PLUTO simulation and imaging procedures

Jonathan Rogers

March 14, 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.

1.1 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. Python 4.2 is used for all examples.

1.1.1 Euler equations

The Euler equations are a set of hyperbolic partial differential equations that govern adiabatic fluid flow. These equations represent the conservation of mass, momentum and energy in a system, are are defined as such:

- Conservation of mass:
$$\frac{\partial}{\partial t}\rho + \nabla \cdot (\rho \vec{u}) = 0 \tag{1.1}$$
- Conservation of momentum:
$$\frac{\partial}{\partial t} + \nabla \cdot (\rho \vec{u}\vec{u}) = \rho \nabla \Phi - \nabla P \tag{1.2}$$
- Conservation of energy:
$$\frac{\partial}{\partial t} + \nabla \cdot (e_i \vec{u}) = -P \nabla \cdot \vec{u} \tag{1.3}$$

This set of equations can be written in vectorized form:

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ m_1\\ m_2\\ m_3\\ E\end{pmatrix} = -\nabla\cdot\begin{pmatrix}\rho\vec{u}\\ m_1\vec{u}+P\vec{E}_1\\ m_2\vec{u}+P\vec{E}_2\\ m_3\vec{u}+P\vec{E}_3\\ (E+P)\vec{u}\end{pmatrix} - \begin{pmatrix}0\\ \rho\frac{\partial}{\partial x_1}\Phi\\ \rho\frac{\partial}{\partial x_2}\Phi\\ \rho\frac{\partial}{\partial x_3}\Phi\\ \rho\vec{u}\cdot\nabla\Phi\end{pmatrix} \tag{1.4}$$

m is the momentum density (rho $\vec{u}\vec{u}$). e is the specific internal energy (i.e., the interal energy per unit of mass.)

$$\frac{\partial}{\partial t}\vec{U} = -\nabla\cdot T(\vec{U}) - S(\vec{U}) \tag{1.5}$$

This vectorised form is a Rieman problem.

1.2 Simulation set up files

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.

1.2.1 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.

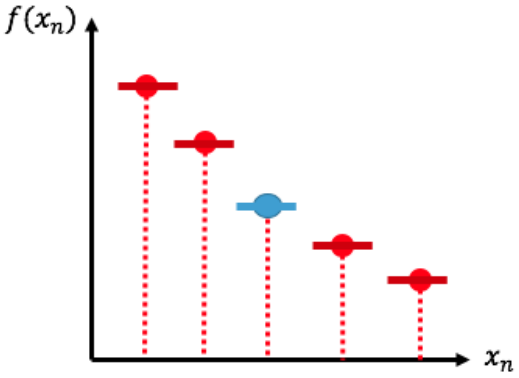
  Figure x shows the cell average solution to U (red dot) as is solved over your spatial grid. The red lines illustrate how the solution curve is reconstructed between the

cells. The blue dot is the interpolated value given the different type of reconstruction schemes.

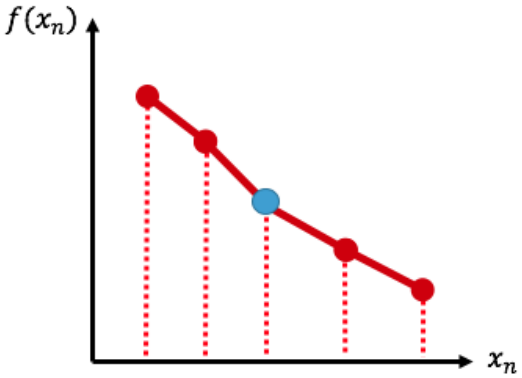
- Figure (x a) shows a flat reconstruction. Quite clearly there are discontinuities in the solution curve. This is realistic for primitive variables that may have discontinuities present. For example, astrophysical jets may show a bow shock present, characterised by a near discontinuous change in in 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.

