PLUTO v. 4.0 (November 2012)

User's Guide

(http://plutocode.ph.unito.it)

Developer: A. Mignone[1,2] (mignone@ph.unito.it)

Contributors: C. Zanni[2] (AMR) (zanni@oato.inaf.it)
P. Tzeferacos[1] (Viscosity, MHD, STS, Finite-Difference)
(petros.tzeferacos@ph.unito.it)

G. Muscianisi[3] (Parallelization, I/O)
T. Matsakos[4] (Resistivity, Thermal Conduction, STS)
O. Tesileanu[5] (Cooling)
B. Vaidya[6] (pyPLUTO visualization tool)

1 Dipartimento di Fisica Generale, Turin University, Via P. Giuria 1 - 10125 Torino (TO), Italy
2 INAF Osservatorio Astronomico di Torino, Via Osservatorio, 20 10025 Pino Torinese (TO), Italy
3 Consorzio Interuniversitario CINECA, via Magnanelli, 6/3, 40033 Casalecchio di Reno (Bologna), Italy
4 CEA, IRAMIS, Service Photons, Atomes et Molècules, 91191 Gif-sur-Yvette, France
5 Department of Physics, University of Bucharest, Str. Atomistilor nr. 405, RO-077125 Magurele, Ilfov, Romania
6 School of Physics and Astronomy, University of Leeds, Leeds LS29JT

Terms & Conditions of Use

Contents

0. Quick Start

0.1 Downloading and unpacking PLUTO

PLUTO can be downloaded from http://plutocode.ph.unito.it under free registration. Once downloaded, extract all the files from the archive:

```
~> gunzip  pluto-xx.tar.gz
~> tar xvf pluto-xx.tar
```

this will create the folder `PLUTO/` in your home directory. At this point, we advise to set the environment variable PLUTO_DIR to point to your code directory. Depending on your shell (e.g. `tcsh` or `bash`) use either one of

```
~> setenv PLUTO_DIR /home/user/PLUTO # if you're using the tcsh shell, or
~> export PLUTO_DIR=/home/user/PLUTO # if you're using the bash shell.
```

0.2 Running a simple shock-tube problem

PLUTO can be quickly configured to run one of the several test problems provided with the distribution. Assuming that your system satisfies all the requirements described in the next chapter (i.e. C compiler, Python, etc..) you can quickly setup **PLUTO** in the following way:

- change directory to any of the test problems under PLUTO/Test_Problems, e.g.

  ```
  ~> cd $PLUTO_DIR/Test_Problem/HD/Sod
  ```

- run the Python script using

  ```
  ~/PLUTO/Test_Prob/HD/Sod> python $PLUTO_DIR/setup.py
  ```

  and select "Setup problem" from the main menu, see Fig. 1.2. You can confirm (by pressing Enter) or modify the default setting using your arrow keys.

- Once you return to the main menu, select "Change makefile", choose a suitable makefile (e.g. Linux-i686.gcc.defs) and press enter.

  All the information relevant to the specific problem should now be stored in the four files init.c (assigns initial condition and user-supplied boundary conditions), pluto.ini (sets the number of grid zones, Riemann solver, output frequency, etc.), definitions.h (specifies the geometry, number of dimensions, interpolation, time stepping scheme, and so forth) and the makefile.

- exit from the main menu ("Quit" or press 'q') and type

  ```
  ~/PLUTO/Test_Prob/HD/Sod> make
  ```

  to compile the code.

- you can now run the code by typing

  ```
  ~/PLUTO/Test_Prob/HD/Sod> ./pluto
  ```

  At this point, **PLUTO** reads the initialization file pluto.ini and starts integrating. The run should take a few seconds (or less) and the integration log should be dumped to screen.

Data can be displayed in a number of different ways. If you have, for example, Gnuplot (v 4.2 or higher) you can display the density output from the last written file using

```
gnuplot> plot "data.0001.dbl" bin array=400:400:400 form="%double" ind 0
```

where `ind 0,1,2` may be used to select density, velocity or pressure. If you have IDL installed on your system, you can easily plot the density by[1]:

```
IDL> pload,1
IDL> plot,x1,rho
```

The IDL procedure `pload` is provided along with the code distribution.

0.3 Running the Orszag-Tang MHD vortex test

- change directory to PLUTO/Test_Problems/MHD/Orszag_Tang,
- run the Python script:

  ```
  ~/PLUTO/Test_Problem/MHD/Orszag_Tang> python $PLUTO_DIR/setup.py
  ```

  select "Setup problem" and confirm the default setting by pressing enter;
- Once you return to the main menu, select "Change makefile" and choose a suitable makefile (e.g. default.defs) and press enter.
- exit from the main menu ("Quit" or press 'q'). Edit pluto.ini and, under the *[Grid]* block, lower the resolution from 512 to 200 in both directions (**`X1-grid`** and **`X2-grid`**). Change **`single_file`**, in the "dbl" output under the *[Uniform Grid Output]* block, to **`multiple_files`**.

  Last, edit definitions.h and change `PRINT_TO_FILE` from *`YES`* to *`NO`*.
- compile the code:

  ```
  ~/PLUTO/Test_Problem/MHD/Orszag_Tang> make
  ```
- If compilation was successful, you can now run the code by typing

  ```
  ~/PLUTO/Test_Problem/MHD/Orszag_Tang> ./pluto
  ```

  At this point, **PLUTO** reads the initialization file pluto.ini and starts integrating. The run should take a few minutes (depending on the machine you're running on) and the integration log should be dumped to screen.

You can display data (e.g. density) with Gnuplot (v 4.2 or higher) from the last written file using

```
gnuplot> set pm3d map     # set map style drawing
gnuplot> set palette gray # set color to black and white
gnuplot> splot "data.0001.dbl" bin array=200x200 format="%double"
```

If you have IDL installed, you can easily display pressure from the last written output files with

```
IDL> pload,1
IDL> display,x1=x1,x2=x2,prs
```

Several other visualization options are described in more details in §6.3.

[1]You need to include PLUTO/Tools/IDL into your IDL search path, §6.3.1

0.4 Setting up your own test problem

As an illustrative example, we show how **PLUTO** can be configured to run a 2D Cartesian hydrodynamic blast wave from scratch. We assume that you have already followed the steps in §0.1.

- First, in your home or work directory, you need to create a folder which will contain the necessary files for the test. For instance,

  ```
  ~> mkdir Blastwave
  ~> cd Blastwave
  ```

- You can now start the setup process by invoking the Python script to set dimensions, geometry, numerical scheme and so on:

  ```
  ~/Blastwave> python $PLUTO_DIR/setup.py
  ```

  and select "Setup problem" from the main menu.

  Using the arrows keys make the following changes: set "DIMENSIONS" and "COMPONENTS" to 2, "USER_DEF_PARAMETERS" to 3 and leave the other fields as they are. User-defined parameters will be used later in the initial condition routine. Press enter to confirm the changes and proceed to the following screen menu. Since we don't have to change anything here you can press enter once more.

- We now set the names of the 3 auxiliary parameters previously introduced. To do so, use the arrow keys to select each of them and explicitly write their names: P_IN, P_OUT and GAMMA and press enter to confirm.

- Finally, we complete the python session setting the architecture for the makefile. In the makefile menu choose your system configuration (e.g. Linux_i686.defs for Linux). Press enter to confirm.

You are now done with the Python script and can exit by pressing either "q" or selecting quit. At this point you should find the following four files inside your Blastwave folder: definitions.h, init.c, makefile, pluto.ini, sysconf.out

Next, we need to edit the two files pluto.ini and init.c. The first one defines the computational domain and certain properties of the run (i.e. time of integration, first timestep etc). The second one sets the initial conditions for the blast wave problem: a circular region of high pressure in a lower pressure ambient.

Edit pluto.ini to make the following changes:

- The domain should span from -1 to 1 in both dimensions with 200 points in each direction.

  ```
  X1-grid    1    -1.0    200    u    1.0
  X2-grid    1    -1.0    200    u    1.0
  ```

- The simulation should stop when time reaches 0.04:

  ```
  tstop            0.04
  ```

  with the first timestep being

  ```
  first_dt         1.e-6
  ```

  Save the files every t=0.004, in double precision and in multiple_files format.

  ```
  dbl        0.004  -1   multiple_files
  ```

- At the end of the file, set the numerical values for the 3 parameters P_IN (the high pressure of a region yet to be specified), P_OUT (the ambient pressure) and GAMMA (polytropic index):

```
P_IN    8.e2
P_OUT   8.0
GAMMA   1.666666666666667
```

Save and exit the editor.

Next, you need to edit init.c.

- Define inside the function `Init()` the radius r, a floating point value which we will be used to set a circular region of high pressure.

  ```
  double r;
  ```

- Set the global variable `g_gamma` (polytropic index) and the radius r. Define the initial ambient pressure (P_OUT) and put an IF statement to specify the high pressure region inside a circle of r= 0.3 (P_IN):

```
g_gamma = g_inputParam[GAMMA]; /* calls  the auxiliary parameter GAMMA*/
r = x1*x1 + x2*x2;
r = sqrt(r);

us[RHO] = 1.0;        /* initial density array */
us[VX1] = 0.0;        /* initial Vx array */
us[VX2] = 0.0;        /* initial Vy array */
us[VX3] = 0.0;        /* initial Vz array */
us[PRS] = g_inputParam[P_OUT]; /* calls  the auxiliary parameter P_OUT */

if (r <= 0.3) us[PRS] = g_inputParam[P_IN]; /* calls the input parameter P_IN */
```

Save and exit the editor. Compile the code and run **PLUTO** with a the following set of commands:

```
~/Blastwave> make
~/Blastwave> ./pluto
```

In order to visualize the results follow the instructions described in the two previous sections.

0.5 Supplied test problems

Several examples and test problems may be found under PLUTO/Test_Problems/.

- HD/Sod: the Sod shock-tube problem;
- HD/Sedov: the Sedov-Taylor blast wave problem in either cylindrical or spherical geometries;
- HD/Rayleigh_Taylor: a 2D setup of the Rayleigh-Taylor instability;
- HD/Wind_Tunnel: a Mach 3 wind tunnel with a step;
- HD/Vortex: advection of an isentropic vortex;
- HD/Mach_Reflection: reflection of a strong shock on a wedge;
- HD/Jet: a simple two-parameter jet configuration in axisymmetric coordinates;
- HD/Disk_Vortex: vortex dynamics in a 2D Keplerian disk in polar coordinates;
- HD/Stratified_Atmosphere: hydrostatic equilibrium atmosphere in 2D cylindrical or 3D Cartesian coordinates using a geometrically-smoothed point-mass gravitational field;
- HD/Viscosity/Taylor_Couette: Taylor vortex formation in an axial flow between two rotating cylinders.
- MHD/Orszag_Tang: the Orszag-Tang MHD vortex problem;
- MHD/CP_Alfven: circularly polarized Alfven waves;

- MHD/Field_Loop: advection of a field loop in a periodic domain;
- MHD/Rotor: the 2D MHD rotor problem;
- MHD/Jet: a Mach 20 magnetized jet in cylindrical axisymmetric coordinates;
- MHD/Shock_Cloud: interaction of a strong magneto-sonic shock with a circular cloud;
- MHD/Shearing_Box: an example of the two-dimensional magneto-rotational instability using the shearing box module;
- MHD/Resistive_MHD/Field_Diffusion: a 3-D test for magnetic field diffusion;
- MHD/Torus: a magnetized accreting torus in 2.5D spherical/cylindrical coordinates or 3D Cartesian;
- MHD/Thermal_conduction/TCfront: propagation of a thermal conduction front;
- MHD/Thermal_conduction/Blast: blast wave with thermal conduction
- RHD/Sedov2D: relativisitc blast wave in 2D Cartesian coordinates;
- RMHD/KH: an example of a relativistic Kelvin-Helmholtz instability;
- RMHD/Blast: a two- or three-dimensional magnetized relativistic blast wave;
- RMHD/Toroidal_Jet: axisymmetric setup for a relativistic jet with a toroidal magnetic field;

Each test problem has (usually) more than one configuration set with a different Riemann solver, numerical scheme, etc... They are stored in pluto_nn.ini, definitions_nn.h, where $nn = 01, 02, ...$; just copy the two files to pluto.ini and definitions.h and run the Python script to generate the corresponding makefile.

0.6 Migrating from PLUTO 3 to PLUTO 4

Users familiar with **PLUTO** version 3 should provide a few modifications in order to upgrade to the current release. The most important changes are listed below.

- Naming convention has been largely revised in order to adhere to a more consistent and better orgainzed programming style. This resolves the mixed-up confusion between function, global variable and macro names present in previous versions of **PLUTO** . In particular:

  1. Function names have been changed from all capital letter style to upper CamelCase style, e.g., **`FUNCTION_NAME()`** → **`FunctionName()`**. This affects the whole code and, in particular, also the functions contained inside init.c. For instance:

     `INIT()` → **`Init()`**
     `ANALYSIS()` → **`Analysis()`**
     `USERDEF_BOUNDARY()` → **`UserDefBoundary()`**

     The argument list inside the previous functions has also been changed, see §2.4 for details.

  2. Macro names retain the capital letter style, e.g.,

     `dmin()` → **`MIN()`**
     `dmax()` → **`MAX()`**
     `dsign()` → **`DSIGN()`**
     `Array_2D()` → **`ARRAY_2D()`**

     Besides, macro names giving the array index of a variable now use a three-letter word:

     - `DN` → `RHO`
     - `PR` → `PRS`
     - `VX` → `VX1`
     - ...

     See Table 2.4 in §2.4.1. However we still keep the old two-letter notation (e.g. "`DN`" or "`PR`") for backward compatibility.

  3. Global variables have been renamed using lower camelCase style and are prefixed with "`g_`", see the header file globals.h. For instance:

     `gmm` → `g_gamma`
     `aux` → `g_inputParam`
     `C_ISO` → `g_isoSoundSpeed`
     ...

     The only exception is for integer global variables that are initialized once at the beginning of the computations (e.g. number of points `NX1_TOT...NX3_TOT`, starting and final indices `IBEG..KEND`, and so forth) and do not change anymore during the computation.

  4. Variable names referring to the number of points and using the notation "`X,Y,Z`" have been replaced with the more general syntax "`X1, X2, X3`" whenever possible. Similarly, variables giving grid indices use the notation "`I, J, K`":

     (`NX, NY, NZ`) → (`NX1, NX2, NX3`);
     (`NX_TOT, NY_TOT, NZ_TOT`) → (`NX1_TOT, NX2_TOT, NX3_TOT`);
     (`NX_PT, NY_PT, NZ_PT`) → (`g_i, g_j, g_k`);

- **PLUTO** 4.0 uses Doxygen as the standard documentation system which, from now on, is meant to replace the old Developer's guide. The API reference guide, although not complete, can be found on the web or in the local distribution under PLUTO/Doc/Doxygen/html/index.hml.

- The **`BODY_FORCE()`** function has been replaced by two new functions, **`BodyForceVector()`** and **`BodyForcePotential()`** (see §2.4.3) included inside the file init.c:

- Limiters are no longer functions and can be specified in your definitions.h header by macro names with upper-case letter, e.g., `minmod_lim` → `MINMOD_LIM`, etc...
- ArrayLib has been removed as a separate library and a more compact, largely debugged subset has been directly incorporated into the code, under the directory Src/Parallel/.
- The structure of the system-dependent configuration file used by the makefile is different, see §2.2.
- I/O has slightly been modified in the following ways:
  - variable names follow the same three-letter patterns used above;
  - the output grid file grid.out employs a different format, see §6.1.6.
- The command line switch `-restart` always requires the restart file number, `-restart n`.
- Assignment of initial condition from external files uses a different, more flexible approach, §2.4.1.1.
- Visualization routines written in the IDL programming language follows the same naming convention adpoted by the code.
- Chombo 3.1 is required for Adaptive Mesh Refinement, see Chapter 7.

1. Introduction

PLUTO is a finite-volume / finite-difference, shock-capturing code designed to integrate a system of conservation laws

$$\frac{\partial \boldsymbol{U}}{\partial t} = -\nabla\cdot\mathsf{T}(\boldsymbol{U}) + \boldsymbol{S}(\boldsymbol{U})\,, \tag{1.1}$$

where $\boldsymbol{U}$ represents a set of conservative quantities, $\mathsf{T}(\boldsymbol{U})$ is the flux tensor and $\boldsymbol{S}(\boldsymbol{U})$ defines the source terms [28, 29]. An equivalent set of primitive variables $\boldsymbol{V}$ is more conveniently used for assigning initial and boundary conditions. The explicit form of $\boldsymbol{U}$, $\boldsymbol{V}$, $\mathsf{T}(\boldsymbol{U})$ and $\boldsymbol{S}(\boldsymbol{U})$ depends on the particular physics module selected:

- *HD*: Newtonian (classical) hydrodynamics, §3.1;
- *MHD*: ideal/resistive magnetohydrodynamics, §3.2;
- *RHD*: special relativistic hydrodynamics, §3.3;
- *RMHD*: special (ideal) relativistic magnetohydrodynamics, §3.4;

PLUTO adopts a structured mesh approach for the solution of the system of conservation laws (1.1). Flow quantities are discretized on a logically rectangular computational grid enclosed by a boundary and augmented with guard cells or ghost points in order to implement boundary conditions on a given computational stencil. Computations are done using double precision arithmetic.

The grid can be either *static* or dynamically *adaptive* as the flow evolves. In the static grid version **PLUTO** comes as a stand-alone package entirely written in the C programming language, see [28] for a comprehensive description. In the adaptive grid version the code relies on the Chombo library for adaptive mesh refinement (AMR) written in C++ and Fortran (Chapter 7). A thorough description of the AMR implementation is given in [29].

Starting with PLUTO 4, we employ Doxygen as the standard documentation system and no longer distribute the *Developer's Guide*. The Application Programming Interface (API) reference guide can be found in PLUTO/Doc/Doxygen/html/index.hml.

1.1 System Requirements

PLUTO can run on most platforms but some software prerequisites must be met, depending on the specific configuration you intend to use. The minimal set to get **PLUTO** running on a workstation with a static grid (no AMR) requires:

- Python (V. 2.0 or higher) + ncurses libraries;
- (ANSI) C compiler;
- GNU make (gmake);

These are usually installed by default on most Linux/Unix platforms. A comprehensive list is shown in Table 1.1.

Starting with **PLUTO** 4.0 parallelization is handled internally and ArrayLib, used in previous versions of the code, is no longer necessary. The Chombo library is required for computations making use of Adaptive Mesh Refinement (Chapter 7), while the PNG library should be installed only if PNG output is desired. The HDF5 library is required for I/O with the Chombo library and may also be used with the static grid version of the code.

PLUTO has been successfully ported to several parallel platforms including Linux, Windows/Cygwin, Mac OS X, Beowulf clusters, IBM power4 / power5 / power6, SGI Irix, IBM BluGene/P and several others. Figure 1.1 shows the strong scaling on a BlueGene/P machine up to $32,768$ processors on a periodic domain with 512^3 computational grid zones.

	Static Grid		**Adaptive Grid**	
	serial	*parallel*	*serial*	*parallel*
Python (> 2.0)	yes	yes	yes	yes
C compiler	yes	yes	yes	yes
C++ compiler	–	–	yes	yes
Fortran compiler	–	–	yes	yes
GNU make	yes	yes	yes	yes
MPI library	–	yes	–	yes
Chombo library	–	–	yes	yes
HDF5 library	opt	opt	yes	yes
PNG library	opt	opt	–	–

Table 1.1: Software requirements for different applications of PLUTO. Here "opt" stands for optional, "serial" refers to single-processor runs and "parallel" to multiple-processor architectures.

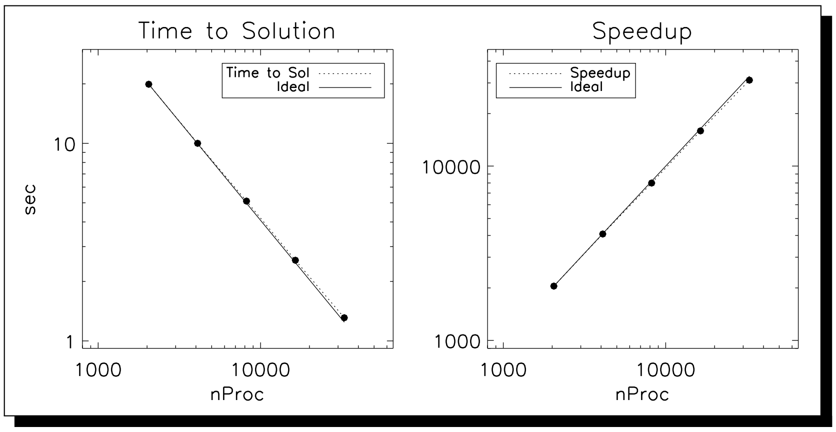


Figure 1.1: Strong scaling of PLUTO on a periodic domain problem with 512^3 grid zones. Left panel: average execution time (in seconds) per step vs. number of processors. Right panel: speedup factor computed as T_1/T_N where T_1 is the (inferred) execution time of the sequential algorithm and T_N is the execution time achieved with N processors. Code execution time is given by black circles (+ dotted line) while the solid line shows the ideal scaling.

1.2 Directory Structure

Once unpacked, your PLUTO/ root directory should contain the following folders:

- Config/: contains machine architecture dependent files, such as information about C compiler, flags, library paths and so on. Important for creating the makefile;
- Doc/: documentation directory;
- Lib/: repository for additional libraries;
- Src/: main repository for all *.c source files with the exception of the init.c file, which is left to the user. The physics module source files are located in their respective sub-directories: HD/ (classical hydrodynamics), RHD/ (special relativistic hydrodynamics), MHD/ (magnetohydrodynamics), RMHD/ (relativistic magnetohydrodynamics). Cooling, viscosity, thermal conduction and additional physics models are located under the folders with similar names (e.g. Cooling/, Viscosity/, Thermal_Conduction). The Templates/ directory contains templates for the user-dependent files such as init.c, pluto.ini, makefile and definitions.h;
- Tools/: Collection of useful tools, such as Python scripts, IDL visualization routines and binary conversion tools;
- Test_Problem/: a directory containing several test-problems commonly used for code verification.

PLUTO should be compiled and executed in a separate working directory which may be anywhere on your local hard drive.

Although most of the current algorithms can be considered in their final stable version, the code is under constant development and updates are released once or twice per year. When upgrading to a

Option	Description
`--with-chombo`	enables support for adaptive mesh refinement (AMR) using the Chombo library, Chapter 7;
`--with-fd`	enables support for finite difference schemes, §5.3
`--with-fargo`	enables support for the FARGO-MHD module, §5.2;
`--with-sb`	enables support for the shearing-box module, §5.1;
`--no-curses`	disables the curses terminal control feature of the Python script. Instead a shell-based setup will be used. This switch can be used to circumvent problems with the ncurses library present on some systems (e.g. Snow Leopard 10.6);

Table 1.2: Command line options available when running the Python setup script.

newer version of **PLUTO** , it is recommended that the entire PLUTO/ directory tree be deleted. Syntax changes are usually listed in the file CHANGES, in the PLUTO/ root directory.

1.3 Configuring PLUTO

In order to configure and setup **PLUTO** for a particular problem, *four* main steps have to be followed; the resulting configuration will then be stored in 4 different files, part of your local working directory:

1. definitions.h: header file containing all problem-dependent flags required at compilation stage (physics module, geometry, dimensions, etc.), see §2.1;
2. makefile: needed to compile **PLUTO** . It depends on your system architecture, §2.2;
3. pluto.ini: startup initialization file containing run-time parameters (grid size, CFL,..., see §2.3);
4. init.c: implements initial, boundary conditions, etc..., see §2.4.

Chapter 2 gives a detailed description for each step. The Python script setup.py must be used for step 1 and 2 and the remaining files (pluto.ini and init.c, step 3 and 4) should be appropriately edited by the user. Templates for all four files can be found in the Src/Templates/ directory. Several examples are located in the test directories under Test_Problem/.

In order to run the Python script anywhere from your hard disk we recommend to set the shell variable PLUTO_DIR to point to your **PLUTO** distribution. Depending on your environment shell, use either one of

```
~> setenv PLUTO_DIR /home/user/PLUTO  # if you're using tcsh shell
~> export PLUTO_DIR=/home/user/PLUTO  # if you're using bash shell
```

The setup.py script can now be invoked with

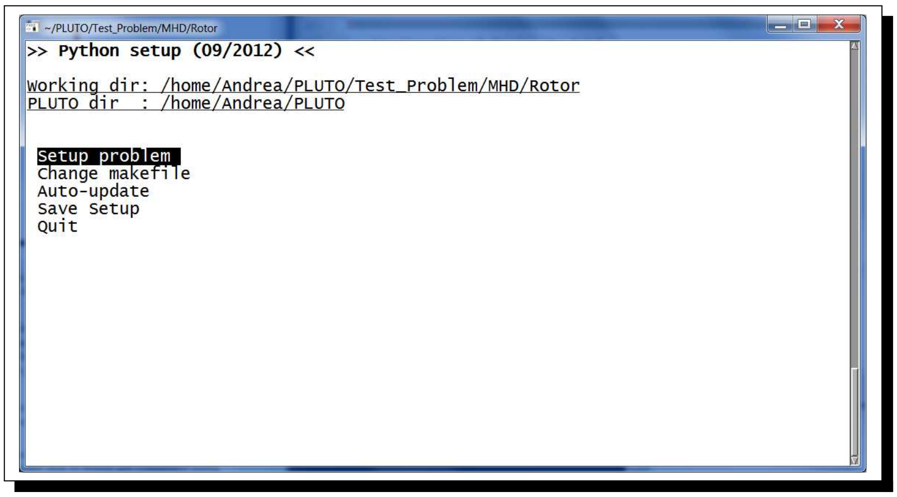
```
~/MyWorkDir > python $PLUTO_DIR/setup.py [options]
```

Command line options are listed in Table 1.2 or can be briefly described by invoking setup.py with `--help`. By default the Python script uses the ncurses library for enhanced terminal control. However, this option may be turned off by invoking the setup script with the `--no-curses` switch. You should then[1] see the menu shown in Fig. 1.2. Additional menus, depending on the physics module, will display later.

1.4 Compiling & Running the Code

After the four steps described in §2.1–§2.4 have been completed, you can compile **PLUTO** in your working directory by typing

[1]Python will first create an architecture-dependent file named sysconf.out containing system-related information: this file does not have any specific purpose but may be helpful for the user. Whenever an internet connection is available, Python will also notify if new versions of the code are available.



```
>> Python setup (09/2012) <<

Working dir: /home/Andrea/PLUTO/Test_Problem/MHD/Rotor
PLUTO dir  : /home/Andrea/PLUTO


 Setup problem
 Change makefile
 Auto-update
 Save Setup
 Quit
```

Figure 1.2: Python script main menu.

```
~/MyWorkDir> make   # 'gmake' is also fine
```

It is important to remember that the makefile created by Python (see §2.2) guarantees that your working directory is always searched before PLUTO/Src. This turns out to be useful when modifying **PLUTO** source files (§1.5).

If compilation is successful, type

```
~/MyWorkDir> ./pluto [flags]
```

for a single processor run, or

```
~/MyWorkDir> mpirun [...] ./pluto [args]
```

for a parallel run; `[...]` are options given to MPI, such as number of processors, etc, while `[args]` are command line options specific to **PLUTO** , see Table 1.3. For example,

```
~/MyWorkDir> ./pluto -restart 5 -maxsteps 840
```

will restart from the 5-th double precision output file and stop computation after 840 steps.

During execution, the integration log will look something like:

```
  ...
step:0 ; t = 0.0000e+00 ; dt = 1.0000e-04 ; 0 % ; [0.000000, 0]
step:1 ; t = 1.0000e-04 ; dt = 1.0000e-04 ; 0 % ; [1.236510, 10]
step:2 ; t = 2.0000e-04 ; dt = 1.1000e-04 ; 0 % ; [1.236510, 7]
step:3 ; t = 3.1000e-04 ; dt = 1.1000e-04 ; 0 % ; [1.236510, 6]
 ...
```

where `step` gives the current integration step, `t` is the current integration time, `dt` is the current time step, `n%` is the percentage of integration. The two numbers in square brackets are, respectively, the maximum Mach number and maximum number of iterations required by the Riemann solver during the previous step. For non-iterative Riemann solvers, the last number will always display 0. The maximum Mach number is a very sensitive function of the numerical method it may be used as a "robustness" indicator. Very large Mach numbers or rapid variations usually indicate problems and/or fixes during the computation.

1.4.1 Command line options

When running **PLUTO** , a number of command-line switches can be given to enable or disable certain features at run time. Some of them are available only in the static grid version, see Table 1.3 for a description of the available flags.

Option	Description	work w/ AMR
`-dec n1 [n2] [n3]`	Enable user-defined parallel decomposition mode. The integers n1, n2 and n3 specify the number of processors along the x1, x2, and x3 directions. There must be as many integers as the number of dimensions and their product must equal the total number of processors used by mpirun or an error will occurr.	No
`-i fname`	Use fname as initialization file instead of pluto.ini.	Yes
`-h5restart n`	Restart computations from the n-th output file in HDF5 double precision format (.dbl.h5).	Yes
`-makegrid`	Generate grid only, do not start computations.	No
`-maxsteps n`	Stop computations after n steps.	Yes
`-no-write`	Do not write data to disk.	Yes
`-no-x1par`, `-no-x2par`, `-no-x3par`	Do not perform parallel domain decomposition along the x1, x2 or x3 direction, respectively.	No
`-restart n`	Restart computations from the n-th output file in double in precision format (.dbl).	No
`-show-dec`	Show domain decomposition when running in parallel mode.	No
`-x1jet`, `-x2jet`, `-x3jet`	Exclude from integration regions of zero pressure gradient that extends up to the end of the domain in the x1, x2 or x3 direction, respectively. This option is specifically designed for jets propagating along one of the coordinate axis. In parallel mode, parallel decomposition is not performed along the selected direction.	No
`-xres n1`	Set the grid resolution in the x1 direction to n1 zones by overriding pluto.ini. Cell aspect ratio is preserved by modifying the grid resolution in the other coordinate directions accordingly.	Yes

Table 1.3: Command line options available when running **PLUTO** . Compatibility with AMR version is given in the last column. †: on parallel architectures only

1.5 Modifying the Distribution Source Files

PLUTO source files are compiled directly from the PLUTO/Src directory. Should you need to modify a C source file other than your init.c, we strongly advise to copy the file in question to your local working directory, since the latter is always searched before PLUTO/Src during the compilation phase. In other words, if you want to modify say, boundary.c, copy the file to your working area and introduce the appropriate changes. When `make` is invoked, your local copy of boundary.c is compiled since it has priority over PLUTO/Src/boundary.c which is actually ignored. In such a way, you can keep track of the problem dependent modification, without affecting the original distribution.

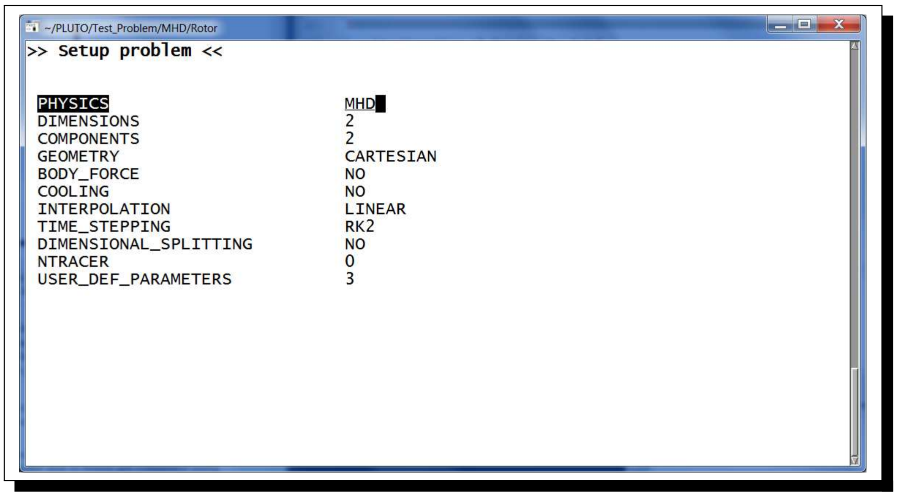
Note, however, that header files (*.h or *.H) do not follow the same convention and should be modified in their original directory.

2. Problem Setup

This chapter explains how to create the four files (definitions.h, makefile, pluto.ini and init.c) required to compile and run **PLUTO** .

2.1 STEP # 1: header file definitions.h

The header file definitions.h is created by the Python script setup.py by selecting *Setup problem* (see Fig. 2.1). If you do not have an existing definitions.h, a new one will be created for you, otherwise the Python script will try to read your current setup from it.



```
>> Setup problem <<

 PHYSICS                       MHD
 DIMENSIONS                    2
 COMPONENTS                    2
 GEOMETRY                      CARTESIAN
 BODY_FORCE                    NO
 COOLING                       NO
 INTERPOLATION                 LINEAR
 TIME_STEPPING                 RK2
 DIMENSIONAL_SPLITTING         NO
 NTRACER                       0
 USER_DEF_PARAMETERS           3
```

Figure 2.1: The Setup problem menu, needed for your definitions.h and makefile creation; by moving the arrow keys you should be able to browse through different options.

The header file definitions.h also contains other more advanced switches that are not accessible via the Python script (§2.1.11) and should be changed manually. We now describe the options accessible through the Python script.

2.1.1 PHYSICS

Specifies the fluid equations to be solved. The available options are:

- *HD*: classical hydrodynamics described by the Euler equations, §3.1;
- *MHD*: single fluid, ideal/resistive magnetohydrodynamics, §3.2;
- *RHD*: special relativistic hydrodynamics, §3.3;
- *RMHD*: special relativistic magnetohydrodynamics, §3.4.

