PLUTO simulation and imaging procedures

Jonathan Rogers

March 1, 2017

This document outlines the general processes required to simulate a 2D astrophysical jet using PLUTO. It also provides general instructions on imaging this simulation using python.

Note that it is assumed that the user has python 2.7 installed. Python 3 is not compatible pyPLUTO, a python module that comes with PLUTO.

1 PLUTO

PLUTO is a user-friendly finite-volume/area, shock-capturing code that was written (in C) to integrate a set of conservation laws such as the hydrodynamical equations. There are four inbuilt physics modules allowing for hydrodynamical, magnetohydrodynamical, special relativistic hydrodynamical, and relativistic magnetohydrodynamical physics to be investigated.

After the initial installation (which is beyond the scope of this document), the code requires 4 files to run: init.c, definitions.h, pluto.ini and an appropriate makefile for your OS. The following sections summarise the role of these files, and give a brief how-to on setting up your own simulation with an emphasis on producing astrophysical jets. The latter sections discuss imaging your simulation using python. A more in-depth guide to PLTUO can be found in the userguide at http://plutocode.ph.unito.it/files/userguide.pdf.

Euler equations:
conservation of mass: conservation of momentum conservation of energy In

conservative form:

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ m_1\\ m_2\\ m_3\\ E\end{pmatrix} = -\nabla\cdot\begin{pmatrix}\rho\vec{u}\\ m_1\vec{u}+P\vec{E}_1\\ m_2\vec{u}+P\vec{E}_2\\ m_3\vec{u}+P\vec{E}_3\\ (E+P)\vec{u}\end{pmatrix} - \begin{pmatrix}0\\ \rho\frac{\partial}{\partial x_1}\Phi\\ \rho\frac{\partial}{\partial x_2}\Phi\\ \rho\frac{\partial}{\partial x_3}\Phi\\ \rho\vec{u}\cdot\nabla\Phi\end{pmatrix} \tag{1}$$

m is the momentum density (rho $\vec{u}\vec{u}$). Which is of the form

$$\frac{\partial}{\partial t}\vec{U} = -\nabla\cdot T(\vec{U}) - S(\vec{U}) \tag{2}$$

is this a rieman problem?

1.1 Simulation set up

The three important files that govern the behaviour of your simulation are the definitions.h, pluto.ini and init.c files. These files define things such as:

- Complexity of the physics
- resolution
- input parameter values
- Grid print timestep size
- simulation run time
- user defined boundary conditions

The files are not independent of each other - that is, there are options that are 'switched on' in one file, but given a value in another. The details of important parameters in these files are discussed below. Note that not all parameters and flags are discussed, only those needed to set up a simple HD, 2D jet simulation.

1.1.1 definitions.h

The definitions.h file specifies:

- which fluid equations will be solved
- what Rieman solver and time stepping technique to use
- any user defined parameters

any other physics dependent parameters (most of which won't be discussed) such as:

- the equation of state
- inclusion of entropy in conservation laws

The definitions.h file can be configured by running the setup.py script (located in the PLUTO/ folder by default). However, there are some parameters (namely those which are physics dependent) that must be edited manually.

Physics and geometry
Most parameters at the beginning of the definitions.h file are somewhat self-explanatory.

- DIMENSIONS determines the number of dimensions in your simulation. For 2D simulations, trivially the DIMENSIONS parameters takes the integer value 2.
- COMPONENTS is the total number of vector components of the simulation. For example, if ignoring magnetic fields, a 2D simulation may only have a velocity field (which is only a function of (x,y)), and therefore COMPONENTS should be set to 2. Generally COMPONENTS = DIMENSIONS.
- GEOMETRY specifies the geometry of your simulation and requires a string. Each spatial coordinate is labelled as x_1, x_2, and x_3 respectively, and can take one of four values:
    - Cartesian
    - Cylindrical

- Polar

- Spherical The importance of selecting the most appropriate geometry is discussed further in section 2.

- PHYSICS is used to select which physics module you want to use. The available modules are given in section 1. Setting this parameter to HD invokes classical hydrodynamics as described by the Euler equations (see section 6.1 of the user guide for more information).

- BODY_FORCE is a switch that enables for a body force term to be included in the momentum and energy equations. It can be input as a scalar potential, or expressed as a vector. This will be discussed further in 'the init.c section'. This is an important flag as it invokes self-gravity of the gas in the simulation which can drastically change the resulting morphology.