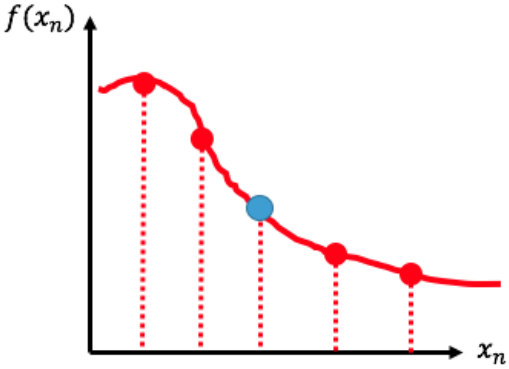
- Figure (x c) shows a parabolic interpolation. This smooths the solution curve, which should not be expected to involve a discontinuity. Parabolic interpolation gives a higher spatial accuracy of the solution over the grid. However, due to the smooth nature of the interpolation, extreme variation in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

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.

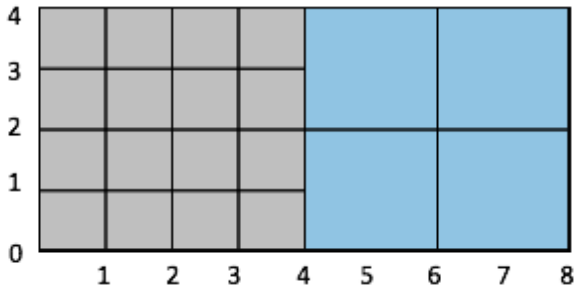


Figure 1.2: A mock grid with two patches, highlighted in brown and blue.

Time and Solver blocks

These blocks includes parameters that relate to the time stepper selected in the definitions.h file. See the userguide for more info. The two parameters we shall briefly explain are tstop and first_dt. The parameter tstop is the integration stop time (in simulation units). The conversion between code units and physical units of time enable for the customisation of jet age. The length of the initial integration step is set by the parameter first_dt. 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}.

Solvers are not explained here, but see Table 4.1 of the userguide to check what the most appropriate solver for your chosen physics module.

Boundary
The boundary block enables you to set the state of the boundary on each side of the simulation. A selection of 'userdef', usually on the X1_BEG parameter, enables for a custom boundary condition to be set in the init.c file. This is how astrophysical jets are introduced onto the grid. Setting the remainder of the injection boundary to reflective mimics the presence of a counter jet, which produces a more realistic simulation. An example of implementing a user defined boundary condition in given in section 2.

Static grid output
This section allows for the customisation of output file type, as well as the frequency at which your simulation is written to file. We are going to only use the dbl format, which is a binary file format. Setting all the other parameters to -1.0 switches them off from being written to file. Setting the dbl parameter to 1 would correspond to a file being written every 1 simulation unit. The user may also choose if every state variable value is written to it's own file, or if a every variable is written to a single 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.

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.

The syntax used to define a state variable is d-¿Vc[PRS][k][j][i], where pressure = PRS. The velocity in each coordinate can be set be replacing PRS with VX1, VX2, or VX3. The initial density is set using the variable RHO.

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 userdef. Selecting a boundary is used via the variable 'side'. For example, if side has a value of 1, then the X1 boundary is selected.

A side value of 0 allows for complete control over the solution arrays during integration (if the definitions.h file has INTERNAL BOUNDARY set to YES). Manipulation of the solution array is a useful way to define, for example a pressure floor. This technique is

best used with the TOT_LOOP function. This functions simply cycles through each cell in the simulation.

Setting a pressure floor of, for example, 10^{-6} may be done so in the following manner:

```
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;
      }
   }
}
```

The 'd→ Vc' simply points to the location in memory that struct Vc exists in. If the array PRS in this struct has any values $< 10^{-6}$, then that valuue is set to 10^{-6}.

The user defined boundary values use a similar function, the BOX_LOOP function. This function loops through cell values the bottom of the boundary zones. When used in conjunction with the side variable, the user may select a boundary cell to change the value of. An example of using the BOX LOOP function to set a boundary condition is shown in Section 2.

1.2.2 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).

