through the side of the cell that has no neighbouring cell. What you need to do depends on the type of boundary conditions you impose. There are various types of boundary conditions (open boundaries, inflow/outflow boundaries...), but only two types that guarantee conservation of the conserved quantities:

1. Periodic boundaries: the simulation box is surrounded by identical copies of itself. In practice, this means the leftmost cell of the box has the rightmost cell of the box as a left neighbour, and vice versa.

2. Reflective boundaries: the boundaries of the simulation box act as a wall with which the fluid collides elastically. In practice this means the leftmost cell of the box has a ghost neighbour which has the same density and pressure as the leftmost cell, and a fluid velocity with the same magnitude but the reverse sign. The rightmost cell has a similar ghost neighbour.

Listing 3: Sample reference solution script.

```
# import the reference solution script (this only
# works if the file 'sodshock_solution.py' is in the
# same folder as your own program)
import sodshock_solution

# get the solution at a time t = 0.2
xref, densityref, uref, pref = \
  sodshock_solution.get_solution(0.2)
```

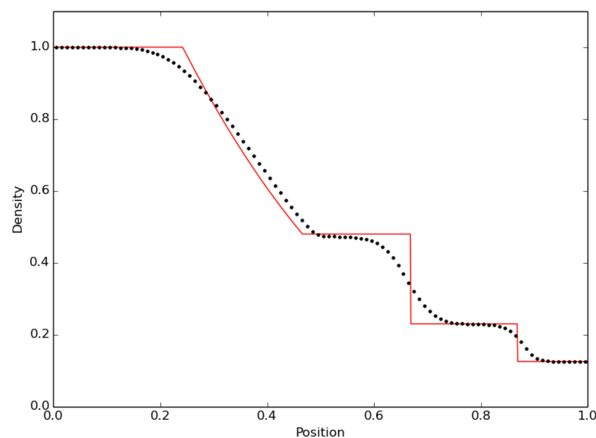

Figure 1: Result for the Sod shock problem at $t = 0.2$.

Implement reflective boundary conditions for the leftmost and rightmost cell in your grid. Since the fluid is initially at rest, nothing should happen at the boundaries until the waves reach the boundary of the box and reflect from it, which happens well after $t = 0.2$. If anything happens at the boundary before that, you did something wrong at the boundaries!

First order result

Once you have successfully implemented the time integration scheme, you should be able to solve the Sod shock problem discussed in the introduction. To see if your solution is also accurate, you can compare it with the reference solution. Listing 3 shows you how to use the reference solution script for this. If everything went well, you should get a result similar to figure 1.

Code stability

If you are sure your solver works, it is time to look at some of the details of your algorithm. Remember that you used a fixed time step for the time integration. As discussed in the lectures, the time step sets the stability of your integration. If the time step is larger than the typical time it takes the fluid to cross a single cell, your integration will become inaccurate and might lead to unphysical values.

Change the value of the time step you use in your solver from $\Delta t = 0.001$ to $\Delta t = 0.01$. What happens to your result?

At this point, it might be a good idea to add some checks to your code, to make sure that you do not end up using unphysical values. It might be very obvious to you that a mass/density and a total energy/pressure are always positive values, but there is no guarantee that this will also be the case in your code. After all, you are subtracting fluxes from you mass and total energy values!

Similarly, you are dividing your masses by volumes and your momenta by masses, quietly assuming that these volumes and masses are always positive and non-zero. What would happen to these operations if this turned out not to be the case?

Go through your code again and identify the operations that might cause masses, densities, total energies or pressures to become negative, or operations where you might divide by zero. Add `if` statements to check occurences of these events, and make sure your code crashes when this happens (displaying a useful error message in the process).

This should help you understand what happens when the integration time step is too large.

Extensions to your basic code

At this point, you have a fully functional 1D hydrodynamical solver, and you have reached the target of this exercise. However, there are lots of extra things you can do with your code:

- You can implement a time step criterion which automatically computes a safe time step for the next step, based on the primitive variables of the cells.
- You can extend your first order scheme and make it second order in space and time. This means you will get a more accurate result for the same number of cells. Doing this requires the computation of gradients for the primitive variables in each cell, and an extra step that uses these gradients to extrapolate the primitive variables from the midpoint of the cell to the position of the interface between neighbouring cells, before the Riemann solver call.

- You can relax the requirement that your grid is static, by letting the midpoints of the cells move with the fluid velocity (as so-called moving mesh scheme). This means that your cell volumes will change after every time step, and also that the interfaces between neighbouring cells are actually moving, which means you need to add a correction term to your fluxes. This should increase the accuracy of your solution around the position of the shock wave.

- You could extend your code to problems in 2D and 3D. Although it might seem unlikely, the 2D and 3D algorithms are very similar to this basic 1D algorithm. You only need to make sure you solve the Riemann problem in a reference frame where the x-axis aligns with the surface normal of the interface between neighbouring cells, and decompose your 2D or 3D flux vector into its coordinate components. And of course the neighbour relations between cells become a little bit more complex. And last but not least, the run time of your code tends to increase a lot if you add a dimension, so it might be a good idea to exchange Python for a high-level programming language if you plan to write a 2D or 3D code.

Most of these improvements require a lot of extra work, which is outside the scope of this exercise. But if you are really interested, please try them out, and do not hesitate to contact me (`bv7@st-andrews.ac.uk`) if you have any questions!