Solvers The solvers block

- RECONSTRUCTION The code works by solving the equations of fluid dynamics in vectorized form. This is done so using Godunov's theorem (getreference). This particular method is chosen as it has proved important "wikipedia: in the development of high resolution schemes for the numerical solutions to partial differential equations.". The method also guarantees a solution to the entropy equation (if convergence).

  For more on Godunov's method, see **get a reference, maybe wikipedias one?**

  During the use of Godunov's method, for time step n, the cell average of the solution vector to your system of conservation equations is calculated. The manner in which this cell average is calculated can determine how physically realistic your simulation is.

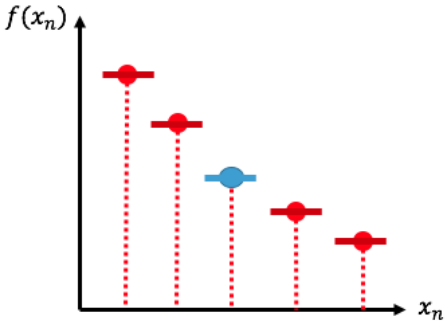
  Figure x shows the cell average solution to U (red dot) as is solved over your spatial grid. The red lines illustrate how the solution curve is reconstructed between the cells. The blue dot is the interpolated value given the different type of reconstruction schemes.

  - Figure (x a) shows a flat reconstruction. Quite clearly there are discontinuities in the solution curve. This is realistic for primitive variables that may have discontinuities present. For example, astrophysical

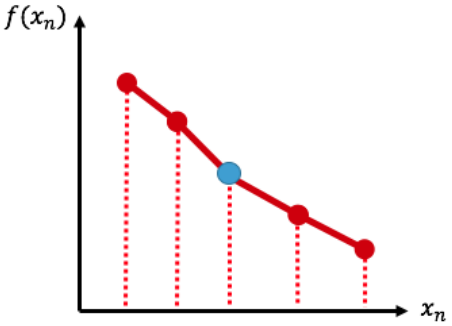
jets may show a bow shock present, characterised by a near discontinuous change in in the pressure, density, and temperate. It is the least computative expensive reconstructor.

- Figure (x b) shows a linear reconstruction. This will still model a discontinuity well, whilst better approximating continuous parts of the function.

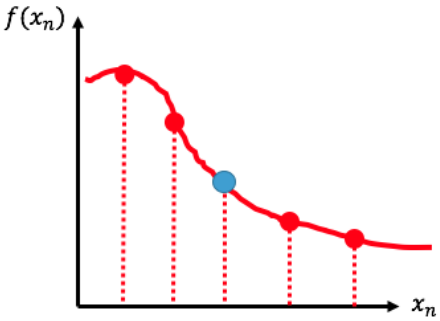
- Figure (x c) shows a parabolic interpolation. This smooths the solution curve, which should not be expected to involve a discontinuity. Parabolic interpolation gives a higher spatial accuracy of the solution over the grid. However, due to the smooth nature of the interpolation, extreme variation in values is not interpolated well.



(a) change the planes of the images to Ux vs (x1,x2) plane. change blue line to red in this one too.



(b) change the planes of the images to Ux vs (x1,x2) plane



(c) change the planes of the images to Ux vs (x1,x2) plane

The advantage of the high resolution shock capturing method that

PLUTO uses is that linear or parabolic reconstruction is used everywhere - except where discontunities are present. Discontinuous parts of the solution are instead reconstructed using a flat scheme. This ensures solutions are of higher spatial resolution where available This is set by the limiter flag.

- TIME_STEPPING The timestepping scheme used in solving the Euler equations can be set with the TIME_STEPPING parameter. The userguide outlines the pros and cons of the different combinations of recontructors and time steppers, which change depending on the geometry of your simulation. For the remainder of this document, we are going to assume a second-order Runge-Kutter timestepper, or RK2, with linear reconstruction.

Defining parameters

- NTRACER essentially adds a colour to the fluid particles. The zeroth tracer (tr0) is one that is continuously injected. Further tracer particles can be added to inject at customisable times, each being given a consecutive value in its variable name. This is useful to determine where the bulk electron population resides as a function of injection time (i.e. how is disperses), which when weighted by the pressure, is proportional to the intensity? (check this..).