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.

$$\text{L1: } \rho_{jet} = \rho_x \tag{1.6}$$

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).

$$L1 = 2\sqrt{2}\left(\frac{Q_0}{\rho_x v_{jet}^3}\right)^{1/2} \tag{1.7}$$

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

$$\text{L2: } P_{jet} = P_x \tag{1.8}$$

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)

$$L_2 = \left(\frac{Q_0}{\rho_x c_x^3} \right)^{1/2} \tag{1.9}$$

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.

$$\rho_{jet} v_{jet}^2 \sin\theta < P_x \tag{1.10}$$

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.

1.2.3 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.

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?

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.

2.1 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, using git clone `https://github.com/JGR89/PLUTO_EXAMPLE_FILES.git` to clone the repository.

An example of an FR-II jet will be produced using a density profile that declines as a function of r^2. The simulation must therefore be defined in spherical coordinates. This is already consistent with the definitions.h and pluto.ini file, but will be discussed in more depth.

2.1.1 Jet parameters

The physical parameters that we choose for the jet are those given in Hardcastle & Krause 2013. Calculation of these values can be found in the accompanying ipython notebok. These initial values are:

- Power, Q = 10^{38}W
- Density (at r = 0), $\rho_0 = 3 \times 10^{-23} \frac{kg}{m^3}$
- Cluster temperature $T = 2.3 \times 10^7 K$
- Sound speed, $c_s = \sqrt{\frac{\gamma k_b T}{\mu m_H}} = 730 \frac{km}{s}$, where μ is the average atomic weight of the particles, and m_H is the mass of a hydrogen atom. μ is given a value of 0.6 (why?)
- Mach number of the jet is assumed to be 25
- the opening angle of the jet, θ_{jet} is assumed to be 15°

The cluster density is further assumed to be given by a King Profile:

$$\rho = \rho_0 \left[1 + \left(\frac{r}{r_c} \right)^2 \right]^{\frac{-3\beta}{2}} \tag{2.1}$$

(where $\beta = \frac{\mu m_p \sigma^2}{kT}$). This example will choose a value of $\beta = 0.35$. See HK13 for further information.

The the body force term is $\nabla\Phi = \frac{-\log\rho}{\gamma}$. As this is calculated as a potential rather than in vector form, this requires the BODY_FORCE switch in the definitions.h file must

be set to POTENTIAL. The init.c file must also be coded to return $\frac{-log(\rho)}{\gamma}$ when the BodyForcePotential function is called. More on this in Section 2.2.

The grid block in the pluto.ini file reflects a quarter quarter of a circle. The $x_2 = \theta$ coordinate therefore has a range of [0,π/2]. We also define 2 Patches in the r coordination, and 2 Patches in the θ coordinate.

2.1.2 Coding the pluto.ini file

```
[Grid]

X1-grid   2    0.1    120    u    1.5    120    u    ###
X2-grid   2    0.0    240    u    0.785  120    u    1.571
```

With X3 being arbitrary in this 2D simulation.

The time block has t_stop to be 50, corresponding to 143.5 Myrs (2.87 Myr per time step). The initial dt is set to 10^{-6}

```
[Time]

CFL             0.4
CFL_max_var     1.1
tstop           50.0
first_dt        1.e-6

[Solver]

Solver       hllc
```

The boundary block is coded to introduce a jet onto the (r,θ) grid from the lower r boundary cell. This will be implemented in the init.c file by adjusting the UserDefBoundary function.

X1_end is set to reflective as this mimics a counterjet (Hardcastle & Krause 2013). To make use of the spherical coordinate system, the X2_beg cell is set to be axisymmetric (rotated about the X1 axis in a cartesian projection), and the X2_end cell to be eqtsymmetric, thus rotating the simulation about the X2 axis.

```
[Boundary]

X1-beg       userdef
X1-end       reflective
X2-beg       axisymmetric
X2-end       eqtsymmetric
X3-beg       outflow
X3-end       outflow
```

The static grid output block will be used to specify that a dbl file (binary file) should be printed every 0.5 timesteps. Every printed variable is also to be located in a single file.