2.1.2 DIMENSIONS & COMPONENTS

DIMENSIONS sets the number of spatial dimensions of your problem whereas COMPONENTS sets the number of vector components (such as velocity and magnetic field) present in the integration. Usually DIMENSIONS=COMPONENTS, but one can also have more COMPONENTS than DIMENSIONS. This is the case, for example, when the "$2 + \frac{1}{2}$ D" formalism is used, where integration is performed along the first two coordinates (say x, y) but the fluid has a non-vanishing velocity component along the third direction as well (say $\partial v_z/\partial x, \partial v_z/\partial y \neq 0$). An example is an axisymmetric 2-D cylindrical problem (such as a disk or a torus) in the (r, z) plane with a uniform rotation in the azimuthal direction ϕ (where it is assumed $\partial/\partial\phi = 0$). In any case it is required that `DIMENSIONS` $\leq$ `COMPONENTS`.

2.1.3 `GEOMETRY`

Sets the geometry of the problem. Spatial coordinates are generically labeled with x_1, x_2 and x_3 and their physical meaning depends on the value assigned to `GEOMETRY`:

- *`CARTESIAN`*: Cartesian coordinates $\{x_1, x_2, x_3\} = \{x, y, z\}$;
- *`CYLINDRICAL`*: cylindrical axisymmetric coordinates $\{x_1, x_2\} = \{r, z\}$ (1 or 2 dimensions);
- *`POLAR`*: polar cylindrical coordinates $\{x_1, x_2, x_3\} = \{r, \phi, z\}$;
- *`SPHERICAL`*: spherical coordinates $\{x_1, x_2, x_3\} = \{r, \theta, \phi\}$.

Note that when `DIMENSIONS` $= 2$, the third coordinate x_3 is meaningless and will be set to zero (similarly in 1-D x_2 and x_3 do not play any role). Whenever present, however, the ϕ component of vectors (both in spherical and cylindrical coordinates) is integrated by discretizing the equations in angular momentum conserving form.

We warn that non-Cartesian geometries are handled better when a multi-stage unsplit integrator (i.e. Runge-Kutta) is used, especially if angular coordinates are present and/or steady state solutions are sought.

2.1.4 `BODY_FORCE`

Include a body force in the momentum and energy equations. Possible values are:

- *`POTENTIAL`*: body force is derived from a scalar potential, $\rho\boldsymbol{a} = -\rho\nabla\Phi$;
- *`VECTOR`*: body force is expressed as a three-component vector $\rho\boldsymbol{a} = \rho\boldsymbol{g}$.
- *`(VECTOR+POTENTIAL)`*: body force is prescribed using both, $\rho\boldsymbol{a} = \rho(-\nabla\Phi + \boldsymbol{g})$.

More details can be found in §2.4.3.

2.1.5 `COOLING`

Optically thin thermal losses can be included by appropriately setting this flag to one of the following:

- *`POWER_LAW`*: radiative losses are proportional to $\rho^2 T^\alpha$ (§4.5.2);
- *`TABULATED`*: radiative losses are computed as $n^2\Lambda(T)$, where $\Lambda(T)$ is a user-supplied tabulated function of temperature, see §4.5.3. Alternatively, this module can be used to provide user-defined cooling functions;
- *`SNEq`*: simplified non-equilibrium cooling function. See §4.5.4 for more details;
- *`MINEq`*: multi-ion non-equilibrium cooling model. It evolves the standard equations augmented with a chemical network of 29 ions, see §4.5.5 and the work by [49].

2.1.6 `INTERPOLATION`

Sets the spatial order of integration. In the standard (finite volume) version of the code, the following options are available:

- *`FLAT`*: first order reconstruction. The stencil is 1 point.
- *`LINEAR`*: piecewise TVD linear interpolation is applied to primitive variables. It is 2nd order accurate in space. Stencil is 3 point wide.

- *`WENO3`*: provide 3rd order weighted essentially non oscillatory reconstruction [52] inside a cell using is 3-point stencil.
- *`LimO3`*: provide 3rd order limiter function [7] based on a 3-point stencil.
- *`PARABOLIC`*: piecewise parabolic method (PPM) as implemented by [8] or [24]. The stencil requires 5 zones.

The default is *`LINEAR`*. Both *`WENO3`* and *`LimO3`* employ a local three-point stencil to achieve piecewise-quadratic reconstruction for smooth data and preserves their accuracy at local extrema thus avoiding clipping of classical second-order TVD limiters and PPM. Non-uniform grid spacing is correctly handled only by *`PARABOLIC`* and *`WENO3`*.

Note that although 3rd-order reconstructions are available, the finite volume version of the code retains a global 2nd-order accuracy as fluxes are computed at the interface midpoint. On the contrary, genuine 3rd and 5th order accurate schemes can be employed using the conservative finite difference framework, §5.3.

2.1.7 `TIME_EVOLUTION`

PLUTO has several time-marching algorithms which can be used in either a spatially split or unsplit fashion. If $\Delta t^n = t^{n+1} - t^n$ is the time increment between two consecutive steps (Table 2.1) and $\boldsymbol{\mathcal{L}}$ denotes the discretized spatial operator on the right hand side of Eq. (1.1), the possible options are:

- *`EULER`*: first order (explicit) Euler algorithm is used to evolve from $\boldsymbol{U}^n$ to $\boldsymbol{U}^{n+1}$:

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n + \Delta t^n \boldsymbol{\mathcal{L}}^n$$

- *`RK2`*, *`RK3`*: second or third order TVD Runge Kutta is used to advance the solution from time t^n to the next step time t^{n+1}:

RK2	RK3
$\boldsymbol{U}^* = \boldsymbol{U}^n + \Delta t^n \boldsymbol{\mathcal{L}}^n$	$\boldsymbol{U}^* = \boldsymbol{U}^n + \Delta t^n \boldsymbol{\mathcal{L}}^n$
−	$\boldsymbol{U}^{**} = \frac{1}{4}\Big(3\boldsymbol{U}^n + \boldsymbol{U}^* + \Delta t^n \boldsymbol{\mathcal{L}}^*\Big)$
$\boldsymbol{U}^{n+1} = \frac{1}{2}\Big(\boldsymbol{U}^n + \boldsymbol{U}^* + \Delta t^n \boldsymbol{\mathcal{L}}^*\Big)$	$\boldsymbol{U}^{n+1} = \frac{1}{3}\Big(\boldsymbol{U}^n + 2\boldsymbol{U}^{**} + 2\Delta t^n \boldsymbol{\mathcal{L}}^{**}\Big)$

(2.1)

When `DIMENSIONAL_SPLITTING` = *`YES`*, the operator $\boldsymbol{\mathcal{L}}$ in Eq. (2.1) is one-dimensional. Setting `DIMENSIONAL_SPLITTING` = *`NO`* makes the scheme dimensionally unsplit and the right hand side include contributions from all directions simultaneously. Unsplit implementation of the Runge-Kutta algorithms usually requires a somewhat more restrictive CFL condition, see Table 2.1.

- *`CHARACTERISTIC_TRACING`*, *`HANCOCK`*: they evolve $\boldsymbol{U}^n$ according to

$$\boldsymbol{U}^{n+1} = \boldsymbol{U}^n + \Delta t^n \boldsymbol{\mathcal{L}}(\boldsymbol{V}^{n+\frac{1}{2}})$$

where $\boldsymbol{V}^{n+\frac{1}{2}}$ is computed by suitable Taylor expansion. Although the final step is in divergence form, these methods require the primitive formulation of the equations, not yet available for all modules. They are 2nd order accurate in space and time and less dissipative than the previous multi-step algorithms. *`HANCOCK`* should be combined with a linear interpolant, while *`CHARACTERISTIC_TRACING`* which does a more sophisticated characteristic limiting, can be combined with all reconstruction algorithms. The original PPM scheme of [8, 24] is available for the `HD, MHD` and `RHD` modules by selecting `TIME_EVOLUTION` = *`CHARACTERISTIC_TRACING`*, together with `INTEPOLATION` = *`PARABOLIC`* and a *`two-shock`* Riemann solver (*`Roe`* or *`hlld`* alternatively).

Setting `DIMENSIONAL_SPLITTING` = *`NO`* yields the spatially unsplit fully corner-coupled method of [10, 26]. This scheme is stable under the condition CFL $\lesssim 1$ (in 2D) and CFL $\lesssim 1/2$ (in 3D) and it is slightly more expensive than *`RK2`*.

Time Step Determination. The time step Δt^n is computed using the information available from the previous integration step and it can be controlled by the Courant-Friedrichs-Lewy (CFL) number C_a within the limits suggested in Table 2.1, see [5]. Thus one immediately sees that, if Δl is the cell physical length, the time step roughly scales as $\sim \Delta l$ for hyperbolic problems and as $\sim \Delta l^2$ when parabolic terms are included (§4.4.1). On the contrary, when parabolic terms are included via Super-Time-Stepping integration (§4.4.2) the time step can be much larger being computed solely from the advection time scale (i.e. $\tau_d = 0$ is the table below).

`SCHEME`	`DIM. SPLIT`	`CFL` Limit
`RK`	*`YES`*	$\Delta t^n \max_d \left[\max_{ijk}\left(\frac{\lambda_d}{\Delta l_d} + \frac{2\tau_d}{\Delta l_d^2}\right)\right] = C_a \le 1$
`MH/ChTr`	*`YES`*	$\Delta t^n \max_d \left[\max_{ijk}\left(\frac{\lambda_d}{\Delta l_d} + \frac{2\tau_d}{\Delta l_d^2}\right)\right] = C_a \le 1$
`RK`	*`NO`*	$\Delta t^n \max_{ijk}\left[\frac{1}{N_{\rm dim}}\sum_d\left(\frac{\lambda_d}{\Delta l_d} + \frac{2\tau_d}{\Delta l_d^2}\right)\right] = C_a \le \frac{1}{N_{\rm dim}}$
`MH/ChTr`	*`NO`*	$\Delta t^n \left[\max_{ijk}\left(\frac{\lambda_d}{\Delta l_d}\right) + \max_{ijk}\left(\frac{2\tau_d}{\Delta l_d^2}\right)\right] = C_a \le \begin{cases} 1 & \text{in } 2D \\ 1/2 & \text{in } 3D \end{cases}$

Table 2.1: CFL conditions used by **PLUTO** for different explicit time stepping methods. For a given direction d, Δl_d represents the cell physical length in that direction, λ_d provides the largest signal speed while τ_d accounts for diffusion processes. Here *`MH`* and *`ChTr`* stand for *`HANCOCK`* and *`CHARACTERISTIC_TRACING`*, respectively. These limits are based on a stability analysis on the constant coefficient advection-diffusion equation by by Beckers (1992), [5].

Multi-step algorithms (*`RK2, RK3`*) work in all system of coordinates and are the default choice. Single-step schemes (*`HANCOCK, CHARACTERISTIC_TRACING`*) are more sophisticated, have less dissipation and have been tested mainly on Cartesian and cylindrical grids. Have a look at Table 2.2 for a comparison between different (suggested) integration schemes commonly adopted in testing the code.

2.1.8 `DIMENSIONAL_SPLITTING`

Set this feature to *`YES`* if you intend to use Strang operator splitting [45] to solve the equations in multi dimensions. If `DIMENSIONAL_SPLITTING` is set to *`NO`* flux contributions are evaluated from all directions simultaneously. Dimensionally unsplit schemes avoid the errors due to operator splitting and are generally preferred. Table 2.2 gives a brief description of commonly used setups.

2.1.9 `NTRACER`

The number of passive scalars or "colors" (denoted with Q_k) obeying simple advection equations of the form:

$$\frac{\partial Q_k}{\partial t} + \boldsymbol{v}\cdot\nabla Q_k = 0 \qquad \Longleftrightarrow \qquad \frac{\partial(\rho Q_k)}{\partial t} + \nabla\cdot\big(\rho Q_k \boldsymbol{v}\big) = 0$$

The array index used to access tracer variables (§2.4.1,§2.4.2) is `[TR]` for the first tracer, `[TR+1]` for the second one and so on. The maximum number is 4.

2.1.10 `USER_DEF_PARAMETERS`

Sets the number of user-defined parameters that can be accessed from anywhere in the code. The maximum number is 32, while the minimum number is 1. The explicit numerical value is read at runtime from pluto.ini and can be changed before execution without re-compiling the code.

The parameters are identified by means of a label corresponding to an integer index of the global array `g_inputParam` visible anywhere in the program. If, for instance, `USER_DEF_PARAMETERS` has been set equal to 3, you will be prompted to define 3 different "labels", say `FOO_1`, `FOO_2` and `FOO_3`, as

INTERP.	TIME STEP.	DIM. SPLIT	Cost	Comments
LINEAR	*RK2*	*YES,NO*	$2N_{\rm dim}$	Default setup. Compatible with almost every algorithms of the code and work in all system of coordinates and physics modules. The dimensionally unsplit version is stable up to $CFL \lesssim 1/N_{\rm dim}$, where $N_{\rm dim}$ is the number of dimensions.
PARABOLIC, WENO3,LimO3	*RK3*	*YES,NO*	$3N_{\rm dim}$	Similar to the previous setup, but it has better stability properties for higher than 2$^{\rm nd}$ order interpolants. The dimensionally unsplit version is stable up to $CFL \lesssim 1/N_{\rm dim}$.
LINEAR	*HANCOCK*	*YES*	$N_{\rm dim}$	MUSCL-Hancock second-order scheme of [51, 50]. Computationally more efficient than RK integrators, it is probably the faster 2$^{\rm nd}$ order algorithm. Works well for the HD, RHD modules and the MHD module with the 8-wave formulation, particularly on Cartesian (1,2,3 dimensions) or cylindrical geometries.
LINEAR	*ChTr*	*YES*	$N_{\rm dim}$	More sophisticated than the previous one, it yields the Piecewise Linear Method of [51, 9].
PARABOLIC	*ChTr*	*YES*	$N_{\rm dim}$	Gives the original Piecewise-Parabolic-Method (PPM) of [8]. Suggested for the HD and RHD on Cartesian (1,2,3 dimensions) or cylindrical geometries. It is stable up to $CFL \lesssim 1$ and it has small dissipation.
LINEAR	*ChTr, HANCOCK*	*NO*	$2N_{\rm dim}$	Yields the Corner-Transport Upwind method of [10, 42, 26] and [14] for the MHD module. It is fully unsplit and stable up to 1 (in 2-D) and $\sim$ 0.5 in 3D. It is one of the most sophisticated algorithms available. It is suitable for computations in Cartesian and cylindrical grids in the HD, RHD and MHD module.

Table 2.2: Suggested algorithm configurations. The cost (4th column) is given in terms of number of Riemann problems per cell per step. $N_{\rm dim}$ is the number of spatial dimensions. *ChTr* stands for *CHARACTERISTIC_TRACING*.

in Fig. 2.2. These names are the integer indexes of the `g_inputParam` array: `g_inputParam[FOO_1]` will contain the actual value of the first user-defined parameter, `g_inputParam[FOO_2]` the second one and so forth.

Parameter names should be chosen with care in order to avoid overlapping wth other program variables. Although there are no strict rules, we advise to use capital letters, avoid short labels such as "`V0`" or "`VX`" and choose a more representative name that explains the use of the variable on its own, e.g., `PAR_INFLOW_VEL`.

Parameter names (and values) are automatically inserted inside pluto.ini in the correct order during the setup script. However, if you use a different initialization file, you may have to set the parameter names together with their values manually.

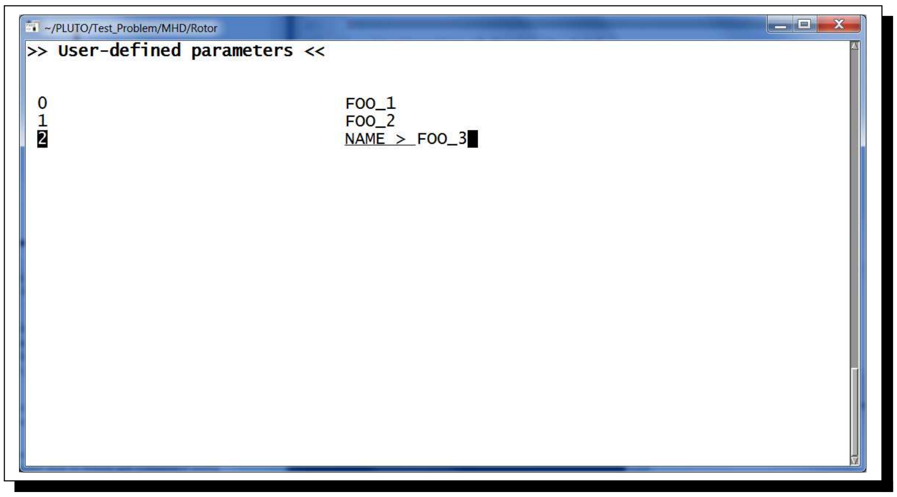
2.1.11 Additional Switches

Besides the options discussed so far, accessible via the Python script, definitions.h contains additional switches. You can skip this section if you are new to **PLUTO** . These additional switches are not accessible via the Python script, but may be changed just by editing your definitions.h:

- `INITIAL_SMOOTHING` (*YES/NO*) :
  when set to *YES*, initial conditions are assigned by sub-sampling and averaging different values inside each cell. It is useful to create smooth profiles of sharp boundaries not aligned with the grid (e.g., a circle in Cartesian coordinates).

- `WARNING_MESSAGES` (*YES/NO*) :
  issue a warning message every time a numerical problem or inconsistency is encountered; setting `WARNING_MESSAGES` to *YES* will tell **PLUTO** to print what, when and where a numerical problem occurred.

- `PRINT_TO_FILE` (*YES/NO*) :
  when set to *YES* it tells **PLUTO** to re-direct the output log to the file pluto.log. If this file does not



```
~/PLUTO/Test_Problem/MHD/Rotor
>> User-defined parameters <<

 0                               FOO_1
 1                               FOO_2
 2                               NAME > FOO_3
```

Figure 2.2: User defined names are chosen in this sub-menu.

exist it will be created; if the file exists but integration starts from initial conditions, it will be over written. Finally, if you restart from a previously saved file, the output will be appended.

- `INTERNAL_BOUNDARY` (*`YES`*/*`NO`*) :
  when turned to *`YES`*, it allows to overwrite or change the solution array anywhere inside the computational domain. This is done inside the **`UserDefBoundary()`** function when `side==0`, see §2.4.2. This option is particularly useful when flow variables must be kept constant in time or to assign lower/upper threshold values to any physical quantity (e.g. density or pressure).

- `SHOCK_FLATTENING` (*`NO`*/*`ONED`*/*`MULTID`*) :
  Provides additional dissipation in proximity of strong shocks. When set to *`ONED`*, spatial slopes are progressively reduced following a one-dimensional shock recognition pattern, as in [8]. This is done separately dimension by dimension.

  When set to *`MULTID`*, a multi dimensional strategy is used by which, upon shock detection, i) interpolation (in every direction) reverts to the *`MINMOD`* limiter and ii) fluxes are computed using the HLL Riemann solver. The flagging strategy is set in States/findshock.c. The *`MULTID`* shock flattening has proven to be an effective adaptation strategy that can noticeably increase the code robustness. It is highly suggested for complex flow structures involving strong shocks.

- `ARTIFICIAL_VISCOSITY` (*`YES`*/*`NO`*) :
  when set to *`YES`*, it includes additional dissipation using Lapidus-type viscosity. This should be used only with the two-shock Riemann solver.

- `CHAR_LIMITING` (*`YES`*/*`NO`*) :
  set to *`YES`* to perform reconstruction on characteristic variables rather than primitive. It is available for the `HD`, `RHD` and `MHD` modules. Although somewhat more expensive, characteristic variable interpolation is known to produce better quality solutions by suppressing unwanted numerical oscillation in proximity of strong discontinuities and leading to a better entropy enforcement. We recommend setting this switch to *`YES`* whenever possible.

- `LIMITER` (*`string`*) :
  Set the limiter(s) to be applied. *`string`* can be one of

  – *`DEFAULT`*: keep the default setting (defined in Src/States/set_limiter.c).
  – *`MINMOD_LIM`*: use the minmod limiter (most diffusive).
  – *`VANALBADA_LIM`*: use the van Albada limiter function.
  – *`UMIST_LIM`*: use the umist limiter.
  – *`VANLEER_LIM`*: use the harmonic mean limiter of van Leer.
  – *`MC_LIM`*: use the monotonized central difference limiter (least diffusive).

  - `FOURTH_ORDER_LIM`: use the fourth-order approximate limiter of [9, 42].

  where `MINMOD_LIM` is the most diffusive and `FOURTH_ORDER_LIM` is the least diffusive limiter. This switch has effect only for `LINEAR` interpolation. All limiters employ a 3-point stencil except for `FOURTH_ORDER_LIM` which uses 5 zones.

- `CT_EMF_AVERAGE` (*string*, only for `CONSTRAINED_TRANSPORT` MHD/RMHD):
  controls how the electromotive force (EMF) is integrated from the face center to the edges. This is discussed in more detailed in §3.2.4.3.

- `CT_EN_CORRECTION` (`YES`/`NO`, only for `CONSTRAINED_TRANSPORT` MHD/RMHD):
  this option is available only in the MHD and RMHD modules. The default is `NO`, implying that energy is not corrected after the conservative update. However, for low-beta plasma one may find useful to switch this option to `YES`, as described in [3].

- `ASSIGN_VECTOR_POTENTIAL` (`YES`/`NO`):
  when set to `YES`, magnetic field components are initialized from the vector potential. In the constrained transport algorithm (CT, §3.2.4.3), this guarantees that the magnetic field has zero divergence. When set to `NO`, assignment proceeds in the usual way, see §3.2.3 for more details.

- `UPDATE_VECTOR_POTENTIAL` (`YES`/`NO`):
  enable this option if you wish to evolve the vector potential in time and save it to disk.

- `STS_nu` (*double*):
  sets the value of the ν parameter used to control the efficiency of Super-Time-Stepping integration for parabolic (diffusion) terms, see chapter 4 and §4.4.2.

2.2 STEP # 2: makefile creation

The makefile contains instructions to compile and link C source code files and produce the executable pluto. The Python script creates a new makefile every time you choose *Change makefile* from the menu; otherwise, it automatically updates the existing one after you have finished the problem setup.

If you choose to create a new makefile, Python will ask you to select an appropriate .defs file containing architecture-dependent flags from the Config/ directory. The template Config/Template.defs can be used to create a new configuration from scratch.

The simplest example is a definition file for a single-processor without any additional library. In this case it suffices to set:

```
CC       = cc
CFLAGS   = -c -O
LDFLAGS  = -lm

PARALLEL = FALSE  # TRUE/FALSE: enable/disable parallel mode
USE_HDF5 = FALSE  # TRUE/FALSE: enable/disable support for HDF5 library
USE_PNG  = FALSE  # TRUE/FLASE: enable/disable support ofr PNG library
```

where CC is the name of your C compiler (cc, gcc, mpicc, etc...), CFLAGS are command line options (such as optimization, search path, etc...) and LDFLAGS contains options to be passed to the linker.

The variables PARALLEL, USE_HDF5 and USE_PNG can be set to either *TRUE* or *FALSE* to enable or disable, respectively, parallel mode, support for HDF5 library and support for PNG library in the *static* grid version of **PLUTO** . When set to *TRUE* the same variable name is passed to **PLUTO** as a #define directive with value 1.

As an example, if USE_HDF5 is set to *TRUE* inside a .defs file then any C source file containing instructions inside a preprocessor directive #ifdef USE_HDF5 ... #endif statement will be compiled.

Note: These switches are effective only in the static grid version of the code and have no effect when creating a **PLUTO**-Chombo makefile, §7.2.

2.2.1 MPI Library (Parallel) Support

Parallel executables for the static grid version of **PLUTO** are created by specifying the name of a MPI C compiler (e.g. mpicc) and by setting the makefile variable PARALLEL to *TRUE* in your .defs file:

```
CC       = mpicc # or similar
...

PARALLEL = TRUE
...

################################
# MPI additional spefications
################################

ifeq ($(strip $(PARALLEL)), TRUE)
 USE_ASYNC_IO =     # if TRUE, enable Asynchronous binary I/O
endif
```

In this case, you may also modify existing variables or add new ones inside the conditional statement beginning with ifeq.

When parallel mode is enabled, C source code sections that are specific to MPI should be enclosed inside #ifdef PARALLEL ... #endif statements.

2.2.1.1 Asynchrounous I/O

By enabling the USE_ASYNC_IO to *TRUE*, **PLUTO** allows to replace the standard blocking calls of MPI with non-blocking and split collective calls available in the MPI-2 I/O standard[1]. Given suitable hardware, this allows the transfer of data out/in the user's buffer to proceed concurrently with computation. A separate request complete call is needed to complete the I/O request, i.e., to confirm that the data has been read or written and that it is safe for the user to reuse the buffer. Overall, this results in an improved performance for intensive I/O computations. More details may be found in http://www.prace-project.eu/IMG/pdf/petascale_astrophysical_simulations_with_pluto.pdf.

Note: This is an *experimental* feature that can be used, in the current version of the code, only for .flt or .dbl binary files for saving cell-centered data.

2.2.2 HDF5 Library Support

If your system is already configured with serial or parallel HDF5 libraries, you may enable support for HDF5 I/O in the static grid version of **PLUTO** by turning the makefile variable USE_HDF5 to *TRUE*. If you do not have HDF5 installed, you may follow the installation guidelines given in §7.1.1. Note that the same HDF5 library can be used for both the static and AMR versions of **PLUTO** although support for HDF5 in the AMR version is enabled differently, see §7.1.2.

Once USE_HDF5 has been set to *TRUE*, add the HDF5 library path to the list of directories to be searched for header files as well as the corresponding linker option for HDF5 library files. Note that different pathnames should be given if you are building **PLUTO** in serial or parallel mode. These information are supplied using the INCLUDE_DIRS and LDFLAGS variables, respectively:

```
...
USE_HDF5 = TRUE
...

################################
#    HDF5 library options
################################

ifeq ($(strip $(USE_HDF5)), TRUE)
 HDF5_LIB = /usr/local/lib/HDF5-1.xx
 INCLUDE_DIRS += -I$(HDF5_LIB)/include
 LDFLAGS      += -L$(HDF5_LIB)/lib -lhdf5 -lz
endif
```

Note: **PLUTO** is compatible with HDF5 `1.6.x` API, although it may be be linked with HDF5 1.8.x without any problem, since the H5_USE_API macro (defined in hdf5_io.c) forces the library to use HDF5 1.6 macro definitions.
Alternatively, you may also configure and install HDF5 1.8.x from scratch by providing the `--with-default-api-version=v16` backward compatibility flag to the configure script. See also §7.1.1.

2.2.3 PNG Library Support

Whenever the portable network graphics (PNG) library is installed on your system, you may enable support for 2D output using PNG color images. To do so, simply set to *TRUE* the corresponding USE_PNG variable inside your .defs file and add the linker option to the LDFLAGS variable:

[1] Contrary to a blocking call which will not return until the I/O request is completed, a non-blocking call initiates an I/O operation but does not wait for it completion

```
...
USE_PNG = TRUE
...

###############################
#     PNG library options
###############################

ifeq ($(USE_PNG), TRUE)
 LDFLAGS += -lpng
endif
```

2.2.4 Including Additional Files: local_make

Additional (e.g. user defined) files may be added to the standard object list created by Python in your makefile. To this end, create a new file named local_make like:

```
OBJ     += myfile.o
HEADERS += myheader.h
```

This will instruct `make` that **PLUTO** has to be compiled and linked together with the (user-supplied) file myfile.c which depends on myheader.h. This is particularly useful when the user wants to compile and link the code together with supplementary routines contained in external files.

2.3 STEP # 3: The initialization file pluto.ini

At start-up, the code checks for the pluto.ini input file (or a different one if the `-i` command flag is given) that contains all the run-time information necessary for integration. A template for this file can be found in the Src/Templates directory. The initialization file is divided into eight different "blocks" enclosed by a pair of square brackets $[\cdots]$. Each block contains a set of labels and associated (mandatory or optional) fields:

```
[Grid]

X1-grid    1    0.0    100    u    1.0
X2-grid    1    0.0    100    u    1.0
X3-grid    1    0.0    1      u    1.0

[Chombo Refinement]

Levels           4
Ref_ratio        2 2 2 2 2
Regrid_interval  2 2 2 2
Refine_thresh    0.3
Tag_buffer_size  3
Block_factor     4
Max_grid_size    32
Fill_ratio       0.75

[Time]

CFL              0.4
CFL_max_var      1.1
CFL_par          0.3      # optional
rmax_par         40.0     # optional
tstop            1.0
first_dt         1.e-4

[Solver]

Solver         tvdlf

[Boundary]

X1-beg        outflow
X1-end        outflow
X2-beg        outflow
X2-end        outflow
X3-beg        outflow
X3-end        outflow

[Static Grid Output]

uservar     0
dbl        1.0  -1   single_file
flt       -1.0  -1   single_file
vtk       -1.0  -1   single_file   # optional
dbl.h5     1.0  2.40h              # optional
flt.h5     1.0  -1                 # optional
tab       -1.0  -1                 # optional
ppm       -1.0  -1                 # optional
png       -1.0  -1                 # optional
log        1
analysis  -1.0  -1                 # optional

[Chombo HDF5 output]

Checkpoint_interval  -1.0  0
Plot_interval         1.0  0

[Parameters]

SCRH   0
```

Tag labels on the left side should not be changed. They identify appropriate field(s) following on the same line. Block ordering is irrelevant. The quantities (and related data-types) read from the file are now described.

2.3.1 The *[Grid]* block

Defines the physical domain and generates the grid.

- `X1-grid`: (*integer*) (*double*) (*integer*) (*char*) (...) (*double*);
- `X2-grid`: (*integer*) (*double*) (*integer*) (*char*) (...) (*double*);
- `X3-grid`: (*integer*) (*double*) (*integer*) (*char*) (...) (*double*);

For each dimension: the first (*integer*) defines the number of non-overlapping, adjacent one-dimensional grid patches making up the computational domain (**Note:** this has nothing to do with parallel decomposition which is separately carried out by MPI).

If, say, a uniform grid covers the whole physical domain this number should be set to 1. If two consecutive adjacent grids are used, then 2 is the correct choice and so on. For each patch, the triplet (*double*) (*integer*) (*char*) specifies, respectively, the leftmost node coordinate value, number of points and grid type for that patch; there must be as many triplets (...) as the number of patches. Since patches do not overlap, the rightmost node value of one grid defines the leftmost node value of the next adjacent one. The last (*double*) specify the rightmost node coordinate value of the last segment, which is also the rightmost node value in that direction. If a dimension is ignored, then 1 grid-point only should be assigned to that grid.

The global domain therefore extends, in each direction, from the first (*double*) node coordinate to the last (*double*) node coordinate. These values can be accessed from anywhere in the code using the global variables `g_domBeg[d]` and `g_domEnd[d]`, where `d=`*IDIR, JDIR, KDIR* is used to select the direction.

The grid-type (*char*) entry can take the following values:

- *u* or *uniform*: a uniform grid patch is constructed; if x_L and x_R are the leftmost and rightmost point of the patch, the grid spacing becomes:

$$\Delta x = \frac{x_R - x_L}{N}$$

- *s* or *stretched*: a stretched grid is generated. Stretched grids can be used only if at least one uniform grid is present. The stretching ratio r is computed as follows:

$$\Delta x \left(r + r^2 + \cdots + r^N\right) = x_R - x_L \quad \Longrightarrow \quad r\frac{1-r^N}{1-r} = \frac{x_R - x_L}{\Delta x}$$

  where Δx is taken from the closest uniform grid, N is the number of points in the stretched grid and x_L and x_R are the leftmost and rightmost points of the patch.

- *l*±: a logarithmic grid is generated. When *l+* is invoked, the mesh size is increasing with the coordinate:

$$\Delta x_i = \left(x_{i-\frac{1}{2}} + |x_L| - x_L\right)\left(10^{\Delta y} - 1\right)\,, \quad \Delta y = \frac{1}{N}\log\left(\frac{x_R + |x_L| - x_L}{|x_L|}\right)$$

  when *l-* is invoked, the mesh size *decreases* with the coordinate:

$$\Delta x_i = \left(x_{i-\frac{1}{2}} - |x_L| - x_R\right)\left(10^{\Delta y} - 1\right)\,, \quad \Delta y = -\frac{1}{N}\log\left(\frac{x_R + |x_L| - x_L}{|x_L|}\right)$$

  In practice, the mesh spacing in the *l+* grid is obtained by reversing the spacing in the *l-* grid.

Note: The interval should not include the origin when using a logarithmic grid.

Beware that non-uniform grids may introduce extra dissipation in the algorithm. Changes in the grid spacing are correctly accounted for when `INTERPOLATION` is set to either *PARABOLIC* or *WENO3*.

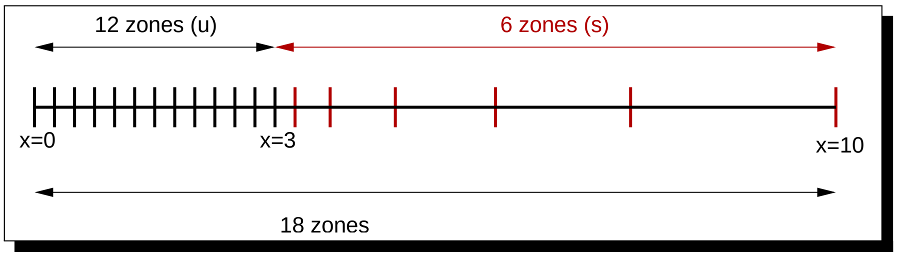


Figure 2.3: Example of one dimensional grid with a uniform (left) and stretched segment (right in red) covering the interval $[0, 10]$.