  - The maximum number of tracers accepted when running setup.py is 8, any more and you will need to manually edit the definitions.h file after running setup.py. The inbuilt GUI is not optimized for more than 8 instances of tracer injections. Attempting to image such a simulation with the GUI will result in not being able to see the colour map of your solution vector.

- USER_DEF_PARAMETERS is the number of parameters that have been defined in the pluto.ini and init.c file. For more info on this parameter see sub/section x.x and x.x

- The pysics dependent declarations will, for simplicity, assume that the gas behaves as an ideal gas. This is done by setting the equation of state (EOS) to IDEAL.

- The user defined paramaters block is where each new variable that is required in the simulation is defined. The number of variables here must match the value of USER_DEF_PARAMETERS, as well as the number defined in the pluto.ini file.

- PRINT_TO_FILE , when given the value YES, outputs the log of the simulation to a file named pluto.log. This file is extremely useful for troubleshooting and should always be switched on.

1.1.2 pluto.ini

On execution of the code, pluto checks for an input file named pluto.ini. This file contains the run-time information for the simulation. The information includes the spatial domain of the grid, temporal parameters such as the simulation length and output frequency, and values for any user defined parameters.

The pluto.ini file is also where Adaptive mesh refinement (AMR) and output parameters associated with this are configured. AMR is a tool that changes the size of your grid where finer resolution is required. AMR is not discussed here.

Grid
The [Grid] section enables for resolution customisation over the simulation grid space. Finer cell spacings may be required in some areas of jet simulations. For example:

- A larger resolution is often required close to the injection point of your jet, otherwise the jet may not accurately (or at all) propagate onto the grid.

- In areas around the jet-environment boundary, a higher resolution grid is needed to be able to resolve turbulence. This is vitally important as turbulence plays a large role in momentum transfer from jets, causing differing morphologies (e.g. Perucho DATE).

The simulation grid is composed of 'grid patches'. Each grid patch may differ in:-

- Size

- Number of cells
- Size of the cells comprising the patch; or
- How the cells are distributed over the patch

For example, Figure 2 shows a 2D grid space extending out to x = 8. There are 2 patches (as indicated by the shading). Patch 1 is defied over $0 \leq x < 4$ and $0 \leq y < 4$ (resolution of 4x4).

Patch 2 has half the resolution of patch one, and covers the domain $4 \leq x < 8$, and $4 \leq y < 8$. This is useful if finer resolution is only needed on smaller scales. The computation time benefits from the smaller grid only being defined in part of the simulation.

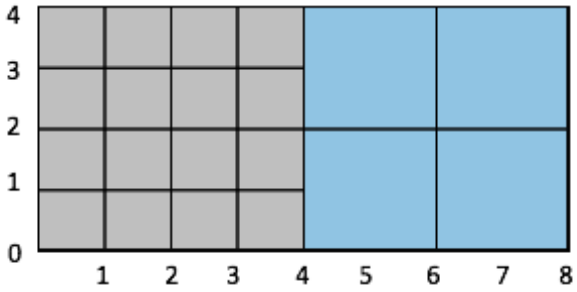


Figure 2: A mock grid with two patches, highlighted in brown and blue.

Time and Solver blocks

- Includes comes values that relate to the time stepper selected in the definitions.h file. See the userguide for more info.

- tstop is the integration stop time (i.e. simulation temporal length)

- first_dt sets the initial time step. Values that are too extreme may cause stability issues at the very beginning on the integration. As per the useguide, a typical value is 10^{-6}

see table 4.1 for appropriate solvers for your chosen physics module.

Boundary

need to figure out a way to not just copy the userguide for this bit

- mention that a reflective boundary mimics the existence of a counter jet, thus giving a more accurate simulation.

Static grid output
- Controls the output frequency and the format. - Only going to worry about the dbl line. - dbl, print step, whats -1?, single_file or multiple_files, depending on if you want all of your variables printed to the one file?

Parameters The parameters section is where values are given to the user defined parameters. Values of these parameters are used in the init.c file.

1.1.3 init.c

The initial conditions and boundary conditions of the problem are set in the init.c file. Here we briefly discuss the functions involved. The manipulation of the lower (x,y) boundary (or lower (r) boundary on a spherical grid) to include an inflow of conserved simulation quantities is used to simulate a fluid jet. Quantities involved may also be user defined parameters (values set in the pluto.ini file)

User defined parameters are assigned to variables by var = g_inputParam[PAR], where PAR is the same of the parameter as set in the definitions.h and pluto.ini files.