The remaining lines in the static grid are to be left as -1 to suppress printing. The log keyword

```
[Static Grid Output]

uservar   0
dbl       0.5  -1   single_file
flt      -1.0  -1   single_file
vtk      -1.0  -1   single_file
tab      -1.0  -1
ppm      -1.0  -1
png      -1.0  -1
log       10
analysis -1.0  -1
```

The parameters section defines the four input variables required for the simulation:

- H_OPEN_ANG is the opening angle of the jet in degrees
- MACH_EXT is the Mach number of the jet
- R_CORE in the core radius term for the density profile (King Profile)
- B_EXPONENT changes the steepness, and distance at which the density profile declines

```
[Parameters]

H_OPEN_ANG          35.0
MACH_EXT            25.0
R_CORE              40.0
B_EXPONENT          0.38
```

2.1.3 Coding the init.c file

Init functon

UserDefBoundary

```
theta_rad = g_inputParam[H_OPEN_ANG] * CONST_PI / 180.0;

if (side == X1_BEG){   /* -- select the boundary side -- */
    BOX_LOOP(box,k,j,i){ /* -- Loop over boundary zones -- */
        if (theta[j] <= theta_rad){
            /* -- set values for r <= theta_rad -- */
            d->Vc[RHO][k][j][i] = vj[RHO];
            d->Vc[VX2][k][j][i] = 0.0;
            d->Vc[VX1][k][j][i] = vj[VX1];
            d->Vc[PRS][k][j][i] = vj[PRS];
            }
        else{ /* -- reflective boundary for r > theta_rad --*/
            d->Vc[RHO][k][j][i] = d->Vc[RHO][k][j][2*IBEG - i - 1];
```

```
            d->Vc[VX1][k][j][i] = d->Vc[VX1][k][j][2*IBEG - i - 1];
            d->Vc[VX2][k][j][i] = -d->Vc[VX2][k][j][2*IBEG - i - 1];
            d->Vc[PRS][k][j][i] = d->Vc[PRS][k][j][2*IBEG - i - 1];
        }
    }
}
```

The cell in which the jet is implemented in a boundary condition is located using side == X1_BEG (i.e., the beginning of the r axis).

The BOX_LOOP function is then used to loop over the boundary zone of in each cordinate, and change the values of RHO, VX1, and PRS (More information on the BOX_LOOP function is given in Section 1.2.3.). The boundary cells which are looped over are constrained for all cells with theta coordinates less than some value theta_rad. This is how the opening angle of the jet is defined. The variable is initialised in the definitions.h fie, and given a via the pluto.ini file. This simulation uses a value of 15°, consistent with Hardcastle & Krause 2013.

The reflective boundary condition for the rest of the boundary is set using the syntax presented in the else statement. 'what does it do?'

BodyForcePotential The BodyForcePotential() function allows for the inclusion of a body force term in the hydrodynamics equations, as discussed in section X.

Assume that a spherical gas cloud is in hydrostatic equilibrium. If the cloud is stable, then gravitational force is unable to overcome the internal pressure, and

$$-\frac{G\rho(r)M_{cloud}(r)}{r^2} = \frac{dP}{dr} \tag{2.2}$$

Therefore the acceleration that a particle would feel can be expressed as

$$-\frac{GM_{cloud}(r)}{r^2} = \frac{1}{\rho(r)}\frac{dP}{dr} = a \tag{2.3}$$

The sound speed is given by

$$c_s^2 = \frac{\gamma P}{\rho} \implies P = \frac{c_s^2 \rho}{\gamma} \tag{2.4}$$

However, in simulation units, we set the sound speed c_s to be 1

$$\therefore \rho = \gamma P \tag{2.5}$$

Therefore, (15) can be written as

$$\frac{1}{\rho(r)}\frac{dP}{dr} = \frac{1}{\rho}\frac{d}{dr}\left(\frac{\rho}{\gamma}\right) = \frac{1}{\gamma}\frac{1}{\rho}\frac{d}{dr}\rho(r) \tag{2.6}$$