Example # 1: A simple uniform grid extending from $x_L = 0.0$ to $x_R = 10.0$ with 128 zones can be specified using:

```
X1-grid   1    0.0  128  u  10.0
```

Example # 2: consider a one-dimensional physical domain extending from 0.0 to 10.0 with a total of 18 zones, but a finer grid is required for $0 \leq x \leq 3$. Then one might specify

```
X1-grid   2    0.0 12 u   3.0 6 s   10.0
```

which generates a uniform grid with 12 zones for $0 \leq x \leq 3$, and a stretched grid with 6 zones for $3 \leq x \leq 10$, see Fig.2.3

When the computational grid is generated, each processor owns a domain portion defined by the global integer variables `IBEG` $\leq$ `i` $\leq$ `IEND`, `JBEG` $\leq$ `j` $\leq$ `JEND` and `KBEG` $\leq$ `k` $\leq$ `KEND`. Ghost cells are added outside the local computational domain to complete the stencil at the boundaries, see Fig. 2.4. The global variables `NX1`, `NX2` and `NX3` define the total number of points (boundaries *excluded*) such that `IEND - IBEG + 1 = NX1`, `JEND - JBEG + 1 = NX2`, `KEND - KBEG + 1 = NX3`. The total number of zones (for a given processor, boundaries *included*) is given by the global variables `NX1_TOT`, `NX2_TOT` and `NX3_TOT`, see Fig. 2.4.

When using Adaptive Mesh Refinement with the Chombo library (see Chapter 7), only one uniform grid patch can be generated in each dimension, defining the base level grid. Also, the cells must in this case have the same physical length in each direction (e.g., squares in 2D, cubes in 3D). The refinement options are set in the `Chombo Refinement` section.

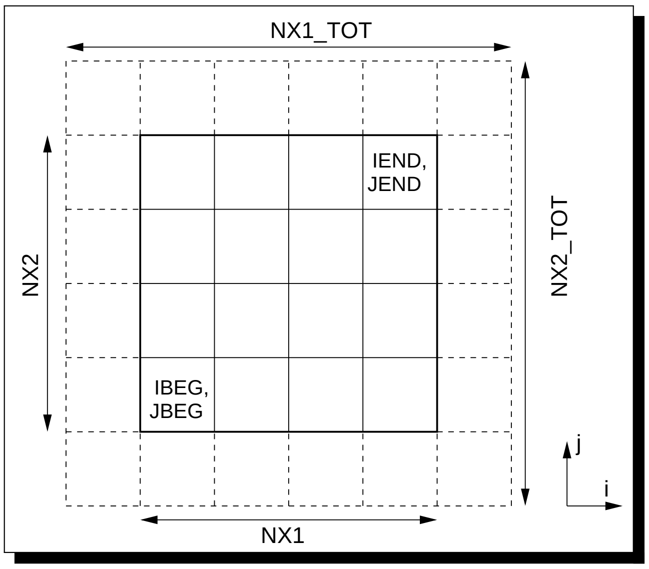


Figure 2.4: Computational grid in 2 dimensions with `NX1 = NX2 = 4` and 1 ghost zone. Internal zones (solid boxed) are spanned by IBEG $\leq i \leq$ IEND, JBEG $\leq j \leq$ JEND. Dashed boxes represent boundary ghost zones.

2.3.2 The *[Chombo Refinement]* Block

Controls the grid refinement if **PLUTO** has been compiled with the Chombo library, see §7.2.3. It is ignored otherwise.

2.3.3 The *[Time]* Block

This section specifies some adjustable time-marching parameters:

- `CFL` (*double*)
  Courant number: it controls the time step length and, in general, it must be less than 1. The actual limit can be inferred from Table 2.1. In the case of unsplit Runge-Kutta time stepping, for instance, `CFL` $\lesssim 1/N_{\rm dim}$ where $N_{\rm dim}$ is the number of spatial dimensions while for dimensionally split methods one has `CFL` $\lesssim 1$. Certain combinations of algorithms may have more stringent limitations: a second-order Runge-Kutta algorithm with parabolic reconstruction, for example, requires `CFL` $\lesssim 0.4$ for stability reasons.
- `CFL_max_var` (*double*)
  Maximum value allowed for $\Delta t^n/\Delta t^{n-1}$ (maximum time step growth between two consecutive steps).
- `CFL_par` (*double*) [optional]
  When parabolic terms are integrated using operator splitting (with Super-Time-Stepping, §4.4.2), it controls the diffusion Courant number. The default value is $0.8/N_{\rm dim}$. This parameter has no effect when parabolic terms are included via standard explicit integration.
- `rmax_par` (*double*) [optional]
  When parabolic terms are integrated using operator splitting, it sets the maximum ratio between the actual time step and the explicit parabolic time step (i.e. $\Delta t^n/\Delta t^n_{\rm par}$). The default value is 100. This parameter has no effect when parabolic terms are included via standard explicit integration.
- `tstop` (*double*)
  integration ends when $t =$ **tstop**, unless a maximum number of steps (§1.4) is given. **tstop** has to be > 0.0.
- `first_dt` (*double*)
  The initial time step. A typical value is 10^{-6}.

2.3.4 The *[Solver]* Block

- `Solver` (*string*)
  sets the Riemann solver for flux computation. From the most accurate (i.e. least diffusive) to the least accurate (i.e. most diffusive):
  - *`two_shock`*: The Riemann problem is solved exactly or approximately (depending on the particular solver implemented for a given physics module) at every interface; this is usually more accurate, but computationally intensive. See [8] for the HD module, and [25] for the relativistic hydro equations;

	two_shock	roe	ausm+	hlld	hllc	hll	tvdlf
HD	$\surd$	$\surd$	$\surd$	-	$\surd$	$\surd$	$\surd$
MHD	-	$\surd$	-	$\surd$	$\surd$	$\surd$	$\surd$
RHD	$\surd$	-	-	-	$\surd$	$\surd$	$\surd$
RMHD	-	-	-	$\surd$	$\surd$	$\surd$	$\surd$

Table 2.3: Available Riemann solvers for the different physics module.

- *roe*: Linearized Roe Riemann solver based on characteristic decomposition of the Roe matrix, [41].
- *ausm+*: Advection Upstream Splitting Method of [19] (only for the HD module);
- *hlld*: The hlld approximate Riemann solver of [36] (for the adiabatic case), [27] (for the isothermal case) and [32] for the relativistic MHD equations;
- *hllc*: Harten, Lax, Van Leer approximate Riemann Solver with the contact discontinuity;
- *hll*: Harten, Lax, Van Leer approximate Riemann Solver;
- *tvdlf*: A simple Lax-Friedrichs scheme is used.

Note that not all the solvers are available for a given physics module, take a look at Table 2.3. We warn the user that, under some circumstances (high Mach number flows, low density plasmas), more diffusive solvers such as HLL or TVDLF turn out to be more robust than accurate solvers. However, hybrid/adaptive strategies can be turned on when `SHOCK_FLATTENING` is set to *MULTID*, §2.1.11.

2.3.5 The *[Boundary]* Block

Specifies the type of boundary condition to be applied in the physical ghost zones of the computational domain:

- `X1-beg` (*string*)
- `X1-end` (*string*)
- `X2-beg` (*string*)
- `X2-end` (*string*)
- `X3-beg` (*string*)
- `X3-end` (*string*)

Assuming that q is a scalar quantity and n is the coordinate direction orthogonal to the boundary plane, `string` can be one of:

- *outflow*
  set zero gradient across the boundary: $\frac{\partial q}{\partial n} = 0\,,\quad \frac{\partial \boldsymbol{v}}{\partial n} = 0\,,\quad \frac{\partial \boldsymbol{B}}{\partial n} = 0$
- *reflective*
  reflective (rigid walls) boundary conditions. Variables are symmetrized across the boundary and normal components of vector fields flip signs,

$$
q \to q\,,\quad \left\{\begin{array}{l} v_n \to -v_n \\ B_n \to -B_n \end{array}\right.\,,\quad \left\{\begin{array}{l} \boldsymbol{v}_t \to \boldsymbol{v}_t \\ \boldsymbol{B}_t \to \boldsymbol{B}_t \end{array}\right.
$$

  where n (t) is normal (tangential) to the interface.
- *axisymmetric*
  same as *reflective*, except for the angular component of v_ϕ or B_ϕ which also changes sign:

$$
q \to q\,,\quad \left\{\begin{array}{l} v_n \to -v_n \\ B_n \to -B_n \end{array}\right.\,,\quad \left\{\begin{array}{l} v_\phi \to -v_\phi \\ B_\phi \to -B_\phi \end{array}\right.\,,\quad \left\{\begin{array}{l} v_{axis} \to v_{axis} \\ B_{axis} \to B_{axis} \end{array}\right.
$$

  where $axis$ is $(r = 0, z)$ or $(r, \theta = 0)$ in cylindrical or spherical coordinates.

- *`eqtsymmetric`*
  sets equatorial symmetry with respect to a given plane. It is similar to *`reflective`*, but with reversed sign for the magnetic field:

$$q \to q\,, \quad \left\{ \begin{array}{l} v_n \to -v_n \\ B_n \to B_n \end{array} \right. , \quad \left\{ \begin{array}{l} \boldsymbol{v}_t \to \boldsymbol{v}_t \\ \boldsymbol{B}_t \to -\boldsymbol{B}_t \end{array} \right.$$

- *`periodic`*: periodic.
- *`shearingbox`*: Shearingbox boundary conditions are similar to periodic, except that they are sheared in one direction (only `X1-beg` and `X1-end` support this type at this moment). This particular boundary condition can be used only if the ShearingBox module (described in §5.1) is enabled.
- *`userdef`*: user-supplied boundary conditions (it requires coding your own boundary conditions in the function **`UserDefBoundary()`** in init.c, §2.4.2).

Like the *[Grid]* block, you should include the x_3 boundaries for 2D runs, even if they will not be considered.

2.3.6 The *[Static Grid Output]* Block

This block controls the output in the static grid version of the code. AMR output is controlled by another, similar block (see Chap. 7). Output files are written at specific times in the directory where pluto is executed by using one or more of the available file formats in **PLUTO** (Chapter 6). The different fields are:

- `uservar` (*`integer`*) (*`...`*)
  defines supplementary variables to be written to disk in any of the format described below. The first integer represents the number of supplementary variables. When greater than zero, it must be followed by as many variable names separated by spaces, see Chapter 6.
- `dbl` (*`double`*) (*`integer/string`*) (*`string`*)
  Assigns the output intervals for double precision (8 bytes) binary data. A negative value suppresses output.
  - The first field (*`double`*) specifies the time interval (in code units) between consecutive outputs.
  - The second field can be an *`integer`* giving the number of steps between consecutive outputs or a *`string`* giving the wall-clock time between consecutive outputs. A value, for instance, of *`2.40h`* tells **PLUTO** to write one **.dbl** file every two hours and 40 minutes. Negative values will be ignored for this control parameter.
  - The last field (*`string`*) can be either *`single_file`* (one single output file per time step containing all of the variables) or *`multiple_files`* (different variables are written to different files).
    When asynchronous I/O is enabled (§2.2.1.1), a third option *`single_file_async`* is permitted for **.flt** or **.dbl** binary files to specify that asynchrounous binary output has to be performed.

  Double-precision format files can be used to restart the code using the `-restart n` command line argument.
- `flt` (*`double`*) (*`integer/string`*) (*`string`*)
  like `dbl`, but for single-precision (4 bytes) data files.
- `vtk` (*`double`*) (*`integer/string`*) (*`string`*)
  like `dbl`, but for VTK legacy file format, see §6.1.3;

- `dbl.h5` (*`double`*) (*`integer/string`*)
  like `dbl`, but for `hdf5` double-precision format §6.1.2. This format can also be used for restarting the code by supplying the `-h5restart n` command line argument.
- `flt.h5` (*`double`*) (*`integer/string`*)
  like `dbl` but for `hdf5` single-precision format §6.1.2;
- `tab` (*`double`*) (*`integer/string`*)
  sets the time and the number of steps interval for tabulated ascii format, §6.1.4;
- `ppm` (*`double`*) (*`integer/string`*)
  sets time and the number of step intervals for 2D color images in PPM format, §6.1.5;
- `png` (*`double`*) (*`integer/string`*)
  sets time and the number of step intervals for 2D color images in PNG format §6.1.5;
- `log` (*`integer`*)
  sets the interval (in number of steps) of the integration log to screen.
- `analysis` (*`double`*) (*`integer`*)
  sets time and number of steps interval between consecutive calls to the function **`Analysis()`**.

2.3.7 The *[Chombo HDF5 output]* Block

Relevant only for AMR-Pluto with the Chombo library, see §7.2.3.

2.3.8 The *[Parameters]* Block

- `PAR_NAME_1` (*`double`*)
- ...
- `PAR_NAME_n` (*`double`*)

User-defined parameter values are read at runtime in this section. The labels on the left identify the parameter *labels* (i.e. the corresponding indices of the array `g_inputParam`) while the (*`double`*) values on the right are the actual user-defined parameter values. The number of parameters specified in this section must exactly match the number and the order given in definitions.h

2.4 STEP # 4: Problem Configuration: init.c

The source file init.c provides a set of user-supplied functions that are used to define, set and configure your specific problem. These include:

- **Init()**: sets initial conditions as functions of the spatial coordinates x_1, x_2, x_3;
- **UserDefBoundary()**: sets user-defined boundary conditions at the physical boundary sides of your computational domain if necessary (additional routines may be required, §3.2.4.3);
- **Analysis()**: run-time data analysis and reduction;
- **BodyForceVector()**, **BodyForcePotential()**: defines the vector components of the acceleration vector and/or the gravitational potential.
- **BackgroundField()**: sets a background, force-free magnetic field.

A template for all of them can be found in Src/Templates/init.c. In what follows we describe their prototypes.

2.4.1 The Init() function

Purpose:

Assign initial conditions.

Syntax:

```
void Init (double *v, double x1, double x2, double x2)
```

Arguments:

- v: a pointer to a vector of primitive quantities. A particular variable is located by means of an index: ρ =v[RHO], v_x =v[VX1], v_y =v[VX2] ... and so on. Although VX1, VX2 and VX3 can be used in any coordinate system, in order to avoid confusion, an alternative set may be adopted if the geometry is not Cartesian, see columns 2-4 in Table 2.4.

Note: PLUTO 3 Users: The old array indexing style using DN, VX, VY and VZ, etc... used in previous versions of the code can still be used for backward compatibility.

- x1,x2,x3: coordinates x_1, x_2, x_3 of the computational cell where v is initialized;

Example:

the following code sets a disk with radius 1 centered around the origin in a Cartesian domain. The disk has higher density and pressure ($\rho = 10, p = 30$) with respect to the background state ($\rho = 1, p = 1$):

```
void Init (double *v, double x1, double x2, double x3)
{
  double r;

  r = sqrt(x1*x1 + x2*x2);
  v[VX1] = v[VX2] = 0.0;
  if (r < 1.0){
    v[RHO] = 10.0;
    v[PRS] = 30.0;
  }else{
    v[RHO] = 1.0;
    v[PRS] = 1.0;
  }
}
```

Index	Cylindrical	Polar	Spherical	Quantity	Module
`RHO`	-	-	-	(proper) density	ALL
`VX1`	`iVR`	`iVR`	`iVR`	x_1-velocity	ALL
`VX2`	`iVZ`	`iVPHI`	`iVTH`	x_2-velocity	ALL
`VX3`	`iVPHI`	`iVZ`	`iVPHI`	x_3-velocity	ALL
`PRS`	-	-	-	(thermal) pressure,	ALL
`BX1`	`iBR`	`iBR`	`iBR`	x_1 cell-centered magnetic field	MHD, RMHD
`BX2`	`iBZ`	`iBPHI`	`iBTH`	x_2 cell-centered magnetic field	MHD, RMHD
`BX3`	`iBPHI`	`iBZ`	`iBPHI`	x_3 cell-centered magnetic field	MHD, RMHD
`BX1s`	`iBRs`	`iBRs`	`iBRs`	x_1 staggered magnetic field	MHD, RMHD
`BX2s`	`iBZs`	`iBPHIs`	`iBTHs`	x_2 staggered magnetic field	MHD, RMHD
`BX3s`	`iBPHIs`	`iBZs`	`iBPHIs`	x_3 staggered magnetic field	MHD, RMHD
`TRC`	-	-	-	tracer (passive scalar, Q)	ALL

Table 2.4: Array indices used for labeling primitive variables. Staggered components ("s" suffix) are used only for magnetic fields *in the boundary conditions*, see §3.2.4.3.

2.4.1.1 Assigning Initial Conditions from Input Files

With **PLUTO** 4.0 it is possible to assign initial conditions from user-supplied binary data by providing i) a grid data file and ii) a single raw binary file containing the variables to be read. The size, dimensions and even the geometry of the input grid may be different from the actual grid employed by PLUTO, as long as the coordinate transformation is supported. This provides a flexible and efficient tool to assign initial conditions by mapping data values originally defined on different computational domains. For instance, you can map a 2D spherical grid onto a 2D axisymmetric cylindrical domain, generate a 3D Cartesian domain by rotating any 2D axisymmetric data and so forth.

The module is initialized by calling the function **`InputDataSet()`** which reads and stores input grid information such as size, number of variables, geometry and dimensions. This function should be called only once from your **`Init()`** function:

```
InputDataSet (grid_file, input_var);
```

where the first argument `grid_file` is a string giving the name of the input grid file while `input_var` is an array of integers specifying which variables are contained in the input data file. The input grid file should be written using the same format employed by **PLUTO** 4.0, see §6.1.6.

After initialization, any subsequent call to

```
InputDataRead (data_file);
```

will read and store into memory the values of the variables contained in the input data file `data_file`. Unless the input data file is changed, this function should also be called only once. The data file should be written using binary format using either single or double precision with extensions ".flt" (for the former) or ".dbl" (for the latter). Variables should be stored sequentially and their order is specified by the elements of the array `input_var` until the value -1 is encountered. You may provide only some of the variables used by **PLUTO** and not necessarily all of them. The number of elements per variable should exactly match the number of grid points defined by the input grid.

Finally, the function **`InputDataInterpolate()`** is used to map the values of the variables contained in the input binary data on the grid employed by PLUTO using bi- or tri-linear interpolation at the desired coordinate location:

```
InputDataInterpolate (v, x1, x2, x3);
```

where `v` is the same array of primitive variables used in the **`Init()`** function and `x1,x2,x3` are the coordinates at which interpolates are required.

In the example below, density and velocity are assigned from the input binary file tmp/data.0010.dbl defined on the computational domain specified in tmp/grid.out:

```
void Init (double *v, double x1, double x2, double x3)
{
  static int first_call = 1;

  if (first_call){
    int k, input_var[256];
    for (k = 0; k < 256; k++) input_var[k] = -1;

    input_var[0] = RHO;
    input_var[1] = VX1;
    input_var[2] = VX2;
    input_var[3] = VX3;

    input_var[4] = -1;

    InputDataSet  ("./tmp/grid.out", input_var);
    InputDataRead ("./tmp/data.0010.dbl");
    first_call = 0;
  }

  InputDataInterpolate(v, x1, x2, x3);
  .
  .
  .
}
```

Beware that interpolation is performed only on the variables specified by the array `input_var[]`. The remaining variables (if any) must still be set inside **`Init()`**.

Note: When the input geometry differs from the one used by **PLUTO** , vector components are *not* automatically transformed to the current geometry.

A configuration example may be found in the Test_Problems/HD/Blast/ directory, where the initial condition sets an isothermal blast wave propagating in a non-uniform density medium. The inital density distribution is created by the separate file Turbulence.c in the same directory and interpolated at runtime by **PLUTO** using the method outlined above.

Note: Staggered magnetic fields may not be assigned in this way since the divergence free condition is not necessarily maintained. Using the vector potential components is more advisable.

2.4.2 The `UserDefBoundary()` function

Purpose:

Assigns user-defined boundary conditions to one or more physical boundaries of the computational domain, see Fig 2.5. Required only if one or more physical boundaries have been tagged "**userdef**" inside your pluto.ini.

Alternatively, this function may also be used to control the solution array at the beginning of every time step inside the computational domain (set floor values, override the solution, etc...).

Syntax:

```
void UserDefBoundary (const Data *d, RBox *box, int side, Grid *grid)
```

Arguments:

- `*d`: a pointer to the **PLUTO** data structure, containing:
  - `d->Vc[nv][k][j][i]`: a four-index array of primitive variables defined at the cell center. The integer `nv=RHO`, `VX1`, ..., `NVAR-1` labels the variable (see Table 2.4), while `k`, `j` and `i` are the zone indices of the x_3, x_2 and x_1 direction (note the reversed order), respectively.
  - `d->Vs[nv][k][j][i]` (staggered MHD only): a four-index array containing the three components of the staggered magnetic field (`BX1s`, `BX2s`, `BX3s`, if any) defined at zone faces, see Fig 2.5. These components only exists in the MHD or RMHD modules when using the Constrained Transport algorithm to control the $\nabla \cdot \boldsymbol{B} = 0$ condition, see §3.2.4.3 for more details.

  Important: Face-centered (staggered) magnetic fields and cell-centered fluid variables are defined on different zone stencils, see Figure 2.5. The zone-centering and the corresponding index range is encoded in the `box` structure (see below).

  We recommend to assign all the variables in use at a user-defined boundary, with the exception of `PSI_GLM` (which is optional) and the staggered component of magnetic field normal to the interface if you are using the Constrained Transport (CT) method, see §3.2.4.3.

- `*box`: a pointer to a `RBox` structure, defining the rectangular portion of the domain over which ghost zone values should be assigned. Since cell-centered and face-centered data are defined on different `box` structures, its usage is maily intended to
  - discriminate between cell-centered variables and face-centered variables using the structure member `box->vpos` which specifies the location of the variable inside the cell (=*`CENTER`*, *`X1FACE`*, *`X2FACE`*, *`X3FACE`*);
  - provide an efficient way of looping through the ghost boundary zones using the macro `BOX_LOOP(box,k,j,i)` which automatically takes care of the index range of definition.

Note: Using the `box` structure is not strictly mandatory and the usual macros `X1_BEG_LOOP()`, ..., `X3_END_LOOP()` may still be employed without any modifications. However, these macros perform loops over cell-centered data stencils and staggered field are not completely defined since the loops do not include one row of zones at the furthest left edges of the boundary zones. On the contrary, the `BOX_LOOP()` macro takes into account the full range of definition of the variable and is preferred whenever possible.

- `side`: an input integer label specifying on which side of the physical domain user-defined values should be prescribed. It can take on the following values:
  - *`X1_BEG`*, *`X1_END`*: boundary conditions can be assigned in the ghost zones at the beginning and end of the physical domain in the x_1 direction

  - *X2_BEG*, *X2_END*: boundary conditions can be assigned in the ghost zones at the beginning and end of the physical domain in the x_2 direction
  - *X3_BEG*, *X3_END* boundary conditions can be assigned in the ghost zones at the beginning and end of the physical domain in the x_3 direction
  - 0 (zero): change/control the solution inside the computational domain. This feature can be used *only* if the macro `INTERNAL_BOUNDARY` has been enabled in your definitions.h, see §2.4.2.1.

  If, say, `X1-beg` has been tagged *`userdef`* inside your pluto.ini, the user has to specify the boundary values at the beginning of the x_1 direction when `side==X1_BEG`.

- `*grid`: a pointer to an array of `Grid` structures containing all the relevant grid information. In this case, `grid[IDIR]` is the structure relevant to the $x_1$direction, `grid[JDIR]` to the x_2 direction and `grid[KDIR]` pertains to the x_3 direction. See the code documentation for more details on the members of the `Grid` structure.

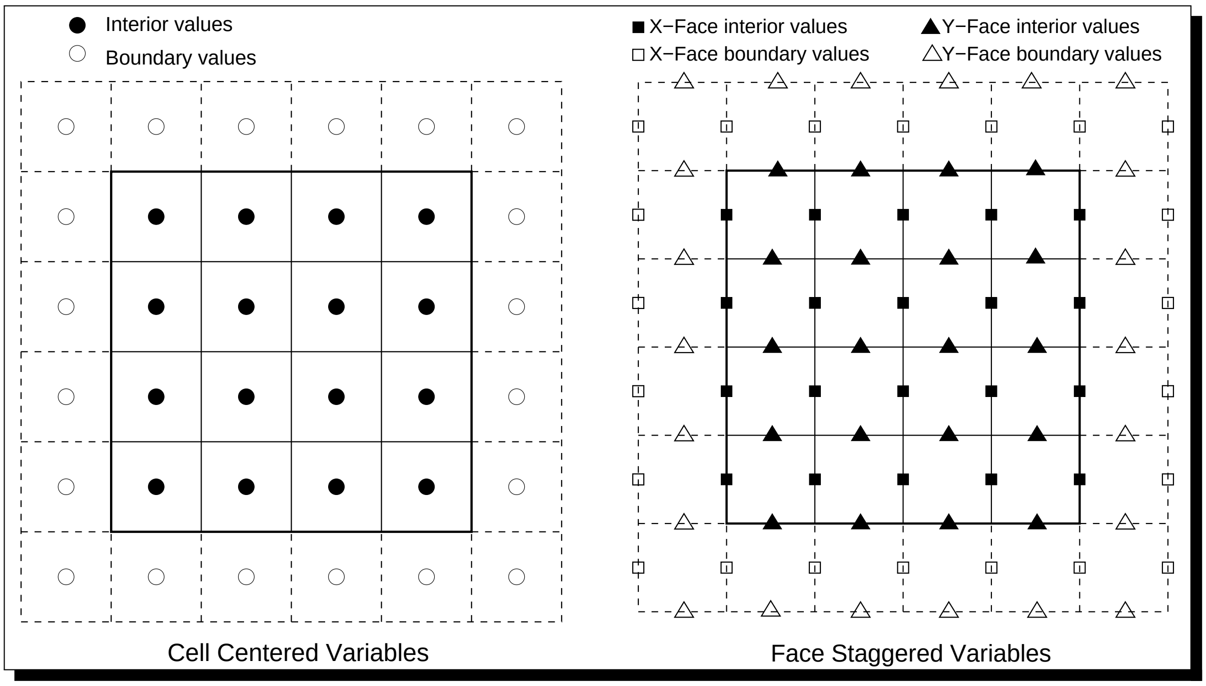


Figure 2.5: Schematic representation of cell-centered (left panel) and face-centered (right panel) collocation of physical variables on a 2D grid. X and Y-face centered staggered quantities are shown by squares and triangles, respectively. Filled symbols (circles, boxes and triangles) are considered interior values part of the solution, whereas boundary values are identified by empty symbols and must be prescribed by the user if the boundary is **userdef**.

Example #1:
As a first example we show how to prescribe a fixed inflow boundary condition for a jet model. The computational domain is a 2D box in cylindrical geometry, so that $x_1 \equiv R$, $x_2 \equiv z$. A constant inflow is prescribed a the jet nozzle located at the lower z boundary for $R \leq 1$ while reflective boundary conditions are assigned for $R > 1$. The inflow values are specified as

$$\left(\begin{array}{c}\rho\\ v_R\\ v_z\\ p\end{array}\right) = \left(\begin{array}{c}1\\ 0\\ M\\ 1/\Gamma\end{array}\right) \quad \text{for} \quad R \leq 1\,, \qquad \left(\begin{array}{c}\rho(R,-z)\\ v_R(R,-z)\\ v_z(R,-z)\\ p(R,-z)\end{array}\right) = \left(\begin{array}{c}\rho(R,z)\\ v_R(R,z)\\ -v_z(R,z)\\ p(R,z)\end{array}\right) \quad \text{for} \quad R > 1$$

where M =g_inputParam[MACH] is specified as a user-defined parameter.

```
void UserDefBoundary (const Data *d, RBox *box, int side, Grid *grid)
{
  int   i, j, k, nv;
  double  *x1, *x2, *x3;

  x1 = grid[IDIR].x;  /* -- array pointer to x1 coordinate -- */
  x2 = grid[JDIR].x;  /* -- array pointer to x2 coordinate -- */
  x3 = grid[KDIR].x;  /* -- array pointer to x3 coordinate -- */

  if (side == X2_BEG){     /* -- select the boundary side -- */
    BOX_LOOP(box,k,j,i){   /* -- Loop over boundary zones -- */
      if (x1[i] <= 1.0){   /* -- set jet values for r <= 1 -- */
        d->Vc[RHO][k][j][i]  = 1.0;
        d->Vc[iVR][k][j][i] = 0.0;
        d->Vc[iVZ][k][j][i] = g_inputParam[MACH];
        d->Vc[PRS][k][j][i]  = 1.0/gmm;
      }else{                /* -- reflective boundary for r > 1 --*/
        d->Vc[RHO][k][j][i] =  d->Vc[RHO][k][2*JBEG - j - 1][i];
        d->Vc[iVR][k][j][i] =  d->Vc[iVR][k][2*JBEG - j - 1][i];
        d->Vc[iVZ][k][j][i] = -d->Vc[iVZ][k][2*JBEG - j - 1][i];
        d->Vc[PRS][k][j][i] =  d->Vc[PRS][k][2*JBEG - j - 1][i];
      }
    }
  }
}
```

This example configuration may be found inside Test_Problems/HD/Jet. Note that the previous piece of code is executed *only* if you have selected *userdef* at the X2-beg boundary inside your pluto.ini.

The macro BOX_LOOP(box,k,j,i) performs a loop over the bottom boundary zones and, for cell-centered data, it is equivalent to the macro X2_BEG_LOOP(k,j,i). Similar macros may be used at any of the other boundaries (X1_BEG, X1_END, X2_END, X3_BEG, X3_END), although the BOX_LOOP() macro has the advantage of being more general since it automatically embeds the stencil index range for the corresponding variable location (i.e. centered or staggered).

Example #2:
As a second example, we discuss the user-defined boundary condition employed in the shock-cloud problem (Test_Problems/MHD/Shock_Cloud/). Here we want to prescribe, at the X1-end boundary, constant pre-shock values on both cell-centered quantities and staggered magnetic fields. The variable box->vpos is used to select the desired data set.

```
void UserDefBoundary (const Data *d, RBox *box, int side, Grid *grid)
{
  int   i, j, k;

  if (side == X1_END){          /* -- select the boundary side -- */
    if (box->vpos == CENTER){   /* -- select the variable position -- */
      BOX_LOOP(box,k,j,i){      /* -- Loop over boundary zones -- */
        d->Vc[RHO][k][j][i] = 1.0;
        EXPAND(d->Vc[VX1][k][j][i] = -11.2536;  ,
               d->Vc[VX2][k][j][i] = 0.0;       ,
               d->Vc[VX3][k][j][i] = 0.0;)
        d->Vc[PRS][k][j][i] = 1.0;
        EXPAND(d->Vc[BX1][k][j][i] = 0.0;       ,
               d->Vc[BX2][k][j][i] = g_inputParam[B_PRE]; ,
               d->Vc[BX3][k][j][i] = g_inputParam[B_PRE];)
      }
    }else if (box->vpos == X2FACE){  /* -- y staggered field -- */
      #ifdef STAGGERED_MHD
       BOX_LOOP(box,k,j,i) d->Vs[BX2s][k][j][i] = g_inputParam[B_PRE];
      #endif
    }else if (box->vpos == X3FACE){  /* -- z staggered field -- */
      #ifdef STAGGERED_MHD
       BOX_LOOP(box,k,j,i) d->Vs[BX3s][k][j][i] = g_inputParam[B_PRE];
      #endif
    }
  }
}
```

As in the previous example, the macro BOX_LOOP() is interchangable, for cell-centered data (box->vpos == CENTER), with the macro X1_END_LOOP(k,j,i) but not rigorously for staggered magnetic fields which are defined on a larger stencil.

The macro `EXPAND(a,b,c)` allows to write component-independent code by conditionally compiling one, two or three lines (separated by commas) depending on the value taken by `COMPONENTS`. Additional macros may be found in the code documentation.

2.4.2.1 Internal Boundary

When **`UserDefBoundary()`** is called with `side==0` and the macro `INTERNAL_BOUNDARY` has been turned to *`YES`* inside your definitions.h, the user has full contol over the solution array. This feature can be used to modify or even overwrite some or all of the cell-centered primitive variables inside a specific region of the computational domain rather than at boundaries. In this case, the **`TOT_LOOP()`** macro should be employed to loop over the (local) computational domain and a user-defined criterion (typically spatially- or variable-dependent) is used to modify the solution array in the selected zones.

A typical example may occur when a lower (or upper) threshold value should be imposed on physical variables such as density, pressure or temperature. For instance, the following piece of code sets a floor value of 10^{-3} on density:

```
void UserDefBoundary (const Data *d, RBox *box, int side, Grid *grid)
{
  int i,j,k;

  if (side == 0){
    TOT_LOOP(k,j,i){
      if (d->Vc[DN][k][j][i] < 1.e-3) {
         d->Vc[DN][k][j][i] = 1.e-3;
      }
    }
  }
...
```

A more complex example consists of a time-independent region of space where variables are fixed in time and should not be evolved by the algorithm. If this is the case, you may additionally tell **PLUTO** not to update the solution in the specified computational zones during the current time step by enabling the `FLAG_INTERNAL_BOUNDARY` flag.

Example:
The following example (taken from Test_Problems/HD/Stellar_Wind) shows how to set up a radially symmetric spherical wind in cylindrical coordinates inside a small spherical region of radius 1 centered around the origin. This is achieved by prescribing fixed inflow values for density, pressure and velocity:

```
void UserDefBoundary (const Data *d, RBox *box, int side, Grid *grid)
{
  int   i, j, k, nv;
  double *x1, *x2, *x3;
  double  r, r0, cs;
  double  Vwind  = 1.0, rho, vr;

  x1 = grid[IDIR].xgc;
  x2 = grid[JDIR].xgc;
  x3 = grid[KDIR].xgc;

  if (side == 0){
    r0 = 1.0;
    cs = g_inputParam[CS_WIND];
    TOT_LOOP(k,j,i){
      r  = sqrt(x1[i]*x1[i] + x2[j]*x2[j]);
      if (r <= r0){
        vr    = tanh(r/r0/0.1)*Vwind;
        rho   = Vwind*r0*r0/(vr*r*r);
        d->Vc[RHO][k][j][i] = rho;
        d->Vc[VX1][k][j][i] = Vwind*x1[i]/r;
        d->Vc[VX2][k][j][i] = Vwind*x2[j]/r;
        d->Vc[PRS][k][j][i] = cs*cs/g_gamma*pow(rho,g_gamma);
        d->flag[k][j][i]   |= FLAG_INTERNAL_BOUNDARY;
      }
    }
  }
...
```

The symbol `|=` (a combination of the bitwise OR operator `|` followed by the equal sign) turns the `FLAG_INTERNAL_BOUNDARY` bit on in the 3D array `d->flag[][][]`. This is used by the code to reset the right hand side of the conservative equations in the selected zones to zero. These computational cells are thus not evolved in time by **PLUTO** .