Init section The init "section"? (not sure what to call it) defines the initial condition of the state variables in each cell over the grid. These conditions are therefore functions of coordinates (which will vary depending on the coordinate system).

This "section"? may involve defining variables such as c_s, and the adiabatic index, γ. An example of this section is used in Section 2.

UserDefBoundary section The UserDefBoundary function is switched on if, in the pluto.ini file, a boundary value is set to USER_DEF_BOUNDARY.

It is also useful to define a pressure floor (code shit beds if negative pressures are involved) in boundary conditions. can do this using the function TOP_LOOP.

```
if (side == 0){
   TOT_LOOP(k,j,i){
```

```
      if (d->Vc[PRS][k][j][i] < 1.e-6) /* set min pressure */
      {
      d->Vc[PRS][k][j][i] = 1.e-6;
      }
   }
}
```

Top loop loops over every cell in the simulation.

What does d− > Vc actually do? What is side==0? may need to ask ross.

BOX_LOOP "The macro BOX LOOP(box,k,j,i) performs a loop over the bottom boundary zones and, for cellcentered data, it is equivalent to the macro X2 BEG LOOP(k,j,i)"

1.2 Nondimensionalization of the simulation

Two limitations of numerical simulations are computation time, and numerical integration error. Both of these factors can minimised by writing optimised code that uses less memory.

Arithimic using large numbers is both computationaly expensive and uses more memory. This issue can be addressed by normalising simulation parameters in the init.c file, named the state variables ρ, P and v_{jet}. The ratio of the length normalisation unit to the sound speed in the environment gives the physical time step of your simulation. This is therefore the unit time, corresponding to 1 in the code units (tstop in the pluto.ini? file therefore being the age of your jet). Furthermore, normalising the length scale to characteristic scales can also be useful. Typical scales are discussed below.

An example of nondimensionalization is given in section 2 (using jupyter notebook with Python 2.7).

1.2.1 Characteristic length scales of astrophysical jets

Krause et al. 2012 outlines the method and arguments leading to these length scales.

There are two bounding length scales to consider referred to as L1 and

L2. L1 is the distance from the AGN that the jet density approaches the environment density, i.e.

$$\text{L1: } \rho_{jet} = \rho_x \tag{3}$$

This is typically referred to as the inner scale. This can be expressed in terms of the jet power, Q_0, jet velocity, v_{jet} and ρ_{jet} (Alenxader 2006).

$$L1 = 2\sqrt{2}\left(\frac{Q_0}{\rho_x v_{jet}^3}\right)^{1/2} \tag{4}$$

The outer scale of a jet, or L2 is where the jet pressure is comparable to the environment pressure, i.e.

$$\text{L2: } P_{jet} = P_x \tag{5}$$

Note that the environment pressure and density could refer to the ambient medium, or the jet cocoon should one have formed. This can be expressed in terms of Q_0, ρ_x, and the soundspeed of the medium c_x (Komissarov & Falle 1998)

$$GIVEEQUATION \tag{6}$$

There exist three more length scales between L1 and L2. However, unlike the inner and outer scale, the order of the remaining three scales depend on the physics of the jet and environment, and will determine the morphological features of the jet and the surrounding medium. These three intermediate scales relate to the locations of jet collimation (L1a), cocoon formation (L1b) and jet termination (L1c). The order to these three scales change with jet morphology, and thus jet morphology can be understood via the ratios of the scales. These scales discussed should be taken as order of magnitude estimates rather than exact physical locations in simulations or observations.

L1a: Scale of jet collimation Collimation happens on scales where the sideways ram pressure of the jet starts to fall below the environmental pressure. I.e.

$$\rho_{jet} v_{jet}^2 \sin\theta < P_x \tag{7}$$

Where subscript x refers to the environment immediately outside the jet, and θ is the half opening angle of the jet.

L1b: Scale of cocoon formation The scale on which the jet density falls below the density of the environment. Should a cocoon form before the jet collimates (i.e. L1a/L1b ¡ 1), then for the purposes of L1a, the subscript x refers to the cocoon.