And from the chain rule

$$\frac{d}{dr}\ln\rho(r) = \frac{1}{\rho}\frac{d}{dr}\rho(r)$$

$$\therefore \frac{1}{\rho(r)}\frac{dP}{dr} = \frac{1}{\gamma}\frac{\partial}{\partial r}\ln\rho = \frac{\nabla \ln\rho}{\gamma}$$

for a potential in spherical coordinates. We therefore code this potential in each of the spatial coordinates in the BodyForcePotential function in the init.c file.

```
double BodyForcePotential(double x1, double x2, double x3){

double rho, g_gamma, r, potential;
g_gamma = 5.0/3.0;
r = x1;

rho = pow(1 + pow(r/g_inputParam[R_CORE],2),-1.5*g_inputParam[B_EXPONENT]);
potential = -log(rho)/g_gamma;
return potential
}
```

2.1.4 Running the simulation

With the 3 files coded, we first need to change the makefile.

On a mac, using your preferred method, edit your .bashrc profile with an alias to initiate the setting up of pluto with a command line keyword. In the code below, the keyword 'sp' is assigned to do this.

```
alias sp='/usr/local/bin/python2.7 /Users/Jonathan/Documents/PLUTO/setup.py'
```

On a mac, run setup.py using your alias from your working directory that contain the files obtained from github. This will produce the PLUTO setup page (Figure 3)

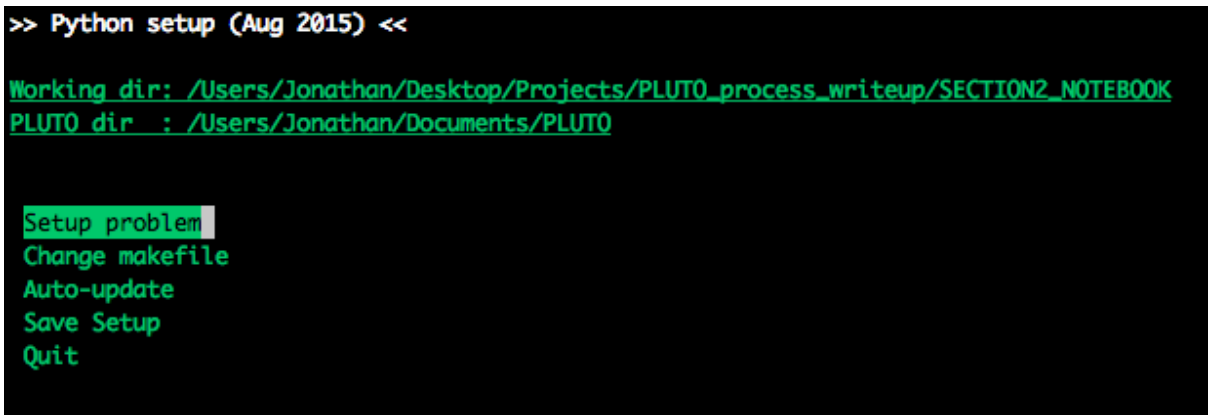


Figure 2.1: The PLUTO menu

The 'Setup problem' selection may first be run - however, each screen may just be entered through for this example as each value was manually set.

The appropriate makefile must be selected for your system. For Macs, this may be Drawin.gcc.defs, or Darwin.mpicc.defs. The gcc.defs extension indicates that this c compiler must be available on your system? Similarily, the mpicc.defs extension indicates that mpicc must be avaliable on your system. Mpi allows for parallel computing, and can

tus utalize a greater number of processors. This example chooses mpicc.defs, as it allows for the specification of number of cores to use when executing PLUTO.

Now exit the setup page, and from your working directory execute the following command in terminal to run the code:

```
mpirun -np # ./pluto
```

As this is a mach 25 jet, the simulation may time some time to run. Should you wish to gain an output immediately, the Mach number can be changed to a lower value. Note however that the Mach number enters the L1 scale through the jet velocity, and thus some scale sof your simulation will change.

2.1.5 PLUTO Imaging example

The following imaging example uses the method outlined in Section 0.2.3 (fornow). The accompanying ipython notebook was obtained from the github page when the initialisation files were obtained.

Note that the simulation can be sped up by lowering the Mach number, and making the grid resolution larger.