Note: The `*box` structure should not be used here and staggered magnetic field variables should not be altered.

2.4.3 The `BodyForce...()` functions

Body forces are introduced by enabling the `BODY_FORCE` flag in your definitions.h. The force is computed in terms of the acceleration vector $\boldsymbol{a}$:

$$\boldsymbol{a} = -\nabla\Phi + \boldsymbol{g}\,, \tag{2.2}$$

where Φ is the scalar potential and $\boldsymbol{g} = (g_1, g_2, g_3)$ is a three-component acceleration vector.

- The scalar potential can be employed when the `BODY_FORCE` flag is set to *`POTENTIAL`* in your definitions.h. The function **`BodyForcePotential()`** should be used to prescribe the analytical form of $\Phi \equiv \Phi(x_1, x_2, x_3)$:

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

  where `x1,x2,x3` are the local zone coordinates and the return value of the function gives the potential. In this way, **PLUTO** employs a conservative discretization that conserves total energy+gravitational energy, see Eq. (3.1) and Eq. (3.7).

  As an example, a spherically symmetric point-mass potential $\Phi = -1/r$ can be defined using

  ```
  double BodyForcePotential(double x1, double x2, double x3)
  {
    return -1.0/x1;
  }
  ```

- The acceleration vector can be employed when the `BODY_FORCE` flag is set to *`VECTOR`* and the three components of $\boldsymbol{g}$ are prescribed using the function **`BodyForceVector()`**:

  ```
  void BodyForceVector(double *v, double *g,
                       double x1, double x2, double x3)
  ```

  where

  – `*v`: a pointer to a vector of primitive quantities (e.g., `v[RHO]`, `v[VX1]`, etc...);
  – `*g`: a three-component array (`g[IDIR]`, `g[JDIR]`, `g[KDIR]`) specifying the gravity vector $\boldsymbol{g}$ components along the coordinate directions;
  – `x1,x2,x3`: local zone coordinates.

It is also possible to prescribe the body force in terms of a vector *and* a potential by setting, in your definitions.h, `BODY_FORCE` to *`(VECTOR+POTENTIAL)`*.

Note: Non-intertial effects due to a rotating frame of reference (such as Coriolis and centrifugal forces) should *not* be specified here since they are automatically handled by **PLUTO** by enabling the `ROTATING_FRAME` flag in the HD and MHD module, see §3.1.2.

2.4.4 The `Analysis()` function

Purpose:

Performs data processing at run-time. pluto.ini sets how often this function has to be called.

Syntax:

```
void Analysis (const Data *d, Grid *grid)
```

3. Basic Physics Modules

In this chapter we describe the basic equation modules available in the **PLUTO** code for the solution of the fluid equations under different regimes: HydroDynamics (HD), MagnetoHydroDynamics (MHD) and their relativistic extensions (RHD and RMHD).

We remind that only first-order spatial derivatives accounting for the hyperbolic part of the equations are described in this chapter whereas the reader is referred to Chap. 4 for a comprehensive description of the diffusion terms (thermal conduction, viscosity and magnetic resistivity) and cooling.

3.1 The HD Module

The HD module implements and solves the Euler or Navier-Stokes equations of classical fluid dynamics. The relevant source files and definitions for this module can be found in the Src/HD directory.

3.1.1 Equations

With the HD module, **PLUTO** evolves in time following system of conservation laws:

$$\frac{\partial}{\partial t}\left(\begin{array}{c}\rho\\ \boldsymbol{m}\\ E+\rho\Phi\end{array}\right)+\nabla\cdot\left(\begin{array}{c}\rho\boldsymbol{v}\\ \boldsymbol{m}\boldsymbol{v}+p\mathsf{I}\\ (E+p+\rho\Phi)\boldsymbol{v}\end{array}\right)^T=\left(\begin{array}{c}0\\ -\rho\nabla\Phi+\rho\boldsymbol{g}\\ \boldsymbol{m}\cdot\boldsymbol{g}\end{array}\right) \tag{3.1}$$

where ρ is the mass density, $\boldsymbol{m}=\rho\boldsymbol{v}$ is the momentum density, $\boldsymbol{v}$ is the velocity, p is the thermal pressure and E is the total energy density:

$$E=\rho\epsilon+\frac{\boldsymbol{m}^2}{2\rho}\,. \tag{3.2}$$

An equation of state provides the closure $p=p(\rho,\epsilon)$.

The source term on the right includes contributions from body forces and is written in terms of the (time-independent) gravitational potential Φ and and the acceleration vector $\boldsymbol{g}$ (§2.4.3).

The right hand side of the system of Eqns (3.1) is implemented in the **`RightHandSide()`** function inside Src/HD/rhs.c employing a conservative discretization that closely follows the expression given in §A.1.1, §A.1.2 and §A.1.3 for Cartesian, polar and spherical geometries (without magnetic fields).

Primitive variables are defined by $\boldsymbol{V}=(\rho,\boldsymbol{v},p)^T$, where $\boldsymbol{v}=\boldsymbol{m}/\rho$ and

$$p\equiv\rho\epsilon(\Gamma-1)=(\Gamma-1)\left(E-\frac{\boldsymbol{m}^2}{2\rho}\right)$$

for an ideal equation of state with Γ being the specific heat ratio. The maps $\boldsymbol{U}(\boldsymbol{V})$ and its inverse are provided by the functions **`PrimToCons()`** and **`ConsToPrim()`**.

Primitive variables are generally more convenient and preferred when assigning initial/boundary conditions and in the interpolation algorithms. The vector of primitive quantities $\boldsymbol{V}$ obeys the quasi-linear form of the equations:

$$\begin{array}{rcl}\displaystyle\frac{\partial\rho}{\partial t}+\boldsymbol{v}\cdot\nabla\rho+\rho\nabla\cdot\boldsymbol{v} &=& 0\\ \displaystyle\frac{\partial\boldsymbol{v}}{\partial t}+\boldsymbol{v}\cdot\nabla\boldsymbol{v}+\frac{\nabla p}{\rho} &=& -\nabla\Phi+\boldsymbol{g}\\ \displaystyle\frac{\partial p}{\partial t}+\boldsymbol{v}\cdot\nabla p+\rho c_s^2\nabla\cdot\boldsymbol{v} &=& 0\,,\end{array} \tag{3.3}$$

where $c_s = \sqrt{\Gamma p/\rho}$ is the adiabatic speed of sound. The quasi-linear form (3.3) is required during the predictor stage when `TIME_EVOLUTION` has been set to either *`CHARACTERISTIC_TRACING`* or *`HANCOCK`* and is implemented in the Src/HD/prim_eqn.c source file.

3.1.2 Available Options

The HD sub-menu available with the Python script allows to enable some additional features for this module:

- `EOS`
  select the Equation of State:
  - *`IDEAL`*: adiabatic equation of state for which the internal energy obeys $\rho\epsilon = p/(\Gamma - 1)$ where Γ is constant specific heat ratio. The global variable `g_gamma` sets the numerical value of Γ and can be assigned in the **`Init()`** function (default $5/3$).
  - *`ISOTHERMAL`*: isothermal equation of state $p = c_{\rm iso}^2\rho$, where $c_{\rm iso}$ (the isothermal sound speed) can be either a constant value or a spatially-varying quantity. By default, $c_{\rm iso} = 1$ is a constant value that can be changed using the global variable `g_isoSoundSpeed` in your init.c, e.g.

    ```
    g_isoSoundSpeed = 2.0; /* sets the sound speed to be 2 */
    ```

    In order to have a space-dependent isothermal speed of sound, one has to copy the source file Src/HD/eos.c to your local working directory and make the appropriate modification.

    Beware that, when this EoS is selected, no energy equation is present and the labels `EN` and `PR` are undefined.

- `ENTROPY_SWITCH (`*`YES`*`/`*`NO`*`)`
  By enabling this switch, the equation of conserved entropy is added to the system of conservation laws:
  $$\frac{\partial(\rho s)}{\partial t} + \nabla\cdot(\rho s\boldsymbol{v}) = 0 \tag{3.4}$$
  where $s = p/\rho^\Gamma$ is the entropy variable for a constant Γ-law. Equations (3.4) is solved by treating entropy as a passive scalar. At the end of the integration step, gas pressure is recovered from either total energy or conserved entropy, according to the control strategy specified in Src/entropy_switch.c which, by default, uses entropy everywhere except at shocks. In other words, if a zone has been flagged with `FLAG_ENTROPY` (by default away from shocks), then $p = p(s)$ and total energy is recomputed using this new value of pressure. Otherwise, $p = p(E)$ and entropy is recomputed using the new pressure value. Note that by enabling this mixed evolution, neither the total energy nor the entropy will generally be conserved at the numerical level. Also, beware that in the current code release, the `ENTROPY_SWITCH` is not compatible if used in conjuction with diffusion operators.

- `THERMAL_CONDUCTION`
  include thermal conduction effects, see §4.3. The available options are
  - *`NO`*: thermal condution is not included;
  - *`EXPLICIT`*: thermal conduction is treated explicitly, §4.4.1;
  - *`SUPER_TIME_STEPPING`*: thermal conduction is treated using super-time-stepping, §4.4.2.

- `VISCOSITY`
  include viscous terms, see §4.1. Options are
  - *`NO`*: viscous terms are not included;
  - *`EXPLICIT`*: viscosity is treated explicitly, §4.4.1;
  - *`SUPER_TIME_STEPPING`*: viscosity is treated using super-time-stepping, §4.4.2.

See §4.1 for details.

- `ROTATING_FRAME` (*`YES`*/*`NO`*)
  Solves the equations in a frame of reference rotating with constant angular velocity Ω_z around the vertical polar axis z. This feature should be enabled only when `GEOMETRY` is one of *`CYLINDRICAL`*, *`POLAR`* or *`SPHERICAL`*. The value of Ω_z is specified using the global variable `g_OmegaZ` inside your **`Init()`** function. The discretization of the angular momentum and energy equations is then done in a conservative fashion [22, 30]. For example, in polar geometry, we solve

$$\begin{aligned}
\frac{\partial}{\partial t}(\rho v_R) + \nabla\cdot(\rho v_R \boldsymbol{v}) + \frac{\partial p}{\partial R} &= \frac{\rho(v_\phi + w)^2}{R} \\
\frac{\partial}{\partial t}\big[R\rho(v_\phi + w_z)\big] + \nabla\cdot\left[R\rho(v_\phi + w)\boldsymbol{v}\right] + \frac{\partial p}{\partial \phi} &= 0 \\
\frac{\partial}{\partial t}\left(E + \frac{w_z^2}{2}\rho + w\rho v_\phi\right) + \nabla\cdot\left[\boldsymbol{F}_E + \frac{w^2}{2}\rho\boldsymbol{v} + w\rho v_\phi \boldsymbol{v}\right] &= 0
\end{aligned} \tag{3.5}$$

  where $w = R\Omega_z$, R is the cylindrical radius and $\boldsymbol{F}_E$ is the standard energy flux and body force terms have been omitted only for the sake of exposition.

  Note that the source term in the radial component of the momentum equation implicitly contains the Colios force and centrifugal terms:

$$\frac{\rho(v_\phi + w)^2}{R} = \frac{\rho v_\phi^2}{R} + 2\rho v_\phi \Omega_z + \rho\Omega_z^2 R \tag{3.6}$$

  On the other hand, the azimuthal component of the Coriolis force has been incorporated directly into the fluxes using the conservation form. An example of such a configuration in polar or spherical geometry may be found in the directory Test_Prob/HD/DiskPlanet.

3.2 The MHD Module

The MHD module is suitable for the solution of the ideal or resistive (non-relativistic) magnetohydrodynamical equations. Source and definition files are located inside the Src/MHD directory tree.

3.2.1 Equations

With the MHD module, **PLUTO** solves the following system of conservation laws:

$$\frac{\partial}{\partial t}\left(\begin{array}{c}\rho\\ \boldsymbol{m}\\ E+\rho\Phi\\ \boldsymbol{B}\end{array}\right)+\nabla\cdot\left(\begin{array}{c}\rho\boldsymbol{v}\\ \boldsymbol{m}\boldsymbol{v}-\boldsymbol{B}\boldsymbol{B}+\mathsf{I}p_t\\ (E+p_t+\rho\Phi)\boldsymbol{v}-\boldsymbol{B}\left(\boldsymbol{v}\cdot\boldsymbol{B}\right)\\ \boldsymbol{v}\boldsymbol{B}-\boldsymbol{B}\boldsymbol{v}\end{array}\right)^T=\left(\begin{array}{c}0\\ -\rho\nabla\Phi+\rho\boldsymbol{g}\\ \boldsymbol{m}\cdot\boldsymbol{g}\\ \boldsymbol{0}\end{array}\right) \tag{3.7}$$

where ρ is the mass density, $\boldsymbol{m}=\rho\boldsymbol{v}$ is the momentum density, $\boldsymbol{v}$ is the velocity, $p_t=p+\boldsymbol{B}^2/2$ is the total pressure (thermal + magnetic), $\boldsymbol{B}$ is the magnetic field[1] and E is the total energy density:

$$E=\rho\epsilon+\frac{\boldsymbol{m}^2}{2\rho}+\frac{\boldsymbol{B}^2}{2}\,. \tag{3.8}$$

where an additional equation of state provides the closure $p=p(\rho,\epsilon)$. The source term on the right includes contributions from body forces and is written in terms of the (time-independent) gravitational potential Φ and and the acceleration vector $\boldsymbol{g}$ (§2.4.3).

Note that the induction equation may equivalently be written as

$$\frac{\partial\boldsymbol{B}}{\partial t}+\nabla\times\boldsymbol{\mathcal{E}}=0\,, \tag{3.9}$$

where $\boldsymbol{\mathcal{E}}=-\boldsymbol{v}\times\boldsymbol{B}$ is the electric field.

The right hand side of the system of Eqns (3.7) is implemented in the **`RightHandSide()`** function inside Src/MHD/rhs.c employing a conservative discretization that closely follows the expression given in §A.1.1, §A.1.2 and §A.1.3 for Cartesian, polar and spherical geometries.

The sets of conservative and primitive variables $\boldsymbol{U}$ and $\boldsymbol{V}$ are given by:

$$\boldsymbol{U}=\Big(\rho,\ \boldsymbol{m},\ E,\ \boldsymbol{B}\Big)^T\,,\quad \boldsymbol{V}=\Big(\rho,\ \boldsymbol{v},\ p,\ \boldsymbol{B}\Big)^T\,.$$

The maps $\boldsymbol{U}(\boldsymbol{V})$ and its inverse are provided by the functions **`PrimToCons()`** and **`ConsToPrim()`**.

The primitive form of the equations is

$$\begin{array}{lcl}\displaystyle\frac{\partial\rho}{\partial t}+\boldsymbol{v}\cdot\nabla\rho+\rho\nabla\cdot\boldsymbol{v} &=& 0\\ \displaystyle\frac{\partial\boldsymbol{v}}{\partial t}+\boldsymbol{v}\cdot\nabla\boldsymbol{v}+\frac{1}{\rho}\boldsymbol{B}\times(\nabla\times\boldsymbol{B})+\frac{1}{\rho}\nabla p &=& -\nabla\Phi+\boldsymbol{g}\\ \displaystyle\frac{\partial\boldsymbol{B}}{\partial t}+\boldsymbol{B}(\nabla\cdot\boldsymbol{v})-(\boldsymbol{B}\cdot\nabla)\boldsymbol{v}+(\boldsymbol{v}\cdot\nabla)\boldsymbol{B} &=& \boldsymbol{v}\,(\nabla\cdot\boldsymbol{B})\\ \displaystyle\frac{\partial p}{\partial t}+\boldsymbol{v}\cdot\nabla p+\rho c_s^2\nabla\cdot\boldsymbol{v} &=& 0\,,\end{array} \tag{3.10}$$

where the $(\nabla\cdot\boldsymbol{B})$ on the right hand side of the third equation is kept for reasons of convenience, although zero at the continuous level.

[1]A factor of $1/\sqrt{4\pi}$ has been absorbed in the definition of magnetic field.

3.2.2 Available Options

When setting up your problem with Python, a second menu should follow up. This menu allows one to choose:

- `EOS`
  Select the equation of state. Options are the same as the HD module (§3.1.2).
- `ENTROPY_SWITCH` (*`YES`*/*`NO`*)
  Solve the entropy equation in addition to the total energy equation in the same way as done for the HD module, §3.1.2.
- `MHD_FORMULATION`
  Select a strategy to enforce the $\nabla \cdot \boldsymbol{B} = 0$ constraint[2]. Possible values are
  - *`NONE`*
    divergence constraint is not controlled by any algorithm. Recommended for one-dimensional problems or 2D configurations with purely azimuthal fields.
  - *`EIGHT_WAVES`*
    magnetic fields retain a cell average representation and the eight wave formulation introduced by Powell [38] is used, see §3.2.4.1;
  - *`DIV_CLEANING`*
    magnetic fields retain a cell average representation and the mixed hyperbolic/parabolic divergence cleaning technique of [13, 34] is used, see §3.2.4.2. A new scalar variable, the generalized Lagrange multiplier ψ (`PSI_GLM`) is introduced.
  - *`CONSTRAINED_TRANSPORT`*
    the magnetic field has a staggered representation and the constrained transport is used, see §3.2.4.3.
- `BACKGROUND_FIELD` (*`YES`*/*`NO`*)
  Split the magnetic field into a static contribution and a time dependent deviation, see §3.2.5 for how to use this feature.
- `RESISTIVE_MHD`
  Include resistive terms in the MHD equations, see §4.2. The available options are
  - *`NO`*: resistivity is not included;
  - *`EXPLICIT`*: resistivity is included explicitly, §4.4.1;
  - *`SUPER_TIME_STEPPING`*: resistivity is treated using super-time-stepping, §4.4.2.
- `THERMAL_CONDUCTION`
  Include anisotropic thermal conduction flux, see §4.3. Options are the same as for the HD module, §3.1.2.
- `VISCOSITY`:
  Include viscosity terms in the MHD equations, §4.1. Options are the same as for the HD module, §3.1.2.
- `ROTATING_FRAME` (*`YES`*/*`NO`*)
  Solves the equations in a rotating frame, see the HD module §3.1.2.

[2]Numerical methods do not naturally preserve the condition $\nabla \cdot \boldsymbol{B} = 0$.

3.2.3 Assigning Magnetic Field Components

Magnetic field components are initialized in your **`Init()`** function just like any other flow quantity. Depending on the value of `ASSIGN_VECTOR_POTENTIAL` in your definitions.h, two alternative initializations are possible:

1. By setting `ASSIGN_VECTOR_POTENTIAL` to *`NO`*, you can assign the component of magnetic field in the usual way by directly prescribing the values for `us[BX1]`, `us[BX2]` and `us[BX3]`.
2. When `ASSIGN_VECTOR_POTENTIAL` is set to *`YES`*, the vector potential $\boldsymbol{A}$ is used instead and the magnetic field is recovered from $\boldsymbol{B} = \nabla \times \boldsymbol{A}$. This option guarantees that the initial field has zero divergence in the discretization which is more appropriate for the underlying formulation (i.e., cell or face centered fields, §3.2.4).

Note: In 2D, only the third component of $\boldsymbol{A}$ (that is `us[AX3]`) should be used. Likewise, the third component of the magnetic field (B_z) cannot be assigned through the vector potential an must be prescribed in the standard way, see the third example in Table 3.1.

Table 3.1 shows some examples of magnetic field initializations with and without using the vector potential.

Magnetic Field	Standard	Using Vector Potential

$\boldsymbol{B} = (0, 5, 0)$
Cartesian, 2D
	`us[BX1] = 0.0;`
	`us[BX2] = 5.0;`
	`us[BX3] = 0.0;`
		`us[AX1] = 0.0;`
		`us[AX2] = 0.0;`
		`us[AX3] = -x1*5.0;`

$\boldsymbol{B} = (0, 5, 0)$
Cylindrical, 2D
	`us[BX1] = 0.0;`
	`us[BX2] = 5.0;`
	`us[BX3] = 0.0;`
		`us[AX1] = 0.0;`
		`us[AX2] = 0.0;`
		`us[AX3] = 0.5*x1*5.0;`

$\boldsymbol{B} = (-\sin y, \sin 2x, 2)$
Cartesian, 2.5D
	`us[BX1] = -sin(x2);`
	`us[BX2] = sin(2.0*x1);`
	`us[BX3] = 2.0;`
		`us[AX1] = 0.0;`
		`us[AX2] = 0.0;`
		`us[AX3] = cos(x2)+0.5*cos(2.0*x1);`
		`us[BX3] = 2.0;`

Table 3.1: Examples of how the magnetic field may be initialized. Direct initialization (standard) is possible when`ASSIGN_VECTOR_POTENTIAL` is set to *`NO`*. Otherwise, the components of the vector potential are used (third column).

3.2.4 Controlling the $\nabla \cdot \boldsymbol{B} = 0$ Condition

3.2.4.1 Eight-Wave Formulation

In the eight-wave formalism [38, 40] magnetic fields have a cell-centered representation. Additional source terms are added on the right hand side of Eqns (3.7):

$$\frac{\partial}{\partial t}\begin{pmatrix} \rho \\ \boldsymbol{m} \\ E \\ \boldsymbol{B} \end{pmatrix} + \cdots = -\nabla \cdot \boldsymbol{B} \begin{pmatrix} 0 \\ \boldsymbol{B} \\ \boldsymbol{v} \cdot \boldsymbol{B} \\ \boldsymbol{v} \end{pmatrix}$$

Contributions to $\nabla \cdot \boldsymbol{B}$ are taken direction by direction. Note that the 8-wave formulation keeps $\nabla \cdot \boldsymbol{B} = 0$ only at the truncation level and NOT to machine accuracy. More accurate treatments of the solenoidal condition can be achieved using the other two formulations. The 8-wave formulation should be used in conjunction any Riemann solver with the exception of `hlld`.

3.2.4.2 Hyperbolic Divergence Cleaning (GLM and EGLM)

In [13] (see also [33, 34] for additional discussion), the divergence free constraint is enforced by solving a modified system of conservation laws, where the induction equation is coupled to a generalized Lagrange multiplier (GLM). Using the mixed GLM hyperbolic/parabolic correction, the induction equation and the solenoidal constraint are replaced, respectively, by

$$\frac{\partial \boldsymbol{B}}{\partial t} + \nabla \cdot (\boldsymbol{vB} - \boldsymbol{Bv}) + \nabla\psi = 0\,, \qquad \frac{\partial \psi}{\partial t} + c_h^2 \nabla \cdot \boldsymbol{B} = -\frac{c_h^2}{c_p^2}\psi\,, \tag{3.11}$$

where $c_h = \mathrm{CFL} \times \Delta l_{\min}/\Delta t^n$ is maximum speed compatible with the step size, $c_p = \sqrt{\Delta l_{\min} c_h/\alpha}$ and $\Delta l_{\min}$ is the minimum cell length. The free parameter α, [33], controls the rate at which monopole are damped and has a default value 0.1. It can be easily modified by introducing the user-defined parameter `ALPHA_GLM`, §2.1, or by directly editing Src/MHD/GLM/glm_solve.c. A number of tests suggests that the optimal range can be found for $0.05 \lesssim \alpha \lesssim 0.3$. In the mixed formulation, divergence errors are transported to the domain boundaries with the maximal admissible speed and are damped at the same time.

By default, ψ is set to zero in the initial and boundary conditions but the user is free to change it at a user-defined boundary by prescribing `d→Vc[PSI_GLM][k][j][i]` (inside **`UserDefBoundary()`**) which has the usual cell-centered representation. The scalar multiplier is not written to disk except for the double format, §6, needed for restart.

The advantage of this formulation (GLM-MHD) is that the equations retain a conservative form (no source terms are introduced), all variables (including magnetic fields) retain a cell-centered representation and standard 7-wave Riemann solvers (with a single value of the normal component of magnetic field) may be used.

A slightly different formulation (EGLM-MHD), breaking momentum and energy conservation, has been found to be more robust in problems involving strongly magnetized media. The EGLM form of the equations [13, 33] can be obtained by setting `#define EGLM` to *YES* inside Src/MHD/GLM/glm.h. For a complete description of the GLM- and EGLM-MHD formulation and its implementation in **PLUTO** refer to [33, 34].

3.2.4.3 Constrained Transport (CT)

In this formulation [3, 20, 14], two sets of magnetic fields are used:

- face-centered magnetic field ($\boldsymbol{b}$ hereafter);
- cell-centered magnetic field ($\boldsymbol{B}$ hereafter).

The primary set is the first one, where the three components of the field are located at different spatial points in the control volume, that is

$$b_{x_1,i+\frac{1}{2},j,k} \quad , \qquad b_{x_2,i,j+\frac{1}{2},k} \quad , \qquad b_{x_3,i,j,k+\frac{1}{2}}$$

see Fig. 3.1. In Cartesian coordinates, for instance, b_x is located at X-faces whereas b_y lives at Y-faces, etc., see the boxes and triangles in Fig. 2.5. *This feature must be used only in conjunction with an unsplit integrator*. With CT, the solenoidal condition is maintained at machine accuracy as long as field initialization is done using the vector potential, §3.2.3.

The staggered magnetic field is treated as an area-weighted average on the zone face and Stoke's theorem is used to update it:

$$\int\left(\frac{\partial \boldsymbol{b}}{\partial t} + \nabla\times\boldsymbol{\mathcal{E}}\right)\cdot d\boldsymbol{S}_d = 0 \quad\Longrightarrow\quad \frac{db_{x_d}}{dt} + \frac{1}{S_d}\oint \boldsymbol{\mathcal{E}}\cdot d\boldsymbol{l} = 0 \tag{3.12}$$

Please note that the staggered components are initialized and integrated also on the boundary interfaces in the corresponding staggered direction. In other words, the interior values are

$$b_{x_1,i+\frac{1}{2},j,k} : \begin{cases} \text{IBEG}-1 & \leq i \leq \text{IEND} \\ \text{JBEG} & \leq j \leq \text{JEND} \\ \text{KBEG} & \leq k \leq \text{KEND} \end{cases}$$

$$b_{x_2,i,j+\frac{1}{2},k} : \begin{cases} \text{IBEG} & \leq i \leq \text{IEND} \\ \text{JBEG}-1 & \leq j \leq \text{JEND} \\ \text{KBEG} & \leq k \leq \text{KEND} \end{cases}$$

$$b_{x_3,i,j,k+\frac{1}{2}} : \begin{cases} \text{IBEG} & \leq i \leq \text{IEND} \\ \text{JBEG} & \leq j \leq \text{JEND} \\ \text{KBEG}-1 & \leq k \leq \text{KEND} \end{cases}$$

Thus $b_{x_1,i+\frac{1}{2},j,k}$ is NOT a boundary value for $i = \text{IBEG}-1$, JBEG $\leq j \leq$ JEND, KBEG $\leq k \leq$ KEND but it is considered part of the solution !! Similar considerations hold for b_{x_2} and b_{x_3} components at the x_2 and x_3 boundaries, respectively.

The electromotive force (EMF) $\mathcal{E}$ is computed at zone edges, see Fig. 3.1 by a proper averaging/reconstruction scheme (set by `CT_EMF_AVERAGE` inside your definitions.h). Options are:

- `CT_EMF_AVERAGE` = *`ARITHMETIC`* yields yields a simple arithmetic averaging [3] of the fluxes computed during the upwind steps. In this case, one has available

$$\begin{pmatrix} 0 \\ -\mathcal{E}_{x_3} \\ \mathcal{E}_{x_2} \end{pmatrix}_{i+\frac{1}{2},j,k} , \quad \begin{pmatrix} \mathcal{E}_{x_3} \\ 0 \\ -\mathcal{E}_{x_1} \end{pmatrix}_{i,j+\frac{1}{2},k} , \quad \begin{pmatrix} -\mathcal{E}_{x_2} \\ \mathcal{E}_{x_1} \\ 0 \end{pmatrix}_{i,j,k+\frac{1}{2}}$$

during the x_1, x_2 and x_3 sweeps, respectively. The arithmetic average follows:

$$\mathcal{E}_{x_1,i,j+\frac{1}{2},k+\frac{1}{2}} = \frac{1}{4}\left(\mathcal{E}_{x_1,j,k+\frac{1}{2}} + \mathcal{E}_{x_1,i,j+1,k+\frac{1}{2}} + \mathcal{E}_{x_1,i,j+\frac{1}{2},k} + \mathcal{E}_{x_1,i,j+\frac{1}{2},k+1}\right)$$

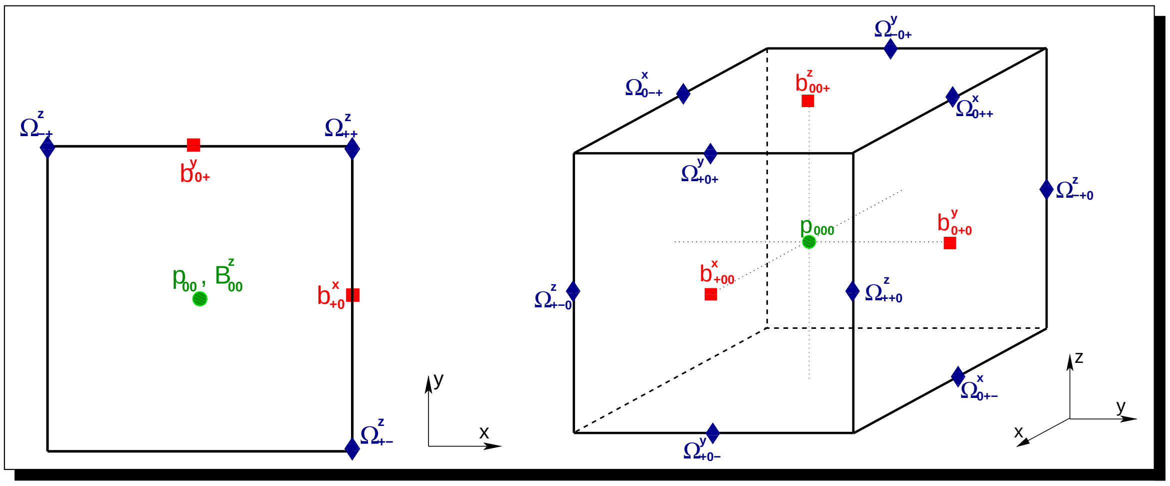


Figure 3.1: Collocation points in 2.x D (left) and in 3D (right). Cell-centered quantities are given as green circles, face-centered as red squares and edge-centered as blue diamonds.

$$\mathcal{E}_{x_2,i+\frac{1}{2},j,k+\frac{1}{2}} = \frac{1}{4}\left(\mathcal{E}_{x_2,i+\frac{1}{2},j,k} + \mathcal{E}_{x_2,i+\frac{1}{2},j,k+1} + \mathcal{E}_{x_2,i,j,k+\frac{1}{2}} + \mathcal{E}_{x_2,i+1,j,k+\frac{1}{2}}\right)$$

$$\mathcal{E}_{x_3,i+\frac{1}{2},j+\frac{1}{2},k} = \frac{1}{4}\left(\mathcal{E}_{x_3,i+\frac{1}{2},j,k} + \mathcal{E}_{x_3,i+\frac{1}{2},j+1,k} + \mathcal{E}_{x_3,i,j+\frac{1}{2},k} + \mathcal{E}_{x_3,i+1,j+\frac{1}{2},k}\right)$$

Although being the simplest one, this average procedure may suffer from insufficient dissipation in some circumstances ([14, 20]) and does not reduce to its one dimensional equivalent algorithm for plane parallel grid aligned flows.

- `CT_EMF_AVERAGE` = *`UCT_HLL`* uses a two dimensional Riemann solver based on a four-state HLL flux function, see [54, 20]. If the fully unsplit *`HANCOCK`* or *`CHARACTRISTIC_TRACING`* scheme is used, the Courant number must be $CFL \lesssim 0.7$ (in 2D) and $CFL \lesssim 0.35$ (in 3D).
- `CT_EMF_AVERAGE` = *`UCT0`* or `CT_EMF_AVERAGE` = *`UCT_CONTACT`* employs the face-to-edge integration procedures proposed by [14], where electromotive force derivatives are averaged from neighbor zones (*`UCT0`*) or selected according to the sign of the contact mode (*`UCT_CONTACT`*). The former has reduced dissipation and is preferably used with linear interpolants and RK integrators, while the latter shows better dissipation properties.

All algorithms, with the exception of the arithmetic averaging, reduce to the corresponding one dimensional scheme for grid aligned flows. However, in our experience, *`UCT_HLL`* and *`UCT_CONTACT`* show the best dissipation and stability properties. The CT formulation works with any of the Riemann solvers.

Assigning Boundary Conditions. Within the CT framework, user-defined boundary conditions (b.c.) must be assigned on the staggered components as well. This is done in your **`UserDefBoundary()`** function using the `d→Vs[nv][k][j][i]` array, where `nv` gives the staggered component: `BX1s`, `BX2s` or `BX3s`.

Note: In **PLUTO** we follow the convention that the cell "center" owns its right interface, e.g., 'i' means $i + \frac{1}{2}$. Thus:

$$b_{x_1,i+\frac{1}{2},j,k} \equiv \texttt{d}\rightarrow\texttt{Vs[BX1s][k][j][i];}$$
$$b_{x_2,i,j+\frac{1}{2},k} \equiv \texttt{d}\rightarrow\texttt{Vs[BX2s][k][j][i];}$$
$$b_{x_3,i,j,k+\frac{1}{2}} \equiv \texttt{d}\rightarrow\texttt{Vs[BX3s][k][j][i];}$$

Beware that the three staggered components have *different spatial locations* and the BOX_LOOP(box,k,j,i) macro introduced in §2.4.2 automatically implements the correct loop over the boundary ghost zones. Thus, at the x_1 boundary, for instance, one needs to assign

$$\left.\begin{array}{lll} b_{x_2,i,j+\frac{1}{2},k} & \text{at} & x_{1,i}, x_{2,j+\frac{1}{2}}, x_{3,k} \\ b_{x_3,i,j,k+\frac{1}{2}} & \text{at} & x_{1,i}, x_{2,j}, x_{3,k+\frac{1}{2}} \end{array}\right\} \quad \text{for} \quad i = 0,\cdots,\text{IBEG-1}$$

The component normal to the interface (b_{x_1} in this case) should *NOT* be assigned since it is automatically computed by **PLUTO** from the $\nabla\cdot\boldsymbol{B} = 0$ condition after the tangential components have been set.

Example:
The following example prescribes user-defined boundary conditions at the lower x_2 boundary for the MHD jet problem (see PLUTO/Test_Problems/MHD/Jet) in cylindrical coordinates ($x_1 \equiv R$, $x_2 \equiv z$). Inflow conditions are given as $(\rho, v_R, v_z, p, B_r, B_z) = (1, 0, 10, 1/\Gamma, 0, 3)$ for $R \leq 1$ while a symmetric counter-jet is assumed for $R > 1$:

```
if (side == X2_BEG){

  JETVAL(vjet);       /* -- beam/jet values -- */
  R  = grid[IDIR].x;  /* -- cylindrical radius -- */

  if (box->vpos == CENTER){  /* -- select cell-centered varaibles only -- */
    BOX_LOOP(box, k, j, i){  /* -- loop on boundary zones -- */
      for (nv = 0; nv < NVAR; nv++) vout[nv] = d->Vc[nv][k][2*JBEG-j-1][i];
      vout[VX2] *= -1.0;
      #if PHYSICS == MHD
       vout[BX1] *= -1.0;
      #endif
      for (nv = 0; nv < NVAR; nv++) /* -- smooth out the two solutions -- */
        d->Vc[nv][k][j][i] = vout[nv] + (vjet[nv] - vout[nv])*Profile(R[i],nv);
    }
  }else if (box->vpos == X1FACE){ /* -- select x1-staggered component -- */
    #ifdef STAGGERED_MHD
     Rp = grid[IDIR].A;           /* -- right interface area -- */
     BOX_LOOP(box, k, j, i){
       bxsout = -d->Vs[BX1s][k][2*JBEG - j - 1][i];
       d->Vs[BX1s][k][j][i] = bxsout*(1.0 - Profile(rp[i],BX));
     }
   }
   #endif
}
```

Here STAGGERED_MHD is defined only in the MHD constrained transport and the boundary conditions are assigned on $b_{x_1} \equiv b_R$ only (i.e. the orthogonal component).

3.2.5 Background Field Splitting

In situations where an intrinsic background magnetic field is present (e.g. planetary magnetosphere, stellar dipole fields), it may be convenient to write the total magnetic field as $\boldsymbol{B}(\boldsymbol{x},t) = \boldsymbol{B}_0(\boldsymbol{x}) + \boldsymbol{B}_1(\boldsymbol{x},t)$ where $\boldsymbol{B}_0$ is a background curl-free magnetic field and $\boldsymbol{B}_1(\boldsymbol{x},t)$ is a deviation. The background field must satisfy the following conditions:

$$\frac{\partial \boldsymbol{B}_0}{\partial t} = 0\,, \quad \nabla\cdot\boldsymbol{B}_0 = 0\,, \quad \nabla\times\boldsymbol{B}_0 = \boldsymbol{0}\,.$$

In this case one can show [38] that the MHD equations reduce to:

$$\begin{array}{rcl} \dfrac{\partial\rho}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}) &=& 0 \\ \dfrac{\partial\boldsymbol{m}}{\partial t} + \nabla\cdot\Big(\boldsymbol{m}\boldsymbol{v} - \boldsymbol{B}_1\boldsymbol{B} - \boldsymbol{B}_0\boldsymbol{B}_1\Big) + \nabla p_t &=& \rho(-\nabla\Phi + \boldsymbol{g}) \\ \dfrac{\partial(E_1 + \rho\Phi)}{\partial t} + \nabla\cdot\Big[(E_1 + p_t + \rho\Phi)\boldsymbol{v} - (\boldsymbol{v}\cdot\boldsymbol{B}_1)\boldsymbol{B}\Big] &=& \boldsymbol{m}\cdot\boldsymbol{g} \\ \dfrac{\partial\boldsymbol{B}_1}{\partial t} - \nabla\times(\boldsymbol{v}\times\boldsymbol{B}) &=& 0 \end{array}$$

where

$$p_t = p + \frac{\boldsymbol{B}_1^2}{2} + \boldsymbol{B}_1 \cdot \boldsymbol{B}_0 \,, \quad E_1 = \frac{p}{\Gamma - 1} + \frac{1}{2}\left(\rho \boldsymbol{v}^2 + \boldsymbol{B}_1^2\right)$$

Thus the energy depends only on $\boldsymbol{B}_1$, a feature that turns out to be useful when dealing with low-beta plasma. The sets of conservative and primitive variables are the same as the original ones, with $\boldsymbol{B} \to \boldsymbol{B}_1$, $E \to E_1$.

In order to enable this feature, the macro `BACKGROUND_FIELD` must be turned to *`YES`* in your definitions.h. The initial and boundary conditions must be imposed on $\boldsymbol{B}_1$ alone while the function **`BackgroundField()`** can be added to your init.c to assign $\boldsymbol{B}_0$:

```
void BackgroundField (double x1, double x2, double x3, double *B0)
```

Examples can be found in the $4^{\rm th}$ configuration of Test_Problems/MHD/Rotor/ and in the $8^{\rm th}$ configuration of Test_Problems/MHD/Blast/.

Note: This feature should only be used with the HLL Riemann solver, a Runge-Kutta time stepping and the CT formulation.

3.3 The RHD Module

The RHD module implements the equations of special relativistic fluid dynamics in 1, 2 or 3 dimensions. Velocities are always assumed to be expressed in units of the speed of light. The special relativistic module comes with 2 different equations of state, and it also works in curvilinear coordinates. Gravity can also be incorporated, but it is treated as a Newtonian approximation. The relevant source files and definitions for this module can be found in the Src/RHD directory.

3.3.1 Equations

The special relativistic module evolves the conservative set $\boldsymbol{U}$ of state variables

$$\boldsymbol{U} = \Big(D,\; m_1,\; m_2,\; m_3,\; E\Big)^T$$

where D is the laboratory density, $m_{x1,x2,x3}$ are the momentum components, E is the total energy (including contribution from the rest mass). The evolutionary conservative equations are

$$\frac{\partial}{\partial t}\left(\begin{array}{c} D \\ \boldsymbol{m} \\ E \end{array}\right) + \nabla\cdot\left(\begin{array}{c} D\boldsymbol{v} \\ \boldsymbol{m}\boldsymbol{v} + p\mathsf{I} \\ \boldsymbol{m} \end{array}\right)^T = \boldsymbol{0}$$

where $\boldsymbol{v}$ is the velocity, p is the thermal pressure. The relativistic module is designed with 2 sets of primitive quantities, $\boldsymbol{V}_v$ and $\boldsymbol{V}_U$. The first set, $\boldsymbol{V}_v$, includes the rest-mass density ρ, three-velocity $\boldsymbol{v} = (v_{x1}, v_{x2}, v_{x3})$ and pressure p, whereas the second one replaces the ordinary velocity with the four-velocity:

$$\boldsymbol{V}_v = \Big(\rho,\; v_{x1},\; v_{x2},\; v_{x3},\; p\Big)^T, \qquad \boldsymbol{V}_U = \Big(\rho,\; u_{x1},\; u_{x2},\; u_{x3},\; p\Big)^T$$

The Lorentz factor γ is readily computed as:

$$\gamma = \frac{1}{\sqrt{1-\boldsymbol{v}^2}} = \sqrt{1+\boldsymbol{u}^2}$$

Using the four-velocity in place of the three-velocity offers in some circumstances the advantage that the total velocity $|\boldsymbol{v}| = |\boldsymbol{u}|/\sqrt{1+\boldsymbol{u}^2}$ is always less than 1 by construction, for any $0 \le |\boldsymbol{u}| < \infty$. This is not always the case when the three-velocity is used and precautionary measures are used to ensure that $|\boldsymbol{v}| < 1$. The relation between $\boldsymbol{U}$ and $\boldsymbol{V}$ is more complicated and is expressed by

$$D = \rho\gamma\,, \qquad \boldsymbol{m} = \rho h\gamma^2\boldsymbol{v} = \rho h\gamma\boldsymbol{u}\,, \qquad E = \rho h\gamma^2 - p$$

where h is the specific enthalpy (see §3.3.2 for available equation of states).

In order to express the equations in primitive (quasi-linear) form, one assumes $\delta p = c_s^2\delta e$, where c_s is the adiabatic speed of sound:

$$\begin{array}{lcl} \displaystyle\frac{\partial\rho}{\partial t} + \boldsymbol{v}\cdot\nabla\rho - \frac{1}{c_s^2 h}\boldsymbol{v}\cdot\nabla p & = & \displaystyle\frac{1}{c_s^2 h}\frac{\partial p}{\partial t} \\ \displaystyle\frac{\partial\boldsymbol{v}}{\partial t} + \boldsymbol{v}\cdot\nabla\boldsymbol{v} + \frac{1}{\rho h\gamma^2}\nabla p & = & \displaystyle -\frac{\boldsymbol{v}}{\rho h\gamma^2}\frac{\partial p}{\partial t} + \boldsymbol{a} \\ \displaystyle\frac{\partial p}{\partial t} + \frac{1}{1-\boldsymbol{v}^2c_s^2}\Big[c_s^2\rho h\nabla\cdot\boldsymbol{v} + (1-c_s^2)\boldsymbol{v}\cdot\nabla p\Big] & = & 0\,. \end{array} \tag{3.13}$$

For more detailed expressions and the characteristic decomposition, see [26].

3.3.2 Available options

The RHD sub-menu allows to change the following switches:

- `EOS`
  select the equation of state:
  - *`IDEAL`*: ideal equation of state with constant polytropic index Γ: $\rho h = \rho + \frac{\Gamma}{\Gamma - 1}p$. The specific heat ratio may be fixed using the global variable `g_gamma`.
  - *`TAUB`*: quadratic approximation to the theoretical relativistic perfect gas EOS ($\Gamma \to 5/3$ in the low temperature limit, and $\Gamma \to 4/3$ in the high temperature limit, see [26]):
  $$(h - \frac{p}{\rho})(h - 4\frac{p}{\rho}) = 1\,.$$
- `USE_FOUR_VELOCITY (`*`YES/NO`*`)`
  Use the four-velocity $\boldsymbol{u} = \gamma\boldsymbol{v}$ instead of the three velocity in the primitive set of variables. Notice that initial and boundary conditions must be given using the four velocity $\boldsymbol{u}$ and NOT $\boldsymbol{v}$.
- `ENTROPY_SWITCH (`*`YES/NO`*`)`
  Replace the total energy equation with the entropy equation away from shocks. This feature is implemented in the same way as for the HD and MHD modules, see §3.1.2.

3.4 The RMHD Module

The RMHD module implements the equations of special relativistic magnetohydrodynamics in 1, 2 or 3 dimensions. Velocities are always assumed to be expressed in units of the speed of light. Source and definition files are located inside the Src/RMHD directory tree.

3.4.1 Equations

The RMHD module solves the following system of conservation laws:

$$\frac{\partial}{\partial t}\left(\begin{array}{c} D \\ \boldsymbol{m} \\ E \\ \boldsymbol{B} \end{array}\right) + \nabla\cdot\left(\begin{array}{c} D\boldsymbol{v} \\ w_t\gamma^2\boldsymbol{v}\boldsymbol{v} - \boldsymbol{b}\boldsymbol{b} + \mathsf{I}p_t \\ \boldsymbol{m} \\ \boldsymbol{v}\boldsymbol{B} - \boldsymbol{B}\boldsymbol{v} \end{array}\right)^T = \boldsymbol{0} \tag{3.14}$$

where D is the laboratory density, $\boldsymbol{m}$ is the momentum density, E is the total energy (including contribution from the rest mass):

$$\begin{array}{rl} D = & \gamma\rho \\ \boldsymbol{m} = & w_t\gamma^2\boldsymbol{v} - b^0\boldsymbol{b} \\ E = & w_t\gamma^2 - b^0b^0 - p_t \end{array} \quad , \quad \left\{\begin{array}{rl} b^0 = & \gamma\boldsymbol{v}\cdot\boldsymbol{B} \\ \boldsymbol{b} = & \boldsymbol{B}/\gamma + \gamma(\boldsymbol{v}\cdot\boldsymbol{B})\boldsymbol{v} \\ w_t = & \rho h + \boldsymbol{B}^2/\gamma^2 + (\boldsymbol{v}\cdot\boldsymbol{B})^2 \\ p_t = & p + \dfrac{\boldsymbol{B}^2/\gamma^2 + (\boldsymbol{v}\cdot\boldsymbol{B})^2}{2} \end{array}\right.$$

Notice that the components of the momentum tensor may also be written as:

$$M^{ij} = w_t u^i u^j - b^i b^j = m^i v^j - \frac{b^i B^j}{\gamma} = m^i v^j - \left(\frac{B^i}{\gamma^2} + v^i\boldsymbol{v}\cdot\boldsymbol{B}\right)B^j$$

The RMHD module is designed with one set of primitive quantities, $\boldsymbol{V} = (\rho, \boldsymbol{v}, p, \boldsymbol{B})$. The quasi-linear form of the RMHD is not available yet and algorithms using the characteristic decomposition of the equations or the quasi-linear form are not available. Therefore, the ***CHARACTERISTIC_TRACING*** step cannot be used and the ***HANCOCK*** scheme works by default using the conservative predictor step rather than the primitive one. On the other hand, Runge-Kutta type integrators works well for the RMHD module.

The available equations of state are the constant Γ-law and the ***TAUB*** EoS already introduced for the RHD module (see [31] for the extension of this EoS to the RMHD equations).

3.4.2 Available Options

The RMHD sub-menu offers some of the switches already discussed in the MHD module (§3.2.2) or in the RHD (§3.3.2) module. Divergence control is achieved using the same algorithms introduced for MHD, namely: 8-wave (§3.2.4.1), divergence cleaning (§3.2.4.2) and the constrained transport (§3.2.4.3).

4. Non Ideal Effects

This chapter shows how non-ideal effects can be included in **PLUTO** . These include

- Viscosity (HD, MHD), described in §4.1;
- Resistivity (MHD), described in §4.2;
- Thermal conduction (HD, MHD), described in §4.3.
- Optically thin radiative cooling §4.5.

Each modules can be individually turned on in the physics sub-menus accessible via the Python script.

Numerical integration of diffusion processes (viscosity, resistivity and thermal conduction) requires the solution of mixed hyperbolic/parabolic partial differential equations which can be carried out using either a standard explicit time-stepping scheme or the Super-Time-Stepping (STS) technique, see §4.4. Depending on the time step restriction, you may include diffusion processes by setting the corresponding sub-menu voice(s) to *EXPLICIT* or to *SUPER_TIME_STEPPING*, respectively.

4.1 Viscosity

The viscous stresses enter the HD and MHD equations with two parabolic diffusion terms in the momentum and the energy conservation. Adding the viscous stress tensor to the original conservation law, Eq. (1.1), we obtain a mixed hyperbolic/parabolic system which, in compact form, may be expressed by the following:

$$\frac{\partial \boldsymbol{U}}{\partial t} + \nabla \cdot \mathsf{T} = \nabla \cdot \Pi + \boldsymbol{S} \tag{4.1}$$

where Π represents the viscous stress tensor, whose components are given by

$$(\Pi)_{ij} = 2\frac{\eta}{h_i h_j}\left(\frac{v_{i;j} + v_{j;j}}{2}\right) + \left(\eta_b - \frac{2}{3}\eta\right)\nabla\cdot\boldsymbol{v}\delta_{ij}\,. \tag{4.2}$$

Coefficients η and η_b are the first (shear) and second (bulk) parameter of viscosity respectively, $v_{i;j}$ and $v_{j;i}$ denote the covariant derivatives whereas h_i, h_j are the geometrical elements of the respective direction. The expression above holds for an isotropic viscous stress and the resulting tensor is symmetric, with $(\Pi)_{ij} = (\Pi)_{ji}$.

The actual diffusion terms will then be given by $\nabla\cdot\Pi$ and $\nabla\cdot(\boldsymbol{v}\cdot\Pi)$ at the right hand side of the momentum and the energy equation respectively. This form allows us to include a parabolic update in the actual momentum and energy fluxes (if *EXPLICIT* is chosen) but the geometrical source terms deriving from the tensor's divergence are added to the right hand side of the equations. On the other hand, if STS is chosen, advection and diffusion terms are separated via operator splitting and viscous contribution is included as a source term and computed with STS. The formalism used is the one described in the developer's guide for symmetric tensors.

The implementation of the previous expressions together with the equation module can be found under the directory Src/Viscosity. Derivative terms are discretized at cell interfaces using second-order accurate finite differences and assuming a uniform grid spacing.

Note that when using FARGO-MHD, this module can operate only with STS.

4.1.1 Viscous Coefficients

The viscous transport coefficients η (shear, **eta1_visc()**) and η_b (bulk, **eta2_visc()**) are defined in the file eta_visc.c which we suggest that the user should copy from the PLUTO/Src/Viscosity/ folder to the actual working directory and then modify it. Even though the behaviour of these coefficients is arbitrary, according to the user's needs, for mono-atomic gases Molecular Theory gives $\eta_b = 0$. The coefficient of shear viscosity η, on the other hand, is usually specified with a power law behaviour with respect to the temperature (e.g. the Sutherland formula). For more information on the analytical and numerical treatment of viscosity see [18] and [50]. It should be noted, nonetheless, that both transport coefficients must have dimensions of $\rho \times \mathrm{length}^2/\mathrm{time}$, for the correct control of the timestep, according to the stability condition discussed at the beginning of this chapter.

4.2 Resistivity

Magnetic field diffusion effects are modeled by introducing the resistivity tensor η so that the electric field becomes $\boldsymbol{\Omega} = -\boldsymbol{v} \times \boldsymbol{B} + \eta \cdot \boldsymbol{J}$, where $\boldsymbol{J} \equiv \nabla \times \boldsymbol{B}$ is the current density. The induction and energy equations gain extra terms on the right hand sides:

$$\begin{aligned} \frac{\partial \boldsymbol{B}}{\partial t} + \nabla \times (-\boldsymbol{v} \times \boldsymbol{B}) &= -\nabla \times (\eta \cdot \boldsymbol{J}) \\ \frac{\partial E}{\partial t} + \nabla \cdot [(E + p_t)\boldsymbol{v} - \boldsymbol{B}(\boldsymbol{v} \cdot \boldsymbol{B})] &= -\nabla \cdot [(\eta \cdot \boldsymbol{J}) \times \boldsymbol{B}]\ . \end{aligned} \tag{4.3}$$

Similarly, the internal energy equation modifies to

$$\frac{\partial p}{\partial t} + \boldsymbol{v} \cdot \nabla p + \rho c_s^2 \nabla \cdot \boldsymbol{v} = (\Gamma - 1)(\eta \cdot \boldsymbol{J}) \cdot \boldsymbol{J}\,. \tag{4.4}$$

The expressions on the right hand side and the corresponding equation module can be found inside Src/MHD/Resistive. The module works on uniform grids in 1, 2 and 3 dimensions in all systems of coordinates.

The *EXPLICIT* integration is compatible with both the cell centered formulations (8-wave and divergence cleaning) and the constrained transport algorithm, whereas the *SUPER_TIME_STEPPING* works only with cell-centered magnetic fields (CT is NOT supported at the moment). Copying the resistivity function PLUTO/Src/MHD/Resistive_MHD/res_eta.c (defining the components of η) into the working directory, the user can modify the file according to the particular problem. The input arguments are the grid coordinates and the face centered local values of the primitive variables. Note that the resistive module is not yet compatible with the entropy switch.

4.3 Thermal Conduction

Thermal conduction can be included for the hydro (HD) or MHD equations by introducing an additional divergence term in the energy equation:

$$\frac{\partial E}{\partial t} + \nabla \cdot [(E + p_t)\boldsymbol{v} - \boldsymbol{B}(\boldsymbol{v} \cdot \boldsymbol{B})] = \nabla \cdot \boldsymbol{F}_c\,, \tag{4.5}$$

where $\boldsymbol{F}_c$ is a flux-limited expression that smoothly varies between the classical and saturated thermal conduction regimes $\boldsymbol{F}_{\rm class}$ and $F_{\rm sat}$, respectively ([44, 37]):

$$\boldsymbol{F}_c = \frac{F_{\rm sat}}{F_{\rm sat} + |\boldsymbol{F}_{\rm class}|}\boldsymbol{F}_{\rm class}\,. \tag{4.6}$$

In the MHD case, thermal conductivity is highly anisotropic being largely suppressed in the direction transverse to the field. Denoting with $\hat{\boldsymbol{b}} = \boldsymbol{B}/|\boldsymbol{B}|$ the unit vector in the direction of magnetic field, the classical thermal conduction flux may be written as ([2]):

$$\boldsymbol{F}_{\rm class} = \kappa_\parallel \hat{\boldsymbol{b}}\left(\hat{\boldsymbol{b}} \cdot \nabla T\right) + \kappa_\perp \left[\nabla T - \hat{\boldsymbol{b}}\left(\hat{\boldsymbol{b}} \cdot \nabla T\right)\right]\,, \tag{4.7}$$

where the subscripts $\|$ and $\perp$ denote, respectively, the parallel and normal components to the magnetic field, T is the temperature, $\kappa_{\|}$ and $\kappa_{\perp}$ are the thermal conduction coefficients along and across the field. In the purely hydrodynamical limit (no magnetic field), Eq. (4.7) reduces to $\boldsymbol{F}_c = \kappa_{\|}\nabla T$.

Saturated effects ([44, 11]) are accounted for by making the flux independent of ∇T for very large temperature gradients. In this limit, the flux magnitude approaches $F_{\rm sat} = 5\phi\rho c_{\rm iso}^3$ where is the isothermal speed of sound and $\phi < 1$ is a free parameter.

The coefficients appearing in Eq. (4.7), (4.6) and in the definition of the saturated flux may be specified using the function **TC_kappa()** in (your local copy of) PLUTO/Src/Thermal_Conduction/tc_kappa.c and by noting the equivalence $\kappa_{\|} \to *\mathrm{kpar}$, $\kappa_{\perp} \to *\mathrm{knor}$ and $\phi \to *\mathrm{phi}$. The variable $*\mathrm{knor}$ can be ignored in the HD case, where $\kappa = \kappa_{\|}$. Proper setting of units and dimensions is briefly discussed in §4.3.1.

The thermal conduction module is implemented inside Src/Thermal_Conduction and works in 1, 2 and 3 dimensions in all systems of coordinates (note that it is not yet compatible with the entropy switch). Derivative terms are discretized at cell interfaces using second-order accurate finite differences and assuming a uniform grid spacing. Integration may proceed via standard explicit time stepping or Super-Time-Stepping, see §4.4.

Note: Thermal conduction behave like a purely parabolic (diffusion) operator in the classical limit ($\phi \to \infty$) and like a hyperbolic operator in the saturated limit ($|\nabla T| \to \infty$). Thus in the general case a mixed treatment is required, where the parabolic term is discretized using standard central differences and the saturated term follows an upwind rule, [4, 29].
In this case and when Super-Time-Stepping integration is used to evolve the equations, several numerical tests have shown that problem involving strong discontinuities may require a reduction of the parabolic Courant number C_p (see §4.4) and a more tight coupling between the hydrodynamical and thermal conduction scale. The latter condition may be accomplished by lowering the **rmax_par** parameter (§2.3) which controls the ratio between the current time step and the diffusion time scale, see also §4.4. An example problem can be found in Test_Problems/MHD/Thermal_conduction/Blast.

4.3.1 Dimensions

Equations (4.5)-(4.7) are solved in dimensionless form by expressing energy and time in units of $\rho_0 v_0^2$ and L_0/v_0 (respectively) and by writing temperature as $T = p/\rho \times KELVIN \times \mu$, where p and ρ are in code units and μ is the mean molecular weight. Here ρ_0, v_0, L_0 and $KELVIN$ are constants (in c.g.s units) giving the units of density, velocity, length and the temperature conversion factor, see §4.5.1 and Eq. (4.12)-(4.13). The thermal conduction coefficients must be properly defined by re-absorbing the correct normalization constants in the **TC_kappa()** function as follows

$$\kappa \to \kappa_{cgs}\frac{\mu m_u}{\rho_0 v_0 L_0 k_B} \tag{4.8}$$

where, for instance, one may use $\kappa_{cgs,\|} = 5.6\cdot 10^{-7}T^{5/2}$ and $\kappa_{cgs,\perp} = 3.3\cdot 10^{-6}n_H^2/(\sqrt{T}\boldsymbol{B}_{\rm cgs}^2)$, both in units of $\mathrm{erg\,s^{-1}\,K^{-1}\,cm^{-1}}$, while $\boldsymbol{B}_{\rm cgs}^2 = 4\pi\rho_0 v_0^2\boldsymbol{B}^2$. An example of such dimensionalization can be found in Test_Problems/MHD/Thermal_Conduction/Blast.

4.4 Numerical Integration of Diffusion Terms

4.4.1 Explicit Time Stepping

With the explicit time integration, parabolic contributions are added to the upwind hyperbolic fluxes at the same time in an unsplit fashion:

$$\boldsymbol{F} \to \boldsymbol{F}_{\rm hyp} + \boldsymbol{F}_{\rm par} \tag{4.9}$$

where "hyp" and "par" are, respectively, the hyperbolic and parabolic fluxes (see also §3.1 of [29]).

Such methods are, however, subject to a rather restrictive stability condition since, in the diffusion-dominated limit, $\Delta t \sim \Delta l^2/\eta$ where η is the maximum diffusion coefficient, see Table 2.1 for the exact limiting factor.

Clearly, high resolution and large diffusion coefficients may lead to drastic reduction of the time step thus making the computation almost impracticable.

4.4.2 Super-Time-Stepping (STS)

STS, [1], is a technique that considerably accelerates the standard explicit treatment of parabolic terms. In this case parabolic terms are treated in a separate step using operator splitting and the solution vector is evolved over a super time step, equal to the advective one. The superstep consists of N sub-steps, properly chosen for optimization and stability, depending on the diffusion coefficient, the grid size and the free parameter $\nu < 1$ (STS_nu set in definitions.h):

$$\Delta t^n = \Delta t_{\rm par}\frac{N}{2\sqrt{\nu}}\frac{(1+\sqrt{\nu})^{2N}-(1-\sqrt{\nu})^{2N}}{(1+\sqrt{\nu})^{2N}+(1-\sqrt{\nu})^{2N}}\,, \qquad \text{with} \qquad \Delta t_{\rm par} = \frac{C_p}{\displaystyle\frac{2}{N_{\rm dim}}\max_{ijk}\left(\sum_d \frac{\mathcal{D}_d}{\Delta l_d^2}\right)}\,. \tag{4.10}$$

Here $\Delta t_{\rm par}$ is the explicit parabolic time step computed in terms of the diffusion coefficient $\mathcal{D}$ and physical length Δl. The previous equation is solved to find N for given values of Δt^n, $\Delta t_{\rm par}$ and ν. For $\nu \to 0$, STS is asymptotically N times faster that the standard explicit scheme. However, very low values of ν may result in an unstable integration whereas values close to 1 can decrease STS's efficiency. By default $\nu = 0.01$, a value which in many cases retains stability whereas giving substantial gain, see Fig 4.1.

Stability analysis for the constant coefficient diffusion equation, [5], indicates that the value of C_p (parabolic Courant number) should be $\le 1/N_{\rm dim}$ ($N_{\rm dim}$ is the number of spatial dimensions) and it may be used to adjust the size of the spectral radius for strongly nonlinear problems. A reduction of C_p will results in increased stability at the cost of more substeps N. The default value is $C_p = 0.8/N_{\rm dim}$ but it may be changed in your pluto.ini through **`CFL_par`**, see §2.3.

Since STS treats parabolic equations in an operator-split formalism, it may be advisable (for highly nonlinear problems involving strong discontinuities) to limit the scale disparity between advection and diffusion time scales by restricting the time step Δt^n to be at most $r_{\rm max}\Delta t_{\rm par}$, with $\Delta t_{\rm par}$ defined by Eq. (4.10) and $r_{\rm max}$ a free parameter, see §2.3. In this cases, $r_{\rm max}$ may be lowered by lowering **`rmax_par`** in pluto.ini from its default value (100) to 40 or even less.

Note that although this method is in many cases considerably more efficient than the explicit one, it is found to be slightly less accurate due to operator splitting. The method is by definition first order accurate in time, although different values of the ν parameter are found to affect the accuracy. On the other hand, STS bypasses the severe time constraint posed by second derivative operators in high resolution simulations.

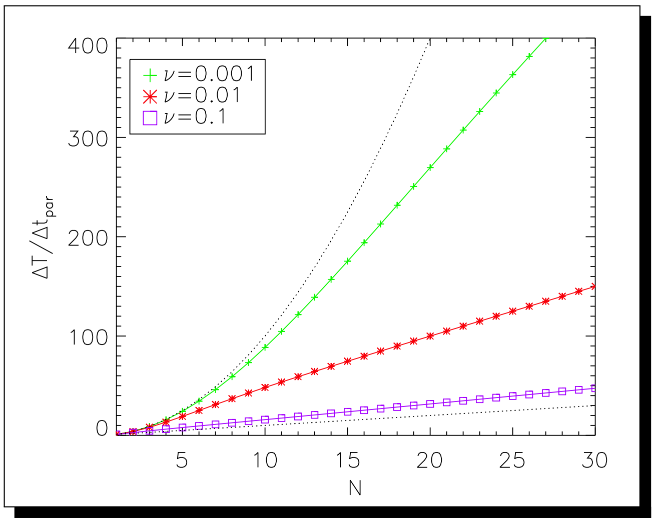


Figure 4.1: Length of a super-step (in units of the explicit one, $\Delta T/\Delta t_{\rm par}$) as function of the number of sub-steps N using different values of $\nu = 10^{-3}$ (green, plus sign), $\nu = 10^{-2}$ (red, asterisk - default), $\nu = 10^{-1}$ (purple, square). The upper dotted lines gives the $\nu \to 0$ limit ($\Delta T \propto N^2$), whereas the lower one represents the explicit limit ($\Delta T \propto N$). If $\Delta T/\Delta t_{\rm par} = 100$, for example, explicit integration would require 100 steps while super time stepping only ≈ 21 (for $\nu = 10^{-2}$) or 11 (for $\nu = 10^{-3}$) steps.

4.5 Cooling

PLUTO can include optically thin losses in a fractional step formalism, which preserves 2^{nd} order in time accuracy if both the advection and source steps are at least 2^{nd} order accurate. The user can select one among several different cooling functions by properly setting the `COOLING` flag during the python script.

Cooling modules are implemented inside the Src/Cooling directory and require a number of global variables to be properly initialized inside your **`Init()`** function. Some of these global variables are introduced in order to handle units and dimensions of the problem, i.e. they properly scale the problem to c.g.s. units (see §4.5.1). In **PLUTO** dimensionalization is achieved by introducing units for density, velocity and length; all other units are derived from combinations of these, see §4.5.1 for more detail.

Other variables are introduced to control crucial parameters such as the maximum allowed cooling rate in each time step, or the cutoff temperature:

- `g_maxCoolingRate`: limit the time step so that the maximum fractional thermal losses cannot exceed `g_maxCoolingRate`. In general $0 <$`g_maxCoolingRate`< 1; the default is 0.1.

- `g_minCoolingTemp`: set the cut-off temperature (in K) below which cooling is artificially set to 0.

Generally speaking, the user can still adopt non-dimensional arbitrary units to assign values for density, velocity, pressure, magnetic field, etc... The "dimensionalization" of the problem is confined to the cooling modules.

4.5.1 Units and Dimensions

In general, **PLUTO** works with non-dimensional, arbitrary units, so that all quantities can be properly scaled to "reasonable" numbers. Although it is possible, in principle, to work directly in c.g.s. units (i.e. cm, sec and gr), it is strongly recommended to scale all quantities to non-dimensional units, in order to avoid occurrences of extremely small ($\lesssim 10^{-9}$) or large numbers ($\gtrsim 10^{12}$) that may be misinterpreted by numerical algorithms.

For simple adiabatic simulations involving no source terms, the dimensionalization process can be avoided. Dimensionalization is strictly necessary when specific length, time or energy scales are intro-

Name	Numerical Value	Physical Meaning
CONST_PI	3.14159265358979	π
CONST_amu	$1.66053886e-24$	atomic mass unit
CONST_mp	$1.67262171e-24$	proton mass
CONST_mn	$1.67492728e-24$	neutron mass
CONST_me	$9.1093826e-28$	electron mass
CONST_mH	$1.6733e-24$	Hydrogen atom mass
CONST_kB	$1.3806505e-16$	Boltzmann constant
CONST_sigma	$5.67051e-5$	Stephan Boltmann constant
CONST_NA	$6.0221367e23$	Avogadro Contant
CONST_c	$2.99792458e10$	Speed of Light
CONST_Msun	$2.e33$	Solar Mass
CONST_Rsun	$6.96e10$	Solar Radius
CONST_Mearth	$5.9736e27$	Earth Mass
CONST_Rearth	$6.378136e8$	Earth Radius
CONST_G	$6.6726e-8$	Gravitational Constant
CONST_h	$6.62606876e-27$	Planck Constant
CONST_pc	$3.0856775807e18$	parsec
CONST_ly	$0.9461e18$	light year
CONST_au	$1.49597892e13$	astronomical unit

Table 4.1: Predefined constant in c.g.s. units available in **PLUTO** .

duced in the problem and they must compare to the dynamical advection scales. For such problems, **PLUTO** requires <u>three</u> fundamental units to be specified using the following global variables:

`g_unitLength`	(L_0)	:	sets the reference length in cm
`g_unitVelocity`	(v_0)	:	sets the reference velocity in cm/s
`g_unitDensity`	(ρ_0)	:	sets the reference density in gr/cm^3

With these choices, time is be measured in units of $t_0 = L_0/v_0$, pressure is assigned in units of $p_0 = \rho_0 v_0^2$, while magnetic field (for the `MHD` module only, see §3.2) is given in units of $B_0 = v_0\sqrt{4\pi\rho_0}$, i.e.:

$$\rho = \frac{\rho_{cgs}}{\rho_0} \quad , \quad v = \frac{v_{cgs}}{v_0} \quad , \quad p = \frac{p_{cgs}}{\rho_0 v_0^2} \quad , \quad B = \frac{B_{cgs}}{\sqrt{4\pi\rho_0 v_0^2}} \tag{4.11}$$

Note that, when the relativistic modules are used, v_0 must be the speed of light.

PLUTO has several predefined physical and astronomical constants in cgs units which may be used anywhere in the code, see Table 4.1.

In practice it may be more convenient to introduce the non-dimensional (adiabatic) sound speed $c = c_{cgs}/v_0$ and obtain the normalized pressure as

$$p = \frac{\rho c^2}{\Gamma}$$

which, for $v_0 = c_{cgs}$, reduces to

$$p = \frac{\rho}{\Gamma}$$

In some circumstances a reference temperature $T_{\rm ref}$ (in Kelvin) may be given. Direct relation between pressure and density (in "code", or non-dimensional units) and temperature (in Kelvin) is provided by

$$T = \frac{p}{\rho}\frac{\mu m_u v_0^2}{k_B} \tag{4.12}$$

where k_B (`CONST_kB`) is the Boltzmann constant, m_u (`CONST_amu`) is the atomic mass unit, μ is the mean molecular weight (which depends on the composition of the gas, see §4.5), and p and ρ have already been scaled to "code" (i.e. non-dimensional) units. The macro `KELVIN` can be used as the conversion factor between code units and temperature in Kelvin,

$$T = \mathrm{KELVIN} \times \mu \times \frac{p}{\rho} \tag{4.13}$$

It requires a call to the **`MeanMolecularWeigth()`** function to compute μ. Thus, if `v` is a one-dimensional array of primitive quantities, the typical way to recover temperature in Kelvin is

```
mu = MeanMolecularWeight(v);
T  = v[PRS]/v[RHO]*KELVIN*mu;
```

From Eq. (4.12) one can define some reference mean molecular weight $\bar{\mu}$ and set

$$v_0 = \sqrt{\frac{\Gamma k_B T_{ref}}{\bar{\mu} m_u}}$$

so that the (adiabatic) speed of sound $c = \sqrt{\Gamma p/\rho}$ is identically $= 1$.

As an example, consider a simple 1-D flow with typical number densities of the order of $n \approx 10\ cm^{-3}$, temperatures of the order of $T \approx 10^4$ K (corresponding to typical sound speeds of $c_{s0} \approx 10$ Km/s) and a magnetic field (if any) of the order of $10\ \mu G$. Suppose, also, that the flow is propagating with uniform speed $v \approx 50$ Km/s and the typical scale size of the problem is $L \approx 1\ pc \approx 3.1 \cdot 10^{18}$ cm.

- *Example 1*:
  One could, for example, define unit density, velocity and length (respectively) as

  $$\rho_0 = n_0 m_p \approx 1.67 \cdot 10^{-23}\ gr/cm^3\,, \quad v_0 = 1\ Km/s = 10^5\ cm/s\,, \quad L_0 = 3.1 \cdot 10^{18}\ cm$$

  With this choice of units, the piece of code describing the initial condition becomes

```
   g_unitDensity  = 1.67e-23; /* reference density (\rho_0) in units of gr/cm^  3 */
   g_unitVelocity = 1.e5;     /* reference velocity (v_0) in units of cm/sec  */
   g_unitLength   = 3.1e18;   /* reference length (L_0) in cm  */

   us[RHO]  = 1.0;     /* means  1 * [1.67 10^{-23}]  gr/cm^3 or 10/cm^3  */
   us[VX1]  = 50.0;    /* means 50 * [1 Km/sec]   */
   c_sound  = 10.0;    /* means 10 * [1 Km/sec]   */

/* -- the following definition of pressure
      gives a sound speed of 1 * 1.e6 = 10 Km/sec  --  */

 us[PRS] = us[RHO]*c_sound*c_sound/gmm;  /* means 100/gmm * [\rho_0*v_0^2] */

/* -- Assign a magnetic field of 10^{-5} Gauss  --  */

 us[BX1] = 1.e-5/sqrt(4.0*CONST_PI*g_unitDensity)/g_unitVelocity;
```

  With this initialization, the sound speed is exactly $c_s = 10$ Km/s.

- *Example 2*:
  Alternatively, one may want to specify a temperature of $T = 10^4$ K, and then use the sound speed as the reference value for the velocity; in this case the previous piece of code should modified to

```
 T      = 1.e4;    /*  initial reference temperature in Kelvin  */
 mu_ave = 1.27;    /*  define some typical molecular weight  */

 /* -- obtain sound speed and set it to
       reference velocity (v_0) in units of cm/sec  --  */

 g_unitVelocity = sqrt(gmm*CONST_kB*T/(mu_ave*CONST_mp));

 g_unitDensity  = 1.67e-23; /*  reference density (\rho_0) in units of gr/cm^3 */
```

```
  g_unitLength   = 3.1e18;   /*  reference length (L_0) in cm  */

  us[RHO] = 1.0;     /* means  1 * 1.67 10^{-23}  gr/cm^3 or 10/cm^3  */
  us[VX1] = 50.0;    /* means 50 * 10 Km/sec   */

 /* -- Assign pressure so that T = 10^4 Kelvin --  */

  us[PRS] = us[RHO]/gmm;
  us[BX1] = 1.e-5/sqrt(4.0*CONST_PI*g_unitDensity)/g_unitVelocity;
```

Note that time is given in units of sound crossing time, L_0/v_0.

The definitions of the molecular weight function $\mu(n_k)$ is necessary when thermal losses are included (§4.5), and it has the form:

```
double MeanMolecularWeight(double *V)
{
  return 0.5; /* for a completely ionized gas   */
}
```

Finally, we notice that it is customary, sometimes, to assign magnetic field values in terms of the plasma $\beta = 2p/B^2$. Since β is already a dimensionless parameter, one should not worry about proper dimensionalization, and the line defining the magnetic field must be replaced by

```
beta   = 4.0;  /*  this is my plasma beta = 2p/B^2  */
us[BX1] = sqrt(2.0*us[PRS]/beta); /* in units of v_0\sqrt{4\pi\rho_0}  */
```

4.5.2 Power Law Cooling

Power law cooling is the most simple form of cooling, where the energy equation becomes:

$$\frac{\partial(\rho\epsilon)}{\partial t} + \Big[\cdots\Big]_{adv} = S_{\rho\epsilon} \tag{4.14}$$

The $\Big[\cdots\Big]_{adv}$ part contains advection terms (separately treated in the advection step), and $S_{\rho\epsilon}$ is a source term of the form:

$$S_{\rho\epsilon} = -a_r\rho^2 T^\alpha$$

There are no new species when this form of cooling is selected. When an ideal equation of state is used, the source step becomes

$$\frac{dp}{dt} = -(\Gamma-1)a_r\rho^{2-\alpha}p^\alpha\Big(\text{KELVIN}\times\mu\Big)^\alpha$$

and since density is not affected during this step, integration is done analytically:

$$p^{n+1} = \left\{\begin{array}{ll} \Big[(p^n)^{1-\alpha} - \Delta t C(1-\alpha)\Big]^{\frac{1}{1-\alpha}} & \text{for} \quad \alpha \neq 1 \\ p^n\exp\left(-C\Delta t\right) & \text{for} \quad \alpha = 1 \end{array}\right.$$

where $C = (\Gamma-1)a_r\rho^{2-\alpha}(\text{KELVIN}\times\mu)^\alpha$ is a constant.

The default power law accounts for bremsstrahlung cooling by solving

$$\frac{dp_{\rm cgs}}{dt_{\rm cgs}} = -(\Gamma-1)\frac{a_{\rm br}}{\mu^2 m_H^2}\rho^2_{\rm cgs}\sqrt{T(K)} \qquad \Longrightarrow \qquad \frac{dp}{dt} = -C\rho^2\sqrt{\frac{p}{\rho}}$$

with p, t and ρ given in code units and

$$C = a_{\rm br}\frac{\Gamma-1}{(k_B\mu m_H)^{3/2}}\frac{\rho_0 L_0}{v_0^2}$$

where ρ_0, v_0 and L_0 are the reference density, velocity and length defined in §4.5.1 and $a_{\rm br} = 2\cdot 10^{-27}$ in expressed in c.g.s. units. The implementation of this cooling step, with $\alpha = 1/2$, can be found under Src/Source_Terms/Power_Law/cooling.c.

4.5.3 Tabulated Cooling

The tabulated cooling module provides a way to solve the internal energy equation

$$\frac{dp}{dt} = -(\Gamma - 1)n^2\Lambda(T) \;, \quad \text{with} \quad n = \frac{\rho}{m_p + m_e} \tag{4.15}$$

when the cooling/heating function $\Lambda(T)$ is not known analytically but rather is available as a table sampled at discrete (not necessarily equidistant) points, i.e., $\Lambda_j \equiv \Lambda(T_j)$. In order to use this module, the user must provide a two-column ascii files in the working directory named cooltable.dat of the form

```
  .          .
  .          .
  .          .
T(j)    Lambda(j)
  .          .
  .          .
  .          .
```

with the temperature expressed in Kelvin and the cooling/heating function Λ in `ergs`$\cdot$`cm`^{3}`/s`. An example of such file[1] can be found in Src/Cooling/Tab/cooltable.dat. As usual, the dimensionalization is done automatically by the cooling module, once `g_unitDensity`, `g_unitLength` and `g_unitVelocity` have been defined in **`Init()`**.

Alternatively, the *tabulated* cooling module can be used to provide a user-defined cooling function,

$$\frac{dp}{dt} = -\Lambda \,, \tag{4.16}$$

where Λ can be an analytic function of the primitive variables. The explicit dependence of Λ can be given by i) copying Src/Cooling/Tab/radiat.c into your local working directory and ii) make the appropriate changes.

4.5.4 Simplified Non-Equilibrium Cooling: SNEq

This module is implemented in the Src/Cooling/SNEq directory and introduces a new variable, with index `FNEUT` used to label the fraction of neutrals f_n:

$$f_n = \frac{n_{H_I}}{n_H} \;.$$

You can assign the fraction of neutrals by setting, in the usual fashion

```
us[FNEUT] = 0.2;   /*  for example  */
```

in your **`Init()`** function. The fraction of neutrals obeys the following non-homogeneous advection equation:

$$\frac{\partial f_n}{\partial t} + \boldsymbol{v}\cdot\nabla f_n = n_e\Big[-(c_r + c_i)\, f_n + c_r\Big] \tag{4.17}$$

together with the energy equation

$$\frac{\partial(\rho\epsilon)}{\partial t} + \Big[\cdots\Big]_{adv} = n_e n_H \left(\sum_{k=1}^{k=16} j_k + w_{i/r}\right) \tag{4.18}$$

where $\Big[\cdots\Big]_{adv}$ contains the advection terms in the energy equation and the summation over k accounts for 16 different line emissions coming from some of the most common elements, k = Ly α, H α, HeI (584+623), CI (9850 + 9823), CII (156μ), CII (2325Å), NI (5200 Å), NII (6584 + 6548 Å), OI (63μ), OI (6300 + 6363 Å), OII (3727), MgII (2800), SiII (35μ), SII (6717 + 6727), FeII (25μ), FeII (1.6μ).

[1]Generated with Cloudy 90.01 for an optically thin plasma and solar abundances, thanks to T. Plewa.

The coefficient j_k in (4.18) has dimensions of $\mathrm{erg/sec\ cm^3}$ and is computed from

$$j_k = \frac{\hbar^2\sqrt{2\pi}}{\sqrt{k_B m_e} m_e} f_k q_{12} \frac{h\nu_k}{1 + n_e(q_{21}/A_{21})}$$

where k is the index of a particular transition, $f_k = n_k/n_H$ is the abundance for that particular species. Here

$$q_{12} = \frac{8.6\cdot 10^{-6}}{\sqrt{T}}\frac{\Omega_{12}}{g_1}\exp\left(-\frac{h\nu_k}{k_B T}\right) \quad , \qquad q_{21} = \frac{8.6\cdot 10^{-6}}{\sqrt{T}}\frac{\Omega_{21}}{g_2}$$

where $\Omega_{12} = \Omega_{21}$ is the collision strength and is tabulated.

In Eq. (4.18) $w_{i/r}$ represents the thermal energy lost by ionization and recombination:

$$w_{i/r} = c_i \times 13.6 \times 1.6\cdot 10^{-12} f_n + c_r \times 0.67 \times 1.6\cdot 10^{-12}(1 - f_n)\frac{T}{11590}$$

where c_r and c_i are the hydrogen ionization and recombination rate coefficients:

$$c_r = \frac{2.6\cdot 10^{-11}}{\sqrt{T}} \quad ; \qquad c_i = \frac{1.08\cdot 10^{-8}\sqrt{T}}{(13.6)^2}\exp\left(-\frac{157890.0}{\sqrt{T}}\right) .$$

The non-homogeneous parts of Eqns (4.17) and (4.18) are solved separately from the advection step using operator splitting.

4.5.5 Multi-Ion Non-Equilibrium Cooling: MINEq

This module computes the dynamical evolution and ionization state of the plasma using the multi-ion model of [49] including with 28 ion species namely HI, HeI HeII and the first five ionization stages of C,N,O,Ne and S. For each ion, **PLUTO** introduces an additional variable – the fractional abundance of the ion with respect to the element it belongs:

$$X_{\rm ion} = \frac{n_{\rm ion}}{n_{\rm elem}} .$$

The names of the additional variables for the corresponding species are: `HI`, `HeI`, `HeII`, `CI`, `CII`, `CIII`, `CIV`, `CV`, `NI`, `NII`, `NIII`, `NIV`, `NV`, `OI`, `OII`, `OIII`, `OIV`, `OV`, `NeI`, `NeII`, `NeIII`, `NeIV`, `NeV`, `SI`, `SII`, `SIII`, `SIV`, `SV`. Ionized hydrogen is simply $1 - X_{\rm HI}$. You can assign the fraction of any ion specie by setting, in the usual fashion

```
us[HeII] = 0.2;   /*  for example  */
```

in your **`Init()`** function.

The fractions of all ion species+ can also be automatically set for equilibrium conditions using the **`CompEquil()`** function in Src/Cooling/MINEq/comp_equil.c:

```
double CompEquil (double N, double T, double *v)
```

where N and T are the plasma number density and temperature respectively and `*v` is a vector of primitive variables. The function will return the electron density as output, and `*v` will contain the computed ionization fractions (the other variables are not affected). The routine solves the system of equations for abundances in equilibrium.

Note: The number of ions for C, N, O, Ne and S may be reduced from 5 (default) to a lower number (> 1) by editing Src/Cooling/MINEq/cooling.h. This may reduce computational time if the expected temperatures are not large enough to produce high ionization stages (e.g. `IV` or `V` if $T < 10^5 K$).

The elements abundances are set in radiat.c from the Src/Cooling/MINEq/ folder. When using the MINEq module, the cooling coefficients tables are generated at the beginning of the simulation by the routines in Src/Cooling/MINEq/make_tables.c. Update or customization of the atomic data can be done by editing this file.

The ion fractions are integrated through advection equations of the form:

$$\frac{\partial X_i}{\partial t} + \boldsymbol{v} \cdot \nabla X_i = S_i \,, \tag{4.19}$$

where the source term S_i is computed taking into account collisional ionization, radiative and dielectronic recombination, as well as charge-transfer with H and He processes, see [49]. Similarly, a source term is added to the energy equation:

$$\frac{\partial(\rho\epsilon)}{\partial t} + \Big[\cdots\Big]_{adv} = -\left(\Gamma - 1\right)\Big[n_{\rm at}n_{\rm el}\Lambda\left(T, \boldsymbol{X}\right) + L_{\rm FF} + L_{\rm I-R}\Big] \,, \tag{4.20}$$

where $\Big[\cdots\Big]_{adv}$ contains the advection terms in the energy equation, $\Lambda(T, \boldsymbol{X})$ is the radiative cooling function, $L_{\rm FF}$ and $L_{\rm I-R}$ denote the energy losses in bremsstrahlung and ionization/recombination processes respectively, $n_{\rm at}$ and $n_{\rm el}$ are the total atom and electron number densities respectively. In Eq. (4.20) $\Lambda(T, \boldsymbol{X})$ is computed as the sum:

$$\Lambda(T, \boldsymbol{X}) = \sum_k X_k \mathcal{L}_k(n_{\rm el}, T) B_k \,, \tag{4.21}$$

where B_k is the fractional abundance of the element, and

$$\mathcal{L}_k = \sum_i N_i \sum_{j<i} A_{ij} h\nu_{ij} \,, \tag{4.22}$$

is the total cooling for one ion specie, that is computed and saved to external files by the tables generation program, then loaded at runtime.

The non-homogeneous parts of Eqns (4.19) and (4.20) are solved separately from the advection step through the Src/cooling_source.c. MINEq uses a dynamically switching integration algorithm for the ion species and energy designed to maximize the accuracy while keeping the computational cost as low as possible.

5. Additional Modules

5.1 The ShearingBox Module

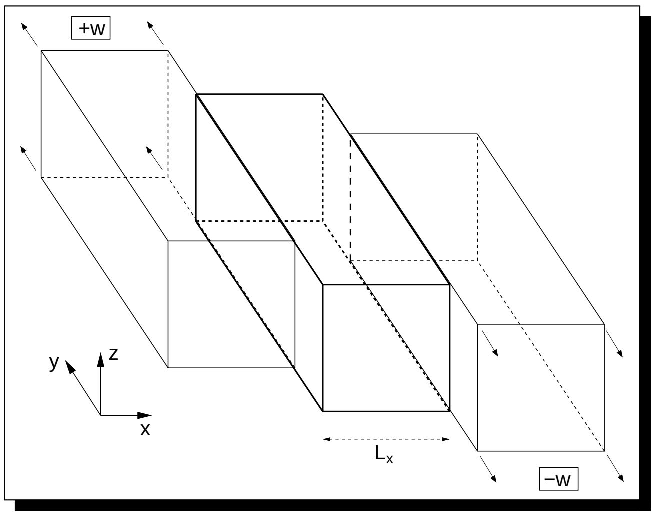


Figure 5.1: Schematic representation of the shearing boundary condition. The computational domain (central box) is assumed to be surrounded by identical boxes sliding with constant velocity $w = |2AL_x|$ with respect to one another.

The shearingbox provides a local model of a differentially rotating system obtained by expanding the tidal forces in a reference frame co-rotating with the disk at some fiducial radius R_0. The validity of the approximation (and of the module itself) is restricted to a Cartesian box (considered small with respect to the global flow) with a steady flow consisting of a linear shear velocity,

$$v_y = -q\Omega_0 x\,, \qquad \text{with} \quad q = -\left.\frac{d\log\Omega(R)}{d\log R}\right|_{R=R_0} \tag{5.1}$$

where Ω_0 is the local constant angular velocity and q is a local measure of the differential rotation ($q = 3/2$ for a Keplerian profile). The module solves the isothermal or adiabatic MHD equations in a non-inertial frame so that the momentum and energy equations become

$$\begin{aligned}\frac{\partial(\rho\boldsymbol{v})}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}\boldsymbol{v} - \boldsymbol{B}\boldsymbol{B}) + \nabla p_t &= \rho\boldsymbol{g}_s - 2\Omega_0\hat{\boldsymbol{z}}\times\rho\boldsymbol{v} \\ \frac{\partial E}{\partial t} + \nabla\cdot\left[(E+p_t)\,\boldsymbol{v} - (\boldsymbol{v}\cdot\boldsymbol{B})\,\boldsymbol{B}\right] &= \rho\boldsymbol{v}\cdot\boldsymbol{g}_s\,,\end{aligned} \tag{5.2}$$

where $\boldsymbol{g}_s = \Omega_0^2(2qx\hat{\boldsymbol{x}} - z\hat{\boldsymbol{z}})$ is the tidal expansion of the effective gravity while the second term in Eq. (5.2) represents the Coriolis force. The continuity and induction equations retain the same form as the original system.

5.1.1 Using the module

The shearingbox module is implemented in Src/MHD/ShearingBox and can be used with the *ISOTHERMAL* or *IDEAL* equations of state. You may enable it by invoking the Python setup script with the `--with-sb` option.

Initial conditions are specified, as usual, in the **Init()** function where in addition you also have to assign the value of the shear parameter q through the global variable `sb_q` and the angular rotation velocity Ω_0 using the global variable `sb_Omega`. Gravitational terms should be set in the body_force.c function following the examples given in Test_Problems/MHD/ShearingBox.

While the computational box should be periodic in the azimuthal (y) direction, radial (x) boundary conditions are determined by "image" boxes sliding with relative velocity $w = |q\Omega_0 L_x|$ relative to the computational domain, Fig 5.1. In other words, the boundary conditions at the left/right x-boundaries are

$$\left\{\begin{array}{lcl} q(x,y,z,t) & = & q\left(x \pm L_x, y \mp wt, z, t\right) \\ v_y(x,y,z,t) & = & v_y\left(x \pm L_x, y \mp wt, z, t\right) \pm w\,, \end{array}\right. \tag{5.3}$$

where q is any other flow quantities except v_y. This particular condition is provided by the module itself and should be selected by assigning *`shearingbox`* to the `X1_BEG` and `X1_END` boundaries in your pluto.ini. Boundary conditions in the vertical (z) direction may be arbitrarily prescribed.

The ShearingBox module is implemented inside Src/MHD/ShearingBox and works, at present only with *`CONSTRAINED_TRANSPORT`* MHD. Unlike previous releases of **PLUTO** , parallelization can now be performed in all three spatial dimensions.

5.2 The FARGO Module

The FARGO-MHD module permits larger time steps to be taken in those computations where a (grid-aligned) supersonic or super-fast dominant background orbital motion exists. A detailed discussion of the module may be found in [35]. The relevant source files can be found inside Src/Fargo.

The algorithm decomposes the total velocity into an average azimuthal contribution and a residual term and the MHD or HD equations are solved through a linear transport operator in the direction of orbital motion and a standard nonlinear solver applied to the MHD equations written in terms of the residual velocity. The Courant condition is then computed only from the residual velocity, leading to substantially larger time steps.

The discretization is fully conservative in both angular momentum and total energy. The module works only the Constrained Transport (CT) method to control divergence-free condition.

5.2.1 Using the Module

The FARGO-MHD module is implemented in Src/Fargo and can be enabled by invoking the python script with the `--with-fargo` option. It works in Cartesian, polar and spherical coordinates with a dimensionally-unsplit time stepping scheme (i.e. with `DIMENSIONAL_SPLITTING` set to *`NO`*). The background velocity can be computed by **PLUTO** in two different ways depending on the value of the macro `FARGO_AVERAGE_VELOCITY` set in Src/Fargo/Fargo.h:

- *`YES`* (default): The azimuthal velocity v_y or v_ϕ is averaged along the corresponding orbital direction. This operation is performed once every fixed number of time steps (set by the macro `FARGO_NSTEP_AVERAGE`, default is 10);
- *`NO`*: The velocity is prescribed analytically with the user supplied function **`FARGO_SetVelocity()`** (to be implemented in your init.c.

Boundary conditions can be assigned as usual by keeping in mind that the velocity defined in `d->Vc[]` is the *total* velocity and not the residual.

The order of reconstruction used during the linear transport step can be set by changing `FARGO_ORDER` inside Src/Fargo/fargo.h. The default value is 3 (third-order PPM) but it can be lowered to 2 (second-order MUSCL-Hancok) by editing Src/Fargo/fargo.h.

5.2.2 A Note on Parallelization

The algorithm has been fully parallelized in all coordinate directions with the requirement that the number of zones per processor in the orbital direction must be larger than the expected transport shift m.

With a large number of processors ($\gtrsim$ 2048), the resulting auto-decomposition mode may result in sub-grids that violate this condition and an error message is issued. To avoid this problem you can specify the parallel decomposition with the `-dec n1 [n2] [n3]` command line argument (§1.4.1) and ensure that not too many processors are used along the ϕ direction. As an example, suppose you wish to use 4096 processors but only 8 along the orbital direction (x_2). You may specify the domain decomposition by giving, say, 32, 8 and 16 in the three directions with

```
 mpirun -np 4096 ./pluto -dec 32 8 16
```

5.3 High-order Finite Difference Schemes

An alternative to the Finite Volume (FV) methodology presented in the previous Chapters and to the interpolation algorithms described in Chapter 2 is the use of conservative, high-order Finite Difference (FD) schemes. 3rd and 5th order accurate in space interpolation can be used in **PLUTO** , invoking setup.py with the following extension:

```
~/MyWorkDir > python $PLUTO_DIR/setup.py --with-fd
```

The available options in `INTERPOLATION` will now be

- *`LIMO3_FD`*: third-order reconstruction of [7];
- *`WENO3_FD`*: an improved version of the classical third-order WENO scheme of [16] based on new weight functions designed to improve accuracy near critical points [52];
- *`WENOZ_FD`*: improved WENO5 scheme proposed by [6];
- *`MP5_FD`* : the monotonicity preserving scheme of [46] based on a fifth-order interface value;

The use of high-order FD schemes is subject to some restrictions:

- The allowed modules are *`HD`* and *`MHD`* as the special relativistic counterparts are not yet implemented.
- In the case of the *`MHD`* module, only cell centered magnetic field collocation is supported, i.e. *`DIV_CLEANING`*.
- Temporal integration can be performed only with RK3 (split or unsplit).
- Only Cartesian coordinates are supported (in any number of dimensions).

FD schemes are based on a global Lax-Friedrichs flux splitting and the reconstruction step is performed (for robustness issues) on the local characteristic fields computed by suitable projection of the positive and negative part of the flux onto the left conservative eigenvectors. For this reason, these schemes are more CPU intensive than traditional FV schemes (approximately a factor $2 \div 3.5$) although can achieve the same accuracy with much fewer points.

Unlike the FV schemes currently present in **PLUTO** (possessing an overall 2nd order accuracy), schemes provided by the conservative FD module are genuinely third- or fifth- order accurate. The latter, in particular, have shown [34] to outperform traditional second-order TVD schemes in terms of reduced numerical dissipation and faster convergence rates for problem involving smooth flows. Figure 5.2 shows, as a qualitative example, a comparison between traditional FV methods (such as Muscl-Hancock or PPM) and some FD methods on a problem involving circularly polarized Alfven waves (see Test_Problems/MHD/CP_Alfven). Although FD schemes can correctly describe discontinuities, the advantages offered by their employment are more evident in presence of smooth flows.

5.3.1 WENO schemes

The WENO schemes are based on the essentially non-oscillatory (ENO) schemes, originally developed by [15] using a finite volume formulation and later improved by [43] into a finite difference form. Unlike TVD schemes that degenerate to first order at smooth extrema, ENO schemes maintain their accuracy successfully suppressing spurious oscillations. This is accomplished utilizing the smoothest stencil among a number of candidates to compute fluxes at the cell faces.

WENO schemes are the natural evolution of ENO schemes, where a weighted average is taken from all the stencil candidates. Weights are adjusted by local smoothness indicators. Originally developed by [39] for 1-D finite volume formulation, WENO schemes were then implemented in multi-dimensional FD by [16], optimizing the original weighing for accuracy.

Currently, the available WENO schemes in **PLUTO** are the 5th order WENOZ of [6] which improves over the original one [16] in that it is less dissipative and provide better resolution at critical points at a very modest additional computational cost. A third order WENO scheme is also provided, namely WENO+3 of [52]. More details can be found in the paper by Mignone, Tzeferacos & Bodo [34].

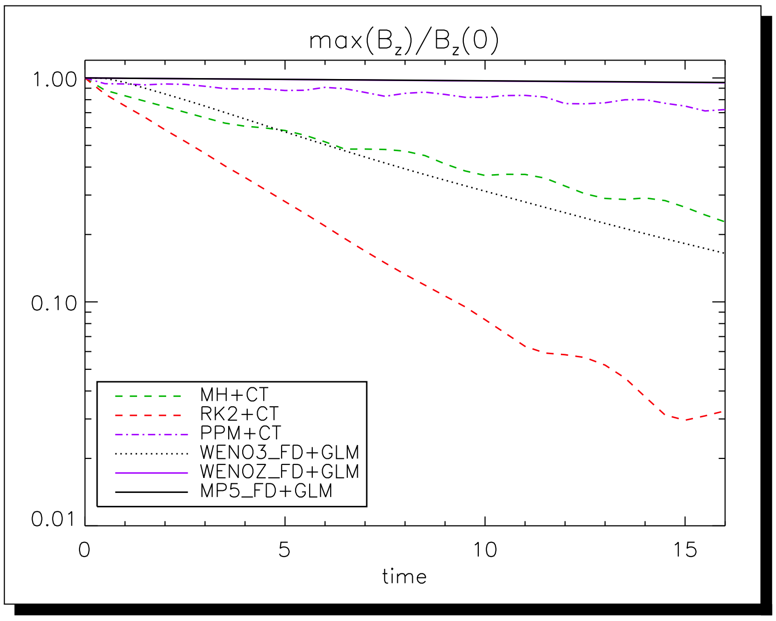


Figure 5.2: Long term (numerical) decay of a circularly polarized Alfven wave on a 2D periodic domain with [120 × 20] zones. The different curves plot the maximum value of B_z as a function of time and thus give a measure of the intrinsic numerical dissipation. Selected finite volume schemes employing constrained transport (CT) are: MUSCL-HANCOCK (MH+CT), Runge Kutta 2 (RK2+CT) and PPM+CT. Finite difference schemes employ the GLM formultation and are, respectively, given by WENO3, WENOZ and MP5.

5.3.2 LimO3 & MP5

As an alternative to the previously described WENO schemes, LimO3 and MP5 interpolations are also available. The former is a new and efficient third order limiter function, proposed by [7]. Utilizing a three point stencil to achieve piecewise-parabolic reconstruction for smooth data, LimO3 preserves its accuracy at local extrema, avoiding the well known clipping of classical second-order TVD limiters. Note that this reconstruction is also available in the finite-volume version of the code.

PLUTO 's MP5 originates from the monotonicity preserving (MP) schemes of [46], which achieve high-order interface reconstruction by first providing an accurate polynomial interpolation and then by limiting the resulting value in order to preserve monotonicity near discontinuities and accuracy in smooth regions. The MP algorithm is better sought on stencils with five or more points in order to distinguish between local extrema and a genuine O(1) discontinuities.

For an inter-scheme comparison and more information on their implementation with the MHD-GLM formulation, consult [34].

6. Output and Visualization

In this Chapter we describe the data formats supported by the static grid version of **PLUTO** and how they can be read and visualized with some of the most popular visualization packages.

6.1 Output Data Formats

With the static version of **PLUTO** , data can be dumped to disk in a variety of different formats. The majority of them is supported on serial as well as parallel systems. The available formats are classified based on their file extensions:

.dbl: double-precision (8 byte) binary data (serial/parallel);

.flt: single-precision (4 byte) binary data (serial/parallel);

.dbl.h5: double-precision (8 byte) HDF5 data (serial/parallel);

.flt.h5: single-precision (4 byte) HDF5 data (serial/parallel);

.vtk: VTK (legacy) file format using structured or rectilinear grids (serial/parallel);

.tab: tabulated multi-column ascii format (serial only);

.ppm: portable pixmap color images of 2D data slices (serial/parallel);

.png: portable network graphics (bitmap image format that employs lossless data compression) color images of 2D data slices (serial/parallel).

Output files are named as base.nnnn.ext, where base is either ”data” (when all variables are written to a single file) or the name of the corresponding variable (when each variable is written to a different file, see Table 6.1), nnnn is a four-digit zero-padded integer counting the output number and ext is the corresponding file extension listed above. There’s no distinction between serial or parallel mode.

Base name	Variable	Single record size
rho	Density	$N_1 \times N_2 \times N_3$
prs	Pressure	$N_1 \times N_2 \times N_3$
vx1	x_1 velocity	$N_1 \times N_2 \times N_3$
vx2	x_2 velocity	$N_1 \times N_2 \times N_3$
vx3	x_3 velocity	$N_1 \times N_2 \times N_3$
bx1	x_1 mag. field	$N_1 \times N_2 \times N_3$
bx2	x_2 mag. field	$N_1 \times N_2 \times N_3$
bx3	x_3 mag. field	$N_1 \times N_2 \times N_3$
bx1s	x_1 stag. mag. field	$(N_1 + 1) \times N_2 \times N_3$
bx2s	x_2 stag. mag. field	$N_1 \times (N_2 + 1) \times N_3$
bx3s	x_3 stag. mag. field	$N_1 \times N_2 \times (N_3 + 1)$
trc	first tracer	$N_1 \times N_2 \times N_3$

Table 6.1: Base prefix for multiple data set. The size is in units of 4 (for the `flt` format) or 8 (for the `dbl` format) bytes.

For each format, it is possible to dump all or just some of the variables. Additional user-defined variables may be written as well, §6.2.0.1. The default setting is described separately for each output in the next subsections and may be changed if necessary, see §6.2.1.

Each format has an independent output frequency and an associated log file (i.e. dbl.out, flt.out, vtk.out and so forth) keeping track of the dump history. Two additional files, grid.out and sysconf.out, contain grid and system-related information, respectively.

Finally we point out that restart is possible only using the .dbl or .dbl.h5 data formats.

6.1.1 Binary Output: dbl or flt data formats

Binary data can be dumped to disk at a given time step as i) one single file containing all variables (by selecting `single_file` in pluto.ini) or ii) as a set of separate individual files for each variable (`multiple_files`). We recommend the second option for large data sets. The base name is set to data for a single data file containing all of the fields, or takes the name of the corresponding variable if multiple sets are preferred, see Table 6.1.

Restart can be performed from double precision binary data files by invoking **PLUTO** with the `-restart n` command line option, where `n` is the output file number from which to restart. In this case an additional file (restart.out) will be dumped to disk.

The corresponding log file (dbl.out or flt.out) is a multi-column ascii files of the form:

```
  .     .     .     .        .          .       .     .     ...
  .     .     .     .        .          .       .     .     ...
  .     .     .     .        .          .       .     .     ...
 nout   t    dt   nstep single_file  little   var1  var2   ...
  .     .     .     .        .          .       .     .     ...
  .     .     .     .        .          .       .     .     ...
```

where `nout`, `t`, `dt` and `nstep` are, respectively, the file number, time, time step and integration step at the time of writing. The next column (`single_file/multiple_files`) tells whether a single-file or multiple-files are expected. The following one (`little/big`) gives the endianity of the architecture, whereas the remaining columns list the variable names and their order in this particular format.
Default: The default is to write ALL fields in dbl format, whereas to exclude staggered magnetic field components (if any) from the flt format.

6.1.2 HDF5 Output: dbl.h5 or flt.h5 data formats

HDF5 output format can be used in the static grid version if **PLUTO** has been succesfully compiled with the serial or parallel version of the HDF5 library, see §2.2.2. The file extension is .h5 (and *not* .hdf5 as used by **PLUTO**-Chombo data files, §7.4) and output files are compatible with the Pixie format, a HDF5 filetype that can be directly opened an visualized by different softwares, like VisIt and Paraview.

The conventions used in writing .dbl.h5 or .flt.h5 files are the same ones adopted for the .dbl and .flt data formats. However, with HDF5, all variables are written to a single file and each one comes along with a supplementary .xmf text file in XDMF format that describes the content of the corresponding HDF5 file. This is useful for visualization with VisIt or ParaView, §6.3.3.

Restart can be performed from double precision HDF5 data files by invoking **PLUTO** with the `-h5restart n` command line option (§1.4.1), where `n` is the output file number from which to restart. In this case an additional file (restart.out) will be dumped to disk.
Default: The default is to write ALL fields in .dbl.h5 format, whereas to exclude staggered magnetic field components (if any) from the .flt.h5 format.
Current Limitations: at present HDF5 format does not support writing of supplementary variables.

6.1.3 VTK Output: vtk data format

VTK (from the Visualization ToolKit format) output follows essentially the same conventions used for the .dbl or .flt outputs. Single or multiple VTK files can be written by specifying either `single_file` or `multiple_files` in your pluto.ini and data values are always written using single precision with byte order set to big endian.

The mesh topology uses a rectilinear grid format for `CARTESIAN` or `CYLINDRICAL` geometry, and a structured grid format for `POLAR` or `SPHERICAL` geometry. Scalar quantities are saved with the "scalar" attribute whereas vector fields (velocity and magnetic field) with the "vector" attribute. Furthermore, both fields and arrays are written with the `CELL_DATA` attribute and grid nodes (or vertices) are used

to store the mesh[1]. If a VTK file is written to disk, the log file vtk.out is updated in the same manner as dbl.out or flt.out.
Default: By default, all variables except staggered magnetic field components (if any) are written.

6.1.4 ASCII Output: tab Data format

The *tab* format may be used for one dimensional data or relatively small two dimensional arrays in serial mode only. We warn that this output is not supported in parallel mode. The output consists in multi-column ascii files named data.nnnn.tab of the form:

```
  .     .        .         .          .     .
  .     .        .         .          .     .
  .     .        .         .          .     .
 x(i)  y(j)   var1(i,j)  var2(i,j)   var3(i,j) ...
  .     .        .         .          .     .
  .     .        .         .          .     .
  .     .        .         .          .     .
```

where the index j changes faster and a blank records separates blocks with different i index.
Default: By default, all variables except staggered magnetic field components (if any) are written.

6.1.5 Graphic Output: ppm and png data formats

PLUTO allows to take two-dimensional slices in the x_1x_2, x_1x_3 or x_2x_3 planes and save the results as color ppm or png images. The Portable Pixmap (ppm) format is quite inefficient and redundant although easy to write on any platform since it does not require additional libraries. The Portable Network Graphics (png) is a bitmap image format that employs lossless data compression. It requires libpng to be installed on your system.

Different images are associated with different variables and can have different sets of attributes defined by the `Image` structure. An image structure has the following customizable elements:

- `slice_plane`: a label (*X12_PLANE, X13_PLANE, X23_PLANE*) setting the slicing 2D plane.
- `slice_coord`: a real number specifying the coordinate orthogonal to `slice_plane`.
- `max,min`: the maximum and minimum values to which the image is scaled to. If `max=min` autoscaling is used;
- `logscale`: an integer (0 or 1) specifying a linear or logarithmic scale;
- `colormap`: the coloramp. Available options are "red" (red map) "br" (blue-red), "bw" (black and white), "blue" (blue), "green" (green).

In 2D the default is always `slice_plane = X12_PLANE` and `slice_coord = 0`. Image attributes can be set independently for each variable in the function **`ChangeDumpVar()`** in Src/userdef_output.c, see §6.2.1.
Default: By default, only density is written.

6.1.6 The grid.out output file

The grid.out file contains information about the computational grid used during the simulation. It is an ASCII file starting with a comment-header containing the creation date, dimension and geometry of the grid:

```
# ******************************************************
# PLUTO 4.0 Grid File
# Generated on  <date>
#
# DIMENSIONS: <DIMENSIONS>
# GEOMETRY:   <GEOMETRY>
# X1: [ <x1_beg>, <x1_end>], <nx1> point(s), <ngh> ghosts
# X2: [ <x2_beg>, <x2_end>], <nx2> point(s), <ngh> ghosts
# X3: [ <x3_beg>, <x3_end>], <nx3> point(s), <ngh> ghosts
# ******************************************************
```

[1]This differs from previous versions of **PLUTO** where point-centered rather than cell-centered attributes were used.

The rest of the file is made up of 3 sections, one for each dimension, giving the (interior) number of point followed by a tabulated multi-column list containing (from left to right) the point number, left and right cell interfaces:

```
 nx1
      .                .               .
      .                .               .
      .                .               .
<point number>  <cell left edge>  <cell right edge>
      .                .               .
      .                .               .
      .                .               .
```

and similarly for the x_2 and x_3 directions.

6.2 Customizing your output

Output can be customized by editing two functions in the source file Src/userdef_output.c in the **PLUTO** distribution. We recommend to copy this file into your working directory and modify the default settings, if necessary. Changes can be made by i) introducing new additional variables and ii) altering the default attributes.

6.2.0.1 Writing Supplementary Variables

New variables can be written to disk in any of the available formats previously described. The number and names of these extra variables is set in your pluto.ini initialization file under the label "uservar". The function **`ComputeUserVar()`** (located inside Src/userdef_output.c) tells **PLUTO** how these variables are computed.

As an example, suppose we want to compute and write temperature ($T = p/\rho$) and the z component of vorticity ($\omega = \partial_x v_y - \partial_y v_x$). Then one has to set

```
uservar 2  T vortz
```

in your pluto.ini under the `[Static Grid Output]` block. This informs **PLUTO** that 2 additional variables named "T" and "vortz" have to be saved. They are computed at each output by editing the function **`ComputeUserVar()`**:

```
void ComputeUserVar (const Data *d, Grid *grid)
{
  int  i,j,k;
  double ***T, ***vortz;
  double ***p, ***rho, ***vx, ***vy;
  double *dx, *dy;

  T     = GetUserVar("T");
  vortz = GetUserVar("vortz");

  rho = d->Vc[RHO];  /* pointer shortcut to density    */
  p   = d->Vc[PRS];  /* pointer shortcut to pressure   */
  vx  = d->Vc[VX1];  /* pointer shortcut to x-velocity */
  vy  = d->Vc[VX2];  /* pointer shortcut to y-velocity */

  dx = grid[IDIR].dx; /* shortcut to dx */
  dy = grid[JDIR].dx; /* shortcut to dy */

  DOM_LOOP(k,j,i){
    T[k][j][i]     = p[k][j][i]/rho[k][j][i];
    vortz[k][j][i] =   0.5*(vy[k][j][i+1] - vy[k][j][i-1])/dx[i]
                     - 0.5*(vx[k][j+1][i] - vx[k][j-1][i])/dy[j];
  }
}
```

The `DOM_LOOP(k,j,i)` macro performs a loop on the whole computational domain (boundary excluded) in order to compute `T[k][j][i]` and `vortz[k][j][i]`. Once **PLUTO** runs, these two variables will automatically be written in all selected formats (except for the ppm and png formats), by default. In order to change the default attributes, follow the example in the next subsection.

6.2.1 Changing Attributes

Defaults attributes (which variables in which output have to be written, image attributes) can be easily changed through the function **`ChangeDumpVar()`** located in the file Src/userdef_output.c.

To include/exclude a variable from a certain output type, use **`SetDumpVar`**`()(var, type, YES/NO)`. Here "`var`" is a string containing the name of a variable listed in Table 6.1 or an additional one defined in your pluto.ini. The "`type`" argument can take any value among: *`DBL_OUTPUT`*, *`FLT_OUTPUT`*, *`VTK_OUTPUT`* *`TAB_OUTPUT`*, *`PPM_OUTPUT`*, *`PNG_OUTPUT`*. This is a sketch of how this function may be used:

```
void ChangeDumpVar ()
{
  Image *image;  /* a pointer to an image structure */

  SetDumpVar("bx1", FLT_OUTPUT, NO);
  SetDumpVar("prs", PPM_OUTPUT, YES);
  SetDumpVar("vortz", PNG_OUTPUT, YES);

  image = GET_IMAGE ("rho");
  image->slice_plane = X13_PLANE;
  image->slice_coord = 1.1;
  image->max = image->min = 0.0;
  image->logscale = 1;
  image->colormap = "red";
}
```

In this example, the variable "`bx1`" is excluded from the flt output, "`prs`" and "`vortz`" (defined in the previous example) are added to the ppm and png outputs, respectively. Furthermore, the default image attributes of "`rho`" (included by default) are changed to represent a cut (in log scale, red colormap) in the xz plane at the point coordinate $y = 1.1$ in the $y-$direction.

Note that the default for dbl should never be changed since restarting from a given file requires ALL variables being evolved in time.

6.3 Visualization

PLUTO data files can be read with a variety of commercial and open source packages. In what follows we describe how **PLUTO** data files can be read and visualized with IDL[2], Gnuplot[3], VisIt[4] and ParaView[5]. In addition, the new quick visualization tool *pyPLUTO* has been kindly provided to us by B. Vaidya and D. Stepanovs and it is described in §6.3.4.

We recall that reading of .dbl or .flt files must be complemented by grid information which is stored in a separate file (grid.out). On the other hand, VTK and HDF5 files (.xmf / .h5 , .vtk or .hdf5) are "standalones" in the sense that they embed grid information and can be opened alone.

6.3.1 Visualization with IDL

Data written with **PLUTO** using .dbl, .flt, .h5 or .hdf5 format can be easily read with IDL the PLOAD procedure (located in Tools/IDL/pload.pro) which initializes common block variables shared by other functions and procedure in the Tools/IDL/ subdirectory. We strongly recommend to add the Tools/IDL to the IDL search path.

The PLOAD procedure allows to read data from disk by storing arrays into memory or through IDL file-association for large datasets. A typical IDL session is

```
IDL> PLOAD,3
IDL> DISPLAY,alog(rho), title='Density',/vbar
IDL> DISPLAY,vx1,title='X-Velocity',nwin=1
```

The first call to PLOAD initializes all the common blocks and reads the 3rd data set from disk. The second line displays the density logarithm and the third line displays (in a new window) the x_1 component of velocity. The display.pro procedure is a general-purpose visualization routine. Consult the available documentation for more information.

6.3.1.1 The PLOAD procedure

The PLOAD function is a multi-purpose routine than can be used to read grid, time information, geometry and solution data from disk in one of the following format: .dbl, .flt, .dbl.h5, .flt.h5 and .hdf5. By default, PLOAD tries to read binary data in double precision if dbl.out is present. To select a different format, a corresponding keyword must be supplied (e.g. */FLOAT*, */H5* or */HDF5* or a combination of them). This procedure must be executed prior to any other function; it initializes the following four common blocks:

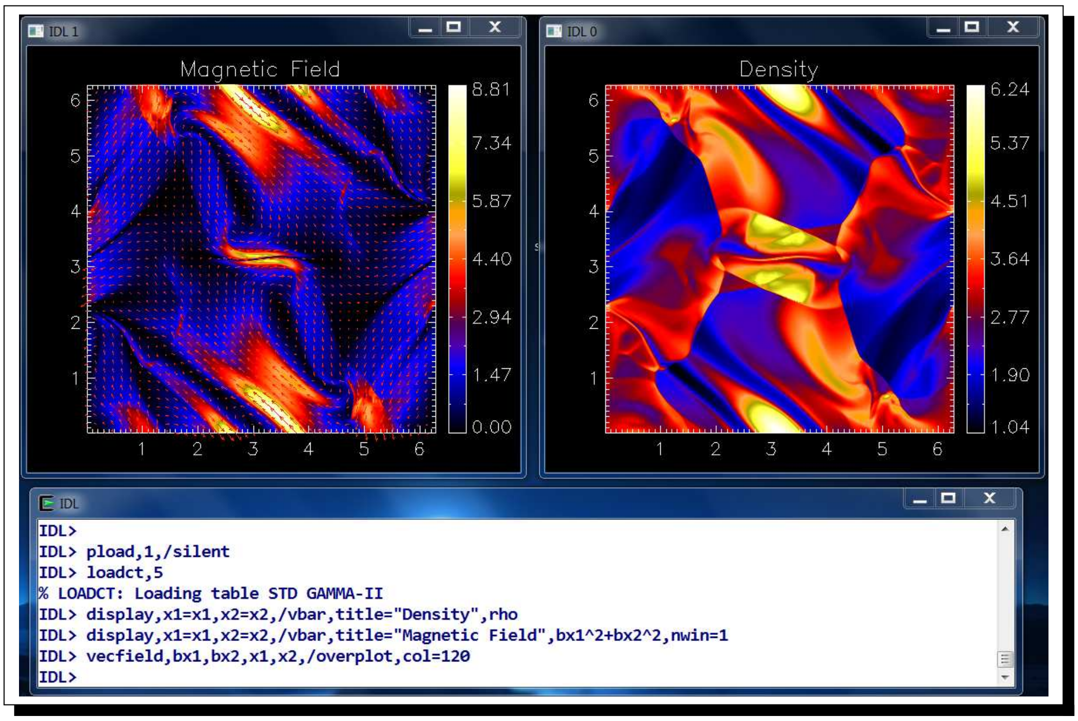
- PLUTO_GRID: contains grid information such as the number of points (nx1,nx2,nx3), coordinates (x1,x2,x3) and mesh spacing (dx1, dx2, dx3);
- PLUTO_VAR: the number (NVAR) and the names of variables being written for the chosen format. Variable names follow the same convention adopted in **PLUTO** , e.g., rho, vx1, vx2, ..., bx1, bx2, prs, .. and so on;
- PLUTO_RUN: time stepping information such as output time (t), time step (dt) and total number of files (nlast).

PLOAD can be used inside a normal IDL script, after it has been invoked at least once (or compiled with `.r pload`). A comprehensive list of all keywords can be found in Doc/idl_tools.html.

6.3.1.2 General-Purpose IDL Routines

The Tools/IDL provides several other routines for data visualization and analysis. Some of the most important are briefly described below. See Doc/idl_tools.html for more detailed information.

[2]http://www.exelisvis.com/
[3]http://www.gnuplot.info
[4]https://wci.llnl.gov/codes/visit/home.html
[5]http://www.paraview.org/



```
IDL>
IDL> pload,1,/silent
IDL> loadct,5
% LOADCT: Loading table STD GAMMA-II
IDL> display,x1=x1,x2=x2,/vbar,title="Density",rho
IDL> display,x1=x1,x2=x2,/vbar,title="Magnetic Field",bx1^2+bx2^2,nwin=1
IDL> vecfield,bx1,bx2,x1,x2,/overplot,col=120
IDL>
```

Figure 6.1: An example of visualization in IDL using the `display.pro` routine.

- COLORBAR: add a color bar to the current graphics window.
- CURL: compute the curl of a two-dimensional vector field.
- DISPLAY: produce nice graphic output for 2-D arrays on screen or eps files. It accepts several keywords, see the documentation in Doc/IDL/idl_tools.html. An example showing density and magnetic field for the Orszag-Tang MHD vortex is given in Fig (6.1).
- DIV: Compute the divergence of a two-dimensional vector field.
- FIELD_LINE: Compute field lines of a two-dimensional vector.
- GET_FRAME: Take a snapshot of the current window and produce an image file.
- HDF5LOAD: read the content of an HDF5 file.
- OPLOTBOX: overplot the AMR box layout on the current window.
- POLAR: interpolates a surface array from polar coordinates (r, θ) to Cartesian coordinates (x, y).
- REGRID: Interpolate irregularly-gridded data to a regular grid.
- SHOCKFIND: look for shocks in a two dimensional domain.
- VECFIELD: produce a 2D velocity field plot.
- WRITE_VTK: write data in vtk format.

6.3.2 Data Visualization with Gnuplot

Data can be visualized under Gnuplot using ascii (.tab) or binary data formats (version 4.2 or higher is recommended).

Ascii Data Files. If you selected the tab output in pluto.ini, you can plot 1D data from, say, your first output file, by typing

```
gnuplot> plot "data.0001.tab" u 1:3  # for density
gnuplot> plot "data.0001.tab" u 1:4  # for velocity
```

In 2-D you can take advantage of the `pm3d` style using

```
gnuplot> set pm3d map
gnuplot> splot "data.0001.tab" u 1:2:3  # for density
gnuplot> splot "data.0001.tab" u 1:2:4  # for velocity
```

Binary Data Files. Starting with Gnuplot 4.2, raw binary files are also supported. If you selected *`multiple_files`* in your dbl or flt output(s), you can display the y-velocity (for instance) using

```
gnuplot> set pm3d map
gnuplot> splot "vx2.0010.dbl" binary array=200x200 format="%double"
```

where `array=200x200` means that the underlying array structure has 200^2 points. For single datafiles (*`single_file`*), you can select the variable to display by skipping the appropriate number of bytes using the **`skip()`** function:

```
gnuplot> set pm3d map
gnuplot> nvar = 2
gnuplot> splot "data.0010.dbl" binary array=200x200 format="%lf" skip=(200*200*8*nvar)
```

In this example, we skip by 200×200 (grid size) × 8 (double precision) × 2 (since v_y is stored after ρ and v_x) bytes. Please refer to table 6.1 for grid sizes. The sequential order of variables can be deduced from the corresponding dbl.out file.

Grid information may be more easily included by taking advantage of the scripts provided with the code distribution in Tools/Gnuplot. To this end, you need to define the `GNUPLOT_LIB` environment variable (in your shell) which will be appended to the loadpath of Gnuplot:

```
> export GNUPLOT_LIB=$PLUTO_DIR/Tools/Gnuplot  # use setenv for tcsh users$
```

You may then start a Gnuplot session as follows:

```
gnuplot> load "grid.gpl"              # read and store grid information
gnuplot> load "setplmap.gpl"          # set the display canvas for pm3d plot style
```

The first line invokes the grid.gpl script which will read your grid.out and set variable values such as grid spacing (`dx,dy`), domain range (`xb,xe` and `yb, ye`) and length (`Lx=xe-xb, Ly=ye-yb`), number of points (`Nx,Ny`). Uniform grid spacing is assumed. The second script, setplmap.gpl, sets a default environment for viewing binary data files using the pm3d style of Gnuplot. It also provides two convenient macros, `@dblform` and `@fltform`, useful for passing grid and data information to `splot`. You can change the variable to be (s)plotted by assigning a different value to `nvar`, which is automatically initialized to 0 when loading setplmap.gpl. As a first example, consider

```
gnuplot> splot "rho.0002.dbl" @dblform # display (double precision) data
```

This will display density from an individual double-precision dataset. As a second example,

```
gnuplot> nvar = 4
gnuplot> splot "data.0017.flt" @fltform # display (single precision) data
```

will display the 5-th variable from the single precision datasets data.0017.flt.

To plot the portion $x \in [2.2 : 3.2]$, $y \in [0 : 1]$ of the domain using log scale, you can

```
gnuplot> set logscale cb
gnuplot> splot [2.2:3.2] [0:1] "rho.0004.flt" @fltform
gnuplot> unset colorbox # get rid of the colorbox
```

6.3.3 Data Visualization with VisIt or ParaView

PLUTO data written using VTK or HDF5 (both .h5 *and* .hdf5 files) formats can be easily visualized using either *VisIt* or *ParaView* available at https://wci.llnl.gov/codes/visit/home.html and http://www.paraview.org/, respectively. *VisIt* is an open source interactive parallel visualization and graphical analysis tool for viewing scientific data. *ParaView* is an open source mutiple-platform application for interactive, scientific visualization.

An example is shown in Fig. 6.2 for both software packages.

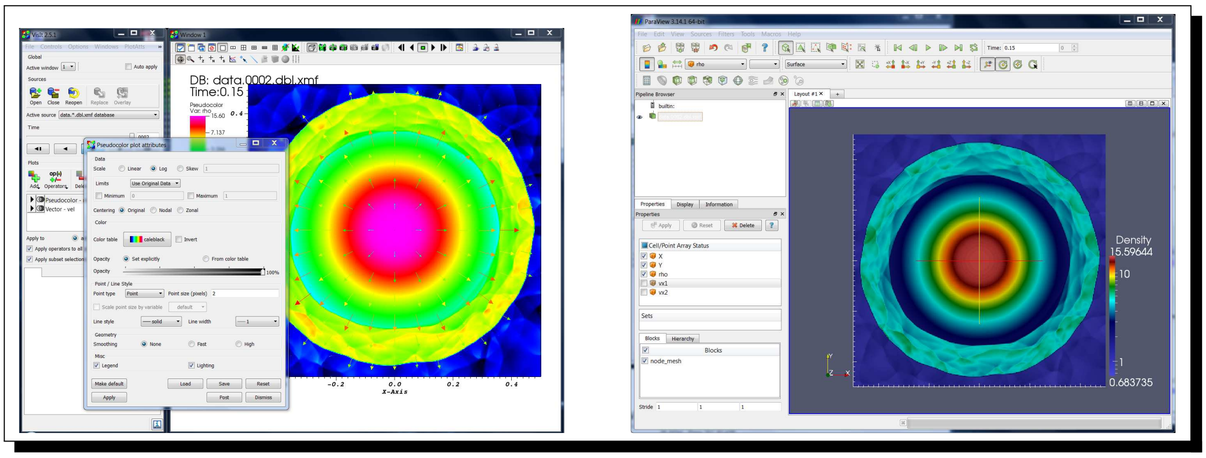
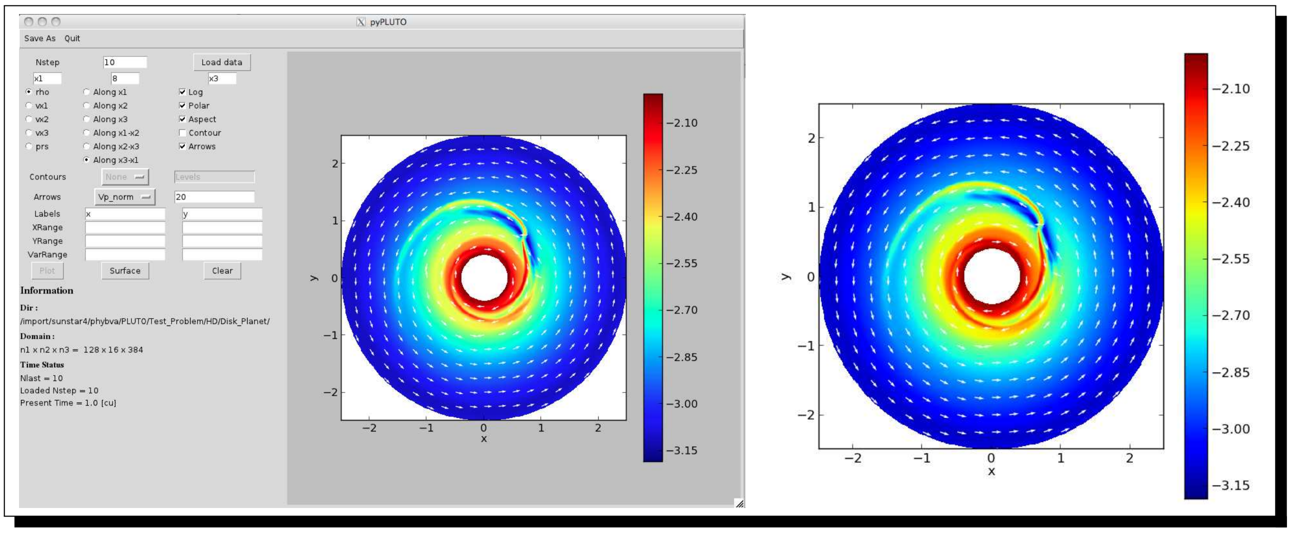


Figure 6.2: An example of visualization of an .xmf (.h5) data file using *VisIt* (left) or *ParaView* (right).

Visualization of HDF5 files. Both *VisiIt* and *Paraview* interpret the cell-centered grid and data contained in the Pixie files as node-centered: as a consequence, the first and the last half cells in every direction are clipped from the images (e.g. a small sector around $\phi = 0$ is chopped from a periodic polar plot covering the 2π angle). Therefore, for every .h5 file **PLUTO** writes also a .xmf text file in XDMF format that describes the content of the corresponding HDF5 file. The .xmf files can be directly opened by *VisIt* and *ParaView*, so as to provide the correct data centering and avoid the image clipping. Besides, we noticed that *ParaView* 3.14 (at least the precompiled binary for MAC OSX 64-bit Intel) crashes when trying to read the .h5 files, but correctly opens the .xmf files. Older versions of *ParaView* (down to 3.10) worked fine. All the variables are read as scalar quantities.

Visualization of VTK files. **PLUTO** 4.0 writes .vtk files using a cell-centered attribute rather than point-centered (as in previous versions). Although this has not been found to be a problem for *VisIt*, many filters in *ParaView* (such as streamlines) may require to apply a `Cell Data to Point Data` filter.

6.3.4 Data Visualization with pyPLUTO


Nstep 10
x1 8 x3
[✓ rho] vx1 vx2 vx3 prs
Along x1, Along x2, Along x3, Along x1-x2, Along x2-x3, [✓ Along x3-x1]
[✓ Log] [✓ Polar] [✓ Aspect] Contour [✓ Arrows]
Contours: None, Levels
Arrows: Vp_norm, 20
Labels: x, y
XRange
YRange
VarRange

Information
Dir :
/import/sunstar4/phybva/PLUTO/Test_Problem/HD/Disk_Planet/
Domain :
n1 x n2 x n3 = 128 x 16 x 384
Time Status
Nlast = 10
Loaded Nstep = 10
Present Time = 1.0 [cu]

Figure 6.3: An example of visualization with the pyPLUTO tool.

Binary data files (.dbl and .flt) can be visualized using the pyPLUTO code developed by Bhargav Vaidya[6] and Denis Stepanovs[7]. This tool is included in the current code distribution in the directory Tools/pyPLUTO/ and provides python modules (Python version > 2.7 is recommended) to load, visualize and analyse data. Additionally, for the purpose of quick check a GUI routine is provided (requires Python Tkinter).

Details of the Installation and Getting Started can be found in the pyPLUTO.pdf attached with this code in the Tools/pyPLUTO/doc/latex folder. Alternatively, the html documentation - index.html can be found in the Tools/pyPLUTO/doc/html folder.

On sucessful installation, the user can load data in the following manner:

```
> ipython --pylab
In [1]: import pyPLUTO as pp
# for loading data.0010.dbl
In [2]: D = pp.pload(10,w_dir=<path to data dir>)
# for loading data.0010.flt
In [3]: D = pp.pload(10,w_dir=<path to data dir>, datatype='float')
```

Here, D is a pload object that has all the information. For example, `D.x1` is the numpy x-array, `D.rho` - is the numpy density array, `D.vx1` - is the numpy vx1 array and so on. These numpy arrays can be easily visualised using matplotlib.

In order to use the GUI version for visualizing the data, append `$PATH` variable to the bin folder where the executable GUI_pyPLUTO.py exists after the installation of source code (see installation notes in Tools/pyPLUTO/doc/) and then apply the following commands in the data directory -

```
> GUI_pyPLUTO.py
```

or

```
> GUI_pyPLUTO.py --float
```

Along with the code, an example folder with some sample .py files are provided for certain test problems. It is required to run the test problems and obtain the data files, after which the user can run the sample .py files as follows :

1. To plot 1D density, pressure and velocity for Test_Problems/HD/Sod/

   ```
   > python sod.py
   ```

[6]School of Physics and Astronomy, University of Leeds, Leeds LS29JT. Email: B.Vaidya@leeds.ac.uk
[7]MPI Astronomy, Heidelberg

2. To plot 2D density for Test_Problems/MHD/Orszag_Tang/

   ```
   > python orszag_tang.py
   ```

3. To plot density along with magentic field lines for Test_Problems/MHD/Jet

   ```
   > python jet.py
   ```

4. To plot the density of a spherically symetric wind with velocity vectors (not normalised) for Test_Problems/HD/Stellar_Wind with $V_{\rm CSM} = 0$:

   ```
   > python stellar_wind.py
   ```

5. To do a multi plot of density at 3 different times for Test_Problems/HD/Rayliegh_Taylor/

   ```
   > python rayleigh_taylor.py
   ```

6. To plot density in the $r - \phi$ plane with velocity vectors (normalized) and to plot magnetic field in $r - \theta$ plane for Test_Problems/MHD/FARGO/Spherical_Disk.

   ```
   > python Sph_Disk.py
   ```

7. Adaptive Mesh Refinement (AMR)

PLUTO supplies adaptive mesh refinement (AMR) functionality in 1, 2 and 3 dimesions through the Chombo library available at https://commons.lbl.gov/display/chombo/. Chombo provides a distributed infrastructure for parallel calculations over block-structured, adaptively refined grids. For compatibility reasons, not all the algorithms available with the static grid version of **PLUTO** have been extended to the AMR version. **PLUTO**-Chombo can be used with all the four physics modules (i.e. HD, MHD, RHD, RMHD) under the following restrictions:

- Cartesian (1, 2 or 3 dimensions) and Cylindrical (2D) coordinates (spherical are underway).
- Only uniform grids, with equal sides (i.e. square or cubic) cells can be used;
- Magnetic fields are evolved using cell-centered formulations (Powell's eight wave or GLM); constrained transport is not yet available.
- Time stepping can be chosen between *`HANCOCK`*, *`CHARACTERISTIC_TRACING`* and *`RK2`* (new in **PLUTO** 4.0).
- I/O provided by the Hierarchical Data Format (HDF5) library[1], designed to store and organize large amounts of numerical data.

A detailed presentation of the implementation method together with an extensive numerical test suite may be found in [29].
The AMR implementation of **PLUTO** is not compatible, at present, with:

- finite difference schemes;
- the ShearingBox module (§5.1)
- the FARGO module;
- Super-Time-Stepping integration for diffusion terms.

Some of the `C` functions normally used in the static grid version of **PLUTO** have been replaced by `C++` codes, in order to interface the structure of **PLUTO** with the Chombo library. As an example, the main function main.c has been replaced by amrPluto.cpp.

7.1 Installation

In order to properly install **PLUTO**-Chombo , you will need (check also Table 1.1):

- C, C++ and Fortran compilers;
- the MPI library (for parallel runs).
- GNU make
- the HDF5 library available at http://www.hdfgroup.org/HDF5/. Chombo has been successfully compiled with hdf5-1.6.x and it can be compiled also with version 1.8.x providing the backward compatibility flag;
- the Chombo library, available under free registration at https://commons.lbl.gov/display/chombo. We strongly recommend to download Chombo version 3.1.

[1] http://www.hdfgroup.org/HDF5/

- the Chombo3.1patch.tar provided with the **PLUTO** distribution, which replaces some of the library source files.

The following sections give a quick headstart on how these libraries can be built for being used by **PLUTO** . Please consult the libraries' respective documentation for additional information.

7.1.1 Installing HDF5

HDF5 (1.6.x and 1.8.x) libraries should install without major troubles anywhere on your system. Beware that different libraries must be created for serial or parallel execution. Since in both cases library names are the same (by default), it is advisable to store them in separate locations. On a single-processor machine, serial libraries can be built, for example, using

```
> ./configure --prefix=/usr/local/HDF5-serial
> make
> make check    # optional
> make install
```

This will install the libraries under `/usr/local/HDF5-serial/lib`. If you do not have root privileges, choose a different location in your home directory (e.g. `$PLUTO_DIR/Lib/HDF5-serial`).

Note: Chombo I/O employs the HDF5 1.6.x API. However, HDF5 version 1.8.x can also be used by adding the `--with-default-api-version=v16` flag to `configure`.
Alternatively, if HDF5 1.8.x is already installed or if you wish to compile HDF5 normally, simply add the `-DH5_USE_16_API` flag to the `HDFINCFLAGS` variable inside your Make.defs.local, see §7.1.2.

On multiple-processor architectures, parallel libraries can be built by specifying the name of the `mpicc` compiler in the `CC` variable and invoking `configure` with the `--enable_parallel` switch, e.g.,

```
> CC=mpicc ./configure --prefix=/usr/local/HDF5-parallel --enable-parallel # bash user
> make
> make check  # optional
> make install
```

This will install both shared (dynamic, *.so) and static (*.a) libraries. If you build shared libraries, the environment variable LD_LIBRARY_PATH should contain the full path name to your HDF5 library (e.g. `/usr/local/HDF5-serial/lib` in the example above). Please make sure to add, for example,

```
> setenv LD_LIBRARY_PATH /usr/local/HDF5-serial/lib:$LD_LIBRARY_PATH
```

to your .tcshrc if you're using the tcsh shell or

```
> export LD_LIBRARY_PATH="/usr/local/HDF5-serial/lib":$LD_LIBRARY_PATH
```

if you're using bash. If you do not want shared libraries, then add `--disable-shared` to the `configure` command.

7.1.2 Installing and Configuring Chombo

Chombo 3.1 can be downloaded by direct access to the SVN server repository after free registration, see https://commons.lbl.gov/display/chombo/Chombo+Download+Page for instructions. The Chombo source code distribution should be (preferably) unpacked under PLUTO/Lib/ and some of the library source files must be replaced using the Chombo3.1Patch.tar patch-archive provided with the **PLUTO** distribution. A typical session is

```
> # get the 3.1 release of Chombo
> svn --username username co https://anag-repo.lbl.gov/svn/Chombo/release/3.1 Chombo-3.1
> tar xvf Chombo3.1Patch.tar -C Chombo-3.1/  # apply PLUTO-Patch
```

In order to use Chombo, you may have to build different libraries depending on the chosen compiler, serial/parallel build, number of dimensions, optimizations, etc... If you intend to run **PLUTO**-Chombo for serial or parallel computations in one, two or three dimensions in we suggest to compile all possible configurations (that is 1, 2 and 3D serial or 1, 2 / 3D parallel). Libraries are automatically named by Chombo after the chosen configuration.

The default configuration can be set by editing manually Chombo/lib/mk/Make.defs.local where, depending on your local system and configuration, you need to set make variables. To this end:

```
> cd Chombo-3.1/lib
> make setup                # create Make.defs.local from template
> cd mk/
```

The command 'make setup' will create this file from a template that contains instructions for setting make variables that Chombo uses. These variables specify the default configuration to build, what compiler to use (together with its flags), where the HDF library can be found and so on.

At this point you should edit Make.defs.local. The normal procedure is to define a default configuration, e.g., 2D serial:

```
## Configuration variables
DIM           = 2
DEBUG         = FALSE
OPT           = TRUE
PRECISION     = DOUBLE
PROFILE       = FALSE
CXX           = /usr/bin/g++
FC            = /usr/bin/g77
MPI           = FALSE
## Note: don't set the MPICXX variable if you don't have MPI installed
MPICXX        = mpic++
#OBJMODEL      =
#XTRACONFIG    =
## Optional features
#USE_64        =
#USE_COMPLEX   =
#USE_EB        =
#USE_CCSE      =
USE_HDF       = TRUE
HDFINCFLAGS   = -I/usr/local/lib/HDF5-serial/include
HDFLIBFLAGS   = -L/usr/local/lib/HDF5-serial/lib -lhdf5 -lz
## Note: don't set the HDFMPI* variables if you don't have parallel HDF installed
HDFMPIINCFLAGS= -I/usr/local/lib/HDF5-parallel/include
HDFMPILIBFLAGS= -L/usr/local/lib/HDF5-parallel/lib -lhdf5 -lz
```

Defaults are used for the remaining field beginning with a '#'.

Libraries can now be built under Chombo-3.1/lib, with

```
> make lib
```

Do not try `make all` since it won't work after the Chombo patch file has been unpacked.

Alternative configurations can be made from the default one by specifying the configuration variables explicitly on the make command line. For example:

```
> make DIM=3 MPI=TRUE lib
```

will build the parallel version of the 3D library. Additional information may be found in the Chombo/README file and by consulting the library documentation.

7.2 Configuring PLUTO-Chombo

In order to configure **PLUTO** with Chombo, you must start the Python script with the `--with-chombo` option (Python assumes that Chombo libraries has been built under PLUTO/Lib/Chombo):

```
~/work> python $PLUTO_DIR/setup.py --with-chombo
```

This will use the default library configuration (2D serial in the example above).

To use a configuration different from the default one, enter the make configuration variables employed when building the library, e.g.:

```
~/work> python $PLUTO_DIR/setup.py --with-chombo: MPI=TRUE
```

Note that the number of dimensions (`DIM`) is specified during the Python script and should NOT be given as a command line argument.

The setup proceeds normally as in the static grid case by choosing `Setup problem` from the Python script to change/configure your test problem. The makefile is then automatically created by the Python script by dumping Chombo makefile variables into the file make.vars, part of your local working directory. Although system dependencies have already been created during the Chombo compilation stage, the `Change makefile` option from the Python menu is still used to specify the name and flags of the C compiler used to compile **PLUTO** source files. This step is achieved as usual, by selecting a suitable .defs file from the Config/ directory, see §2.2. Beware that, during this step, additional variables such as `PARALLEL`, `USE_HDF5`, etc...(normally used in the static grid version) have no effect since Chombo has its own independent parallelization strategy and I/O. Fortran and C++ compilers are the same ones used to build the library.

Initial and boundary conditions are coded in the usual way and pluto.ini is still read at runtime, see next section.

7.2.1 Header File definitions.h

The header file definitions.h contains the same switches already illustrated in §2.1 and few new ones that described in the following.

7.2.2 `AMR_EN_SWITCH`

By turning this switch to *`YES`*, AMR operations such as projection, coarse-to-fine prolongation and restriction are performed on the conserved entropy rather than on the total energy density. This has the advantage of preserving entropy and pressure positivity in those situations where kinetic and/or magnetic energies are the dominant contributions to the total energy density. Using this switch, however, total energy will not be conserved at a fine/coarse interface.

7.2.3 The pluto.ini initialization file

The pluto.ini initialization file described in §2.3 still retains its functionality as in the static grid version of the code.

The *[Grid]* block is used to specify the base grid, corresponding to level 0. Only a single uniform patch per dimension should be specified. Also, the domain size and number of zones for each direction must be such that equal-sized zones are created (i.e. squares in 2D and cubes in 3D).

The *[Chombo Refinement]* and *[Chombo HDF5 output]* blocks can be used to control the refinement criteria and HDF5 output, respectively. The *[Static Grid output]* block is completely ignored with **PLUTO**-Chombo .

The two new blocks take the form:

```
...

[Chombo Refinement]

Levels           4
Ref_ratio        2 2 2 2 2
Regrid_interval  2 2 2 2
Refine_thresh    0.3
Tag_buffer_size  3
Block_factor     8
Max_grid_size    64
Fill_ratio       0.8

...

[Chombo HDF5 output]

Checkpoint_interval  -1.0  0
Plot_interval         1.0  0
```

7.2.3.1 The *[Chombo Refinement]* Block

This block sets all the relevant parameters for refinement:

- **`Levels`** (*integer*)
  set the finest allowable refinement level, starting from the base grid (level 0) defined by the *[Grid]* block. 0 means there will be no refinement.

- **`Ref_ratio`** (*integer*) (*integer*) (...)
  set the refinement ratios between all levels. First number is ratio between levels 0 and 1, second is between levels 1 and 2, etc. There must be at least **`Levels`**+1 elements or an error will result.

- **`Regrid_interval`** (*integer*) (*integer*) (...)
  set the number of timesteps to compute between regridding. A negative value means there will be no regridding. There must be at least **`Levels`** elements or an error will result.

- **`Refine_thresh`** (*double*)
  set the threshold value of the functional χ_r (see §7.2.4) above which cells are tagged for refinement during the grid generation process. When $\chi_r >$ **`Refine_thresh`**, the cell is tagged for refinement to a finer level.

- **`Tag_buffer_size`** (*integer*)
  set the amount by which to grow tags (as a safety factor) before passing to MeshRefine.

- **`Block_factor`** (*integer*)
  set the number of times that grids will be coarsenable by a factor of 2. A higher number produces "blockier" grids.

- **`Max_grid_size`** (*integer*)
  set the largest allowable size of a grid in any direction. Any boxes larger than that will be split up to satisfy this constraint.

- **`Fill_ratio`** (*double*)
  a real number between 0 and 1 used to set the efficiency of the grid generation process. Lower number means that more extra cells which are not tagged for refinement wind up being refined along with tagged cells. The tradeoff is that higher fill ratios lead to more complicated grids, and the extra coarse-fine interface work may outweigh the savings due to the reduced number of fine-level cells.

7.2.3.2 The *[Chombo HDF5 output]* Block

Similarly to the *[Static Grid Output]*, this block controls how often restart and plot files are dumped to disk:

- **`Checkpoint_interval`** (*double*) (*integer*)
  set the output frequency in time (*double*) and/or in number of timesteps (*integer*) between writing checkpoint (restart) files. Negative number means that checkpoint files are never written, 0 means that checkpoint files are written before the initial timestep and after the final one.

- **`Plot_interval`** (*double*) (*integer*)
  set the output frequency in time (*double*) and/or number of timesteps (*integer*) between writing plotfiles. Negative number means that plotfiles are never written, 0 means that plotfiles are written before the initial timestep and after the final one.

Output files are stored using the HDF5 file format and numbered as data.nnnn.hdf5 where n is a zero-padded, sequentially increasing integer (as for the static grid output, §6.1). Data files contain primitive variables where checkpoint files contain conservative variables.

7.2.4 Controlling Refinement

Zones are tagged for refinement whenever a prescribed function $\chi(\boldsymbol{U})$ of the conserved variables and of its derivatives exceeds the threshold value assigned to **`Refine_thresh`** in your pluto.ini. Generally speaking, the refinement criterion may be problem-dependent thus requiring the user to provide an appropriate definition of $\chi(\boldsymbol{U})$.

A standard criterion based on the second derivative error norm [21] is implemented in the function **`computeRefGradient()`** in the source file Src/Chombo/TagCells.cpp. The test function adopted for this purpose is

$$\chi(\boldsymbol{U}) = \sqrt{\frac{\sum_d |\Delta_{d,+\frac{1}{2}}U - \Delta_{d,-\frac{1}{2}}U|^2}{\sum_d \left(|\Delta_{d,+\frac{1}{2}}U| + |\Delta_{d,-\frac{1}{2}}U| + \epsilon U_{d,\mathrm{ref}}\right)^2}} \tag{7.1}$$

where $U \in \boldsymbol{U}$ is a conserved variables (total energy density is used by default), $\Delta_{d,\pm\frac{1}{2}}U$ are the undivided forward and backward differences in the direction d, e.g., $\Delta_{x,\pm\frac{1}{2}}U = \pm(U_{i\pm1} - U_i)$ (see also section 4.1 in [29]). The last term appearing in the denominator, $U_{d,\mathrm{ref}}$, prevents regions of small ripples from being refined and it is defined by

$$U_{x,\mathrm{ref}} = |U_{i+1}| + 2|U_i| + |U_{i-1}| \tag{7.2}$$

with $\epsilon = 0.01$. Similar expressions hold when $d = y$ or $d = z$.

7.3 Running PLUTO-Chombo

Once **PLUTO**-Chombo has been compiled and the executable pluto has been created, **PLUTO** runs in the same way, i.e.

```
~/MyWorkDir> ./pluto [flags]
```

where the supported command line options are given in Table 1.3 in §1.4. Note that `-restart` *must* be followed by the restart (checkpoint) file number. An error will occur otherwise.

Parallel runs proceeds in the usual way, e.g.,

```
~/MyWorkDir> mpirun -np 8 ./pluto [flags]
```

Note that when running in parallel, each processor redirects the output on a separate file pout.n (instead of pluto.log) where n=0...Np-1 and Np is the total number of processors. However, pout.0 also contains additional information regarding the chosen configuration.

7.4 Reading and Visualizing HDF5 Files

HDF5 is a data model, library, and file format for storing and managing large amounts of data. It supports an unlimited variety of datatypes and is designed for flexible and efficient I/O.

HDF5 data can be visualized by a number of commercial or open source packages. At present, we have successfully opened and displayed Chombo data files with IDL[2], VisIt[3] and ParaView[4]. A comprehensive list of application software using HDF5 may be found at at http://www.hdfgroup.org/tools5app.html. A set of utilities for manipulating, visualizing and converting HDF5 data files is provided by H5utils, a set of utilities available at http://www.hdfgroup.org/products/hdf5_tools/. H5utils offers a simple tool for batch visualization as PNG images and also includes programs to convert HDF5 datasets into the formats required by other free visualization software (e.g. plain text, Vis5d and VTK).

In what follows we describe some of the routines provided with **PLUTO**-Chombo for viewing and analyzing HDF5 data using the IDL programming language.

[2] http://www.exelisvis.com/
[3] https://wci.llnl.gov/codes/visit/home.html
[4] http://www.paraview.org/

7.4.1 Visualization with IDL

PLUTO-Chombo comes with a set of visualization routines for the IDL programming language. For more information consult idl_tools.html.

The procedure HDF5LOAD (located in /Tools/IDL/hdf5load.pro) can read a HDF5 data file and store its content on the usual set of variables used during a typical IDL session. HDF5LOAD is directly called from PLOAD (§6.3.1) when the latter is invoked with the `/HDF5` keyword. For instance, in order to read data.0001.hdf5 at the equivalent resolution provided by the 4^{th} refinement level, you need

```
IDL> pload, /hdf5,2,level=4   # will load data.0002.hdf5, ref level = 4
```

As an example, in what follows we load and visualize three density maps of the relativistic Kelvin-Helmholtz test problem, available in the current release (Test_Prob/RMHD/KH). The simulation was performed with six levels of refinement, and we consider the fifth output file, data.0005.hdf5. We load the data set with

```
IDL> pload, /hdf5, 5, level=6   # will load data.0005.2d.hdf5, ref level = 6
```

and visualize with

```
IDL> loadct,6
IDL> display, x1=x1,x2=x2, rho, imax=1.1, imin=0.65
IDL> oplotbox, ctab=3
```

The last command (oplotx) overplots the levels of refinement, utilizing the color table that c_table specifies (in our case 3). The resulting image is seen in the lower part of Fig. 7.1

It may occur that the dataset one wishes to load exceeds the available memory. In that case, it is useful to load only a portion of it. This can be accomplished by specifying in the loading process the patch of the domain that one wishes to display. This is done with the `x1range`, `x2range` and `x3range` keywords. For example

```
IDL> pload, /hdf5, 5,lev=6, x1range=[0.25,0.75], x2range=[0.75,1.25]
     # will load data.0005.2d.hdf5, ref level = 6
     # but only inside the region x in [0.25,0.75], y in [0.75,1.25]
IDL> display, x1=x1,x2=x2, rho, nwin=1, imax=1.1,imin=0.65
IDL> oplotbox, ctab=3
```

The resulting density map is displayed on the upper right part of Fig. 7.1. This is a useful way to effectively zoom in the region of interest without allocating too much memory on loading the whole dataset (see top left of Fig. 7.1):

```
IDL> pload, /hdf5, 5, lev=6, xrange=[0.35,0.45], yrange=[0.9, 1.1]
IDL> display, x1=x1, x2=x2, rho, ims=2, nwin=2, imax=1.1, imin=0.65
IDL> oplotbox, c_table=3
```

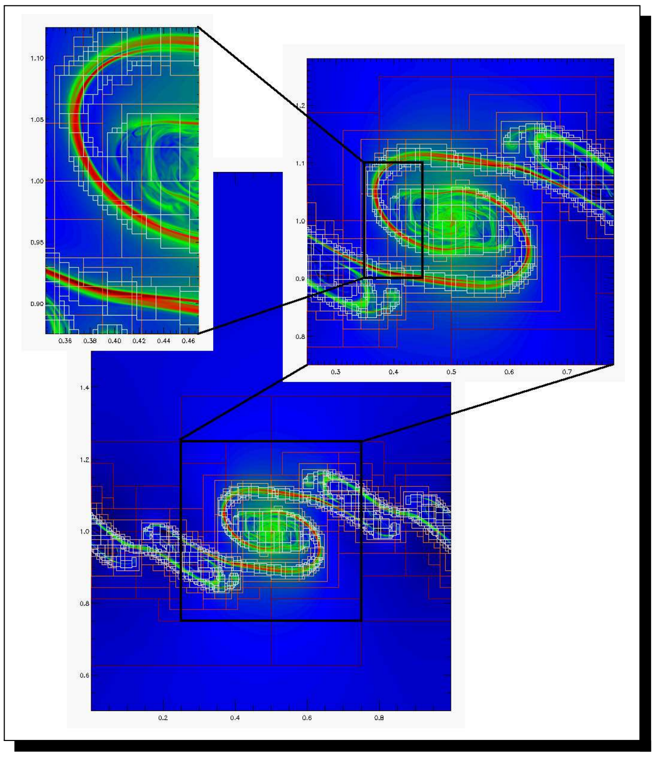


Figure 7.1: Density maps of the relativistic Kelvin-Helmholtz test problem, at the end of integration. The large scale view is on the background/lower part, whereas the thick black boxes denote the zoomed in area of the upper right and upper left panels consecutively. In all panels the refinement levels are displayed, utilizing the oplotbox routine. The close-ups are produced via partially loading selected regions of the entire dataset (see text), considerably reducing memory requirements.

Acknowledgements

Several people have contributed through valuable suggestions and bug reports to the improvement of the code in several aspects. In particular, we wish to thank

- S. Orlando, M. Guarrasi and the group in Palermo (Italy) for valuable suggestions during the development of the thermal conduction module;
- S. Brinkmann, M. Flock for insightful comments and ideas that contributed to the development of some of the new features introduced in **PLUTO** 3;
- M.Flock, M. Stute and S. Kolb for testing and strongly supporting the development of the FARGO-MHD scheme;
- J. Mackey and D. Meyer for pointing out and motivating a careful revision of interpolation on non-cartesian geometries;
- B. Vaidya for the development of the pyPLUTO visualization tool.

A. Equations in Different Geometries

In this section we give the explicit form of the MHD and RMHD equations written in different systems of coordinates. Non-ideal terms such as viscosity, resistivity and thermal conduction are not included here. The discretizations used in the Src/MHD/rhs.c and Src/RMHD/rhs.c strictly follow these form. Equations for the non-magnetized version (HD and RHD) are obtained by setting the magnetic field vector $\boldsymbol{B} = \boldsymbol{0}$.

A.1 MHD Equations

A.1.1 Cartesian Coordinates

In Cartesian coordinates (x, y, z), the conservative ideal MHD Equations (3.7) are discretized using the following divergence form

$$
\begin{array}{lcl}
\dfrac{\partial\rho}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}) & = & 0 \\
\dfrac{\partial m_x}{\partial t} + \nabla\cdot(m_x\boldsymbol{v} - B_x\boldsymbol{B}) + \dfrac{\partial p_t}{\partial x} & = & \rho\left(g_x - \dfrac{\partial\Phi}{\partial x}\right) \\
\dfrac{\partial m_y}{\partial t} + \nabla\cdot(m_y\boldsymbol{v} - B_y\boldsymbol{B}) + \dfrac{\partial p_t}{\partial y} & = & \rho\left(g_y - \dfrac{\partial\Phi}{\partial y}\right) \\
\dfrac{\partial m_z}{\partial t} + \nabla\cdot(m_z\boldsymbol{v} - B_z\boldsymbol{B}) + \dfrac{\partial p_t}{\partial z} & = & \rho\left(g_z - \dfrac{\partial\Phi}{\partial z}\right) \\
\dfrac{\partial}{\partial t}(E+\rho\Phi) + \nabla\cdot\Big[(E+p_t+\rho\Phi)\boldsymbol{v} - \boldsymbol{B}\left(\boldsymbol{v}\cdot\boldsymbol{B}\right)\Big] & = & \rho\boldsymbol{v}\cdot\boldsymbol{g} \\
\dfrac{\partial B_x}{\partial t} + \dfrac{\partial\mathcal{E}_z}{\partial y} - \dfrac{\partial\mathcal{E}_y}{\partial z} & = & 0 \\
\dfrac{\partial B_y}{\partial t} + \dfrac{\partial\mathcal{E}_x}{\partial z} - \dfrac{\partial\mathcal{E}_z}{\partial x} & = & 0 \\
\dfrac{\partial B_z}{\partial t} + \dfrac{\partial\mathcal{E}_y}{\partial x} - \dfrac{\partial\mathcal{E}_x}{\partial y} & = & 0
\end{array}
\tag{A.1}
$$

where $\boldsymbol{v} = (v_x, v_y, v_z)$ and $\boldsymbol{B} = (B_x, B_y, B_z)$ are the velocity and magnetic field vectors, $(\mathcal{E}_x, \mathcal{E}_y, \mathcal{E}_z)$ are the components of the electromotive force $\boldsymbol{\mathcal{E}} = -\boldsymbol{v}\times\boldsymbol{B}$, $\boldsymbol{g}$ is the body force vector and Φ is the gravitational potential.

A.1.2 Polar Coordinates

In polar cylindrical coordinates (R, ϕ, z), the conservative ideal MHD Equations (3.7) are discretized using the following divergence form

$$
\begin{array}{lcl}
\dfrac{\partial \rho}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}) & = & 0 \\
\dfrac{\partial m_R}{\partial t} + \nabla\cdot(m_R\boldsymbol{v} - B_R\boldsymbol{B}) + \dfrac{\partial p_t}{\partial R} & = & \rho\left(g_R - \dfrac{\partial\Phi}{\partial R}\right) + \dfrac{\rho v_\phi^2 - B_\phi^2}{R} \\
\dfrac{\partial m_\phi}{\partial t} + \nabla^R\cdot(m_\phi\boldsymbol{v} - B_\phi\boldsymbol{B}) + \dfrac{1}{R}\dfrac{\partial p_t}{\partial \phi} & = & \rho\left(g_\phi - \dfrac{1}{R}\dfrac{\partial\Phi}{\partial \phi}\right) \\
\dfrac{\partial m_z}{\partial t} + \nabla\cdot(m_z\boldsymbol{v} - B_z\boldsymbol{B}) + \dfrac{\partial p_t}{\partial z} & = & \rho\left(g_z - \dfrac{\partial\Phi}{\partial z}\right) \\
\dfrac{\partial}{\partial t}(E+\rho\Phi) + \nabla\cdot\Big[(E+p_t+\rho\Phi)\boldsymbol{v} - \boldsymbol{B}\,(\boldsymbol{v}\cdot\boldsymbol{B})\Big] & = & \rho\boldsymbol{v}\cdot\boldsymbol{g} \\
\dfrac{\partial B_R}{\partial t} + \dfrac{1}{R}\dfrac{\partial \mathcal{E}_z}{\partial \phi} - \dfrac{\partial \mathcal{E}_\phi}{\partial z} & = & 0 \\
\dfrac{\partial B_\phi}{\partial t} + \dfrac{\partial \mathcal{E}_R}{\partial z} - \dfrac{\partial \mathcal{E}_z}{\partial R} & = & 0 \\
\dfrac{\partial B_z}{\partial t} + \dfrac{1}{R}\dfrac{\partial (R\mathcal{E}_\phi)}{\partial R} - \dfrac{1}{R}\dfrac{\partial \mathcal{E}_R}{\partial \phi} & = & 0\,,
\end{array}
\tag{A.2}
$$

Note that curvature terms are present in the radial component while the azimuthal component is discretized in angular momentum conserving form. The corresponding divergence operators are

$$
\begin{array}{lcl}
\nabla\cdot\boldsymbol{F} & = & \dfrac{1}{R}\dfrac{\partial(RF_R)}{\partial R} + \dfrac{1}{R}\dfrac{\partial F_\phi}{\partial\phi} + \dfrac{\partial F_z}{\partial z}\,, \\
\nabla^R\cdot\boldsymbol{F} & = & \dfrac{1}{R^2}\dfrac{\partial(R^2F_R)}{\partial R} + \dfrac{1}{R}\dfrac{\partial F_\phi}{\partial\phi} + \dfrac{\partial F_z}{\partial z}
\end{array}
\tag{A.3}
$$

In the previous equations $\boldsymbol{v} = (v_R, v_\phi, v_z)$ and $\boldsymbol{B} = (B_R, B_\phi, B_z)$ are the velocity and magnetic field vectors, $(\mathcal{E}_R, \mathcal{E}_\phi, \mathcal{E}_z)$ are the components of the electromotive force $\boldsymbol{\mathcal{E}} = -\boldsymbol{v}\times\boldsymbol{B}$, $\boldsymbol{g}$ is the body force vector and Φ is the gravitational potential.

A.1.3 Spherical Coordinates

In spherical coordinates (r, θ, ϕ) the ideal MHD equations (3.7) are discretized using the following divergence form

$$\begin{array}{lcl}
\dfrac{\partial \rho}{\partial t} + \nabla \cdot (\rho \boldsymbol{v}) & = & 0 \\
\dfrac{\partial m_r}{\partial t} + \nabla \cdot (m_r \boldsymbol{v} - B_r \boldsymbol{B}) + \dfrac{\partial p_t}{\partial r} & = & \rho \left(g_r - \dfrac{\partial \Phi}{\partial r} \right) + \dfrac{\rho v_\theta^2 - B_\theta^2}{r} + \dfrac{\rho v_\phi^2 - B_\phi^2}{r} \\
\dfrac{\partial m_\theta}{\partial t} + \nabla \cdot (m_\theta \boldsymbol{v} - B_\theta \boldsymbol{B}) + \dfrac{1}{r}\dfrac{\partial p_t}{\partial \theta} & = & \rho \left(g_\theta - \dfrac{1}{r}\dfrac{\partial \Phi}{\partial \theta} \right) - \dfrac{\rho v_\theta v_r - B_\theta B_r}{r} + \cot\theta \dfrac{\rho v_\phi^2 - B_\phi^2}{r} \\
\dfrac{\partial m_\phi}{\partial t} + \nabla^r \cdot (m_\phi \boldsymbol{v} - B_\phi \boldsymbol{B}) + \dfrac{1}{r\sin\theta}\dfrac{\partial p_t}{\partial \phi} & = & \rho \left(g_\phi - \dfrac{1}{r\sin\theta}\dfrac{\partial \Phi}{\partial \phi} \right) \\
\dfrac{\partial}{\partial t}(E + \rho\Phi) + \nabla \cdot \Big[(E + p_t + \rho\Phi)\boldsymbol{v} - \boldsymbol{B}\,(\boldsymbol{v}\cdot\boldsymbol{B}) \Big] & = & \rho \boldsymbol{v} \cdot \boldsymbol{g} \\
\dfrac{\partial B_r}{\partial t} + \dfrac{1}{r\sin\theta}\dfrac{\partial(\sin\theta \mathcal{E}_\phi)}{\partial \theta} - \dfrac{1}{r\sin\theta}\dfrac{\partial \mathcal{E}_\theta}{\partial \phi} & = & 0 \\
\dfrac{\partial B_\theta}{\partial t} + \dfrac{1}{r\sin\theta}\dfrac{\partial \mathcal{E}_r}{\partial \phi} - \dfrac{1}{r}\dfrac{\partial (r\mathcal{E}_\phi)}{\partial r} & = & 0 \\
\dfrac{\partial B_\phi}{\partial t} + \dfrac{1}{r}\dfrac{\partial (r\mathcal{E}_\theta)}{\partial r} - \dfrac{1}{r}\dfrac{\partial \mathcal{E}_r}{\partial \theta} & = & 0
\end{array} \tag{A.4}$$

Note that curvature terms are present in the radial and meridional components while the azimuthal component is discretized in angular momentum conserving form. The corresponding divergence operators are

$$\begin{array}{lcl}
\nabla \cdot \boldsymbol{F} & = & \dfrac{1}{r^2}\dfrac{\partial (r^2 F_r)}{\partial r} + \dfrac{1}{r\sin\theta}\dfrac{\partial(\sin\theta F_\theta)}{\partial\theta} + \dfrac{1}{r\sin\theta}\dfrac{\partial F_\phi}{\partial \phi} \\
\nabla^r \cdot \boldsymbol{F} & = & \dfrac{1}{r^3}\dfrac{\partial (r^3 F_r)}{\partial r} + \dfrac{1}{r\sin^2\theta}\dfrac{\partial(\sin^2\theta F_\theta)}{\partial\theta} + \dfrac{1}{r\sin\theta}\dfrac{\partial F_\phi}{\partial \phi}
\end{array} \tag{A.5}$$

In the previous equations $\boldsymbol{v} = (v_r, v_\theta, v_\phi)$ and $\boldsymbol{B} = (B_r, B_\theta, B_\phi)$ are the velocity and magnetic field vectors, $(\mathcal{E}_r, \mathcal{E}_\theta, \mathcal{E}_\phi)$ are the components of the electromotive force $\boldsymbol{\mathcal{E}} = -\boldsymbol{v} \times \boldsymbol{B}$, $\boldsymbol{g}$ is the body force vector and Φ is the gravitational potential.

A.2 (Sperical) Relativistic MHD Equations

A.2.1 Cartesian Coordinates

In Cartesian coordinates (x, y, z), the relativistic MHD equations (3.14) take the form

$$\begin{array}{lcl}
\dfrac{\partial D}{\partial t} + \nabla\cdot(D\boldsymbol{v}) & = & 0 \\
\dfrac{\partial m_x}{\partial t} + \nabla\cdot\left[(w+b^2)v_x\boldsymbol{v} - b_x\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial x} & = & \rho g_x \\
\dfrac{\partial m_y}{\partial t} + \nabla\cdot\left[(w+b^2)v_y\boldsymbol{v} - b_y\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial y} & = & \rho g_y \\
\dfrac{\partial m_z}{\partial t} + \nabla\cdot\left[(w+b^2)v_z\boldsymbol{v} - b_z\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial z} & = & \rho g_z \\
\dfrac{\partial E}{\partial t} + \nabla\cdot(\boldsymbol{m} - D\boldsymbol{v}) & = & D\boldsymbol{v}\cdot\boldsymbol{g} \\
\dfrac{\partial B_x}{\partial t} + \dfrac{\partial \mathcal{E}_z}{\partial y} - \dfrac{\partial \mathcal{E}_y}{\partial z} & = & 0 \\
\dfrac{\partial B_y}{\partial t} + \dfrac{\partial \mathcal{E}_x}{\partial z} - \dfrac{\partial \mathcal{E}_z}{\partial x} & = & 0 \\
\dfrac{\partial B_z}{\partial t} + \dfrac{\partial \mathcal{E}_y}{\partial x} - \dfrac{\partial \mathcal{E}_x}{\partial y} & = & 0
\end{array} \tag{A.6}$$

where $D = \gamma\rho$ is the lab density, $\boldsymbol{m} = (w + b^2)\boldsymbol{v} - \gamma(\boldsymbol{v}\cdot\boldsymbol{B})\boldsymbol{b}$ is the momentum density, w is the gas enthalpy, $b^2 = \boldsymbol{B}^2/\gamma^2 + (\boldsymbol{v}\cdot\boldsymbol{B})^2$, $\boldsymbol{v} = (v_x, v_y, v_z)$ is the velocity, $\boldsymbol{B} = (B_x, B_y, B_z)$ is the magnetic field in the lab frame, $\boldsymbol{b} = \boldsymbol{B}/\gamma + \gamma(\boldsymbol{v}\cdot\boldsymbol{B})\boldsymbol{v}$ is the covariant field, $(\mathcal{E}_x, \mathcal{E}_y, \mathcal{E}_z)$ are the components of the electromotive force $\boldsymbol{\mathcal{E}} = -\boldsymbol{v}\times\boldsymbol{B}$ and $\boldsymbol{g}$ is the body force vector.

A.2.2 Polar Coordinates

In polar cylindrical coordinates (R, ϕ, z), the RMHD Equations (3.14) are discretized using the following form

$$\begin{array}{lcl}
\dfrac{\partial D}{\partial t} + \nabla\cdot(D\boldsymbol{v}) & = & 0 \\
\dfrac{\partial m_R}{\partial t} + \nabla\cdot\left[(w+b^2)v_R\boldsymbol{v} - b_R\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial R} & = & \rho g_R + \dfrac{m_\phi v_\phi}{R} - \left(\dfrac{B_\phi}{\gamma^2} + (\boldsymbol{v}\cdot\boldsymbol{B})v_\phi\right)\dfrac{B_\phi}{R} \\
\dfrac{\partial m_\phi}{\partial t} + \nabla^R\cdot\left[(w+b^2)v_\phi\boldsymbol{v} - b_\phi\boldsymbol{b}\right] + \dfrac{1}{R}\dfrac{\partial p_t}{\partial \phi} & = & \rho g_\phi \\
\dfrac{\partial m_z}{\partial t} + \nabla\cdot\left[(w+b^2)v_z\boldsymbol{v} - b_z\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial z} & = & \rho g_z \\
\dfrac{\partial E}{\partial t} + \nabla\cdot(\boldsymbol{m} - D\boldsymbol{v}) & = & D\boldsymbol{v}\cdot\boldsymbol{g} \\
\dfrac{\partial B_R}{\partial t} + \dfrac{1}{R}\dfrac{\partial \mathcal{E}_z}{\partial \phi} - \dfrac{\partial \mathcal{E}_\phi}{\partial z} & = & 0 \\
\dfrac{\partial B_\phi}{\partial t} + \dfrac{\partial \mathcal{E}_R}{\partial z} - \dfrac{\partial \mathcal{E}_z}{\partial R} & = & 0 \\
\dfrac{\partial B_z}{\partial t} + \dfrac{1}{R}\dfrac{\partial (R\mathcal{E}_\phi)}{\partial R} - \dfrac{1}{R}\dfrac{\partial \mathcal{E}_R}{\partial \phi} & = & 0\,,
\end{array} \tag{A.7}$$

Note that curvature terms are present in the radial component while the azimuthal component is discretized in angular momentum conserving form. The corresponding divergence operators are

$$\begin{aligned}
\nabla\cdot\boldsymbol{F} &= \frac{1}{R}\frac{\partial(RF_R)}{\partial R}+\frac{1}{R}\frac{\partial F_\phi}{\partial\phi}+\frac{\partial F_z}{\partial z}\,, \\
\nabla^R\cdot\boldsymbol{F} &= \frac{1}{R^2}\frac{\partial(R^2F_R)}{\partial R}+\frac{1}{R}\frac{\partial F_