L1c: Scale of jet termination A jet termination shock appears on scales of L1c. At L1c, the forward ram pressure of the jet falls below the environmental density. Should the scale of L1b be smaller than L1c, then this environment refers to the cocoon.

Krause et al. 2012 discusses the derivation of these length scales in some detail, calling on work from Komissarov & Falle 1998, and Alexander 2006. However, the length scales calculated relate to an environment of constant density. Hardcastle & Krause 2013 used these length scales in numerical modelling of radio galaxy globes in cluster (non-constant) environments. This however, is beyond the goals of this tutorial.

1.3 Viewing your data

There is an pre-coded GUI that uses the pyPLUTO python module to display 2D images, or slices of your simulation output. However, the resolution at which the GUI displays the data is less than the actual resolution of the simulation (this is usually the case when the grid spacing is low enough to resolve turbulence). It is easier to get a better quality image by importing the pyPLUTO module into pluto yourself, and manually extracting, then imaging the data yourself. This section outlines how to use the pyPLUTO code, how to create images, and a small introduction to making movies of your simulations.

1.3.1 pyPLUTO module

pyPLUTO is a pluto module developed by Bhargav Vaidya and Denis Stepanovs, and is extremely useful for analysing PLUTO simulations. Within the module is the function pload, which creates a pyPLUTO object with attributes that include the output matrices of your simulation, grid information and information on the simulated quantities. The function requires two inputs:

- ns: the output file number (eg, 0 would be the output binary file data.0000.dbl)
- run_dir: the directory that your simulation resides in

For example, to load the density map of your simulation:

```
import pyPLUTO as pp

ns = 1 #define output file to analyse
run_dir = 'MyDirectory/MyPlutoSimulation/' #set run directory
curObject = pp.pload(ns,run_dir) #create pyPLUTO object
density = curObject.rho #get attribute (density in this case) from
    the pyPLUTO object
X1_AXIS = curObject.x1 #get x axis of simulation, is a vector
X2_AXIS = curObject.x2 #get y axis of simulation, is a vector
```

It is often useful to define simulations of astrophysical jets in spherical coordinates (see section x), in which case the X1_vector and X2_vector would actually represent the (R,Θ) axes.
is there ay more to say?

1.3.2 Plotting with matplotlib

Set up a grid using meshgrid

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

#Create a grid by producing a 2D array of the combination of the
    values in the two *_AXIS vectors
X1, X2 = np.meshgrid(X1_AXIS, X2_AXIS)

#Create the density map using the pcolormesh plotting function
plt.pcolormesh(X1,X2,density)

#Set axis lavels
plt.xlabel('x [code units]')
plt.ylabel('y [code units]')

#Add a colorbar
```

```
cb = plt.colorbar()

#Display the plot
plt.show()
```

Alternatively, is it better practise to write functions for getting the data and plotting it. An example of this is given in section 2.

2 PLUTO Example (github required)

The following section works through setting up and imaging a simple 2D HD simulation of an astrophysical jet using the PLUTO code, and jpython 2.7

We begin by obtaining the necessary run files using github.

Require the user to download some example files from gitgub - probably also talk about how to put your own files on github, giving special care not to prompt the user to upload their entire simulation.

Importance of geometry selection in jet simulations
- want to simulate a Fanaroff and Riley type 2 jets (FR-II). - assume a constant density profile (for simplification puroses) $10^{something}$ kg per mcubed, which is 1 in simulation units - normalise simulation by L1 - Assume a cluster temp of $10^{something}$ Kelvin, corrosponding to a sound speed of x - Derive time step, work out the physical age of the jet that we want to simulate and set tstop accordingly in the definitions file

- Want to simulate jet in spherical coordinates - Coordinate system must be taken into consideration when setting the resolution in the pluto.ini file

- Define new parameters (make a diagram) - go through the init.c file and code up each section - run the code - image the code using the code in the section before - plot in R,Theta space, then transform to cartesian and replot - should look like this: pic goes here - congrats, you've got a jet.

- the geometry and body force selection must match the coding of your simulation in the init.c file - can specify opening angle of jet in terms of theta (mention Krause et al. 2012 and how it is a morpholoy divide between jets)

-have pretty much written this example - just need to copy my python code.

2.1

TO DO ON MY AUSCOPE SHIFT :D (hopefully..) - copy jet example - download the init.c, definitions.h and pluto.ini file from github - define H_OPEN_ANG - define MACH_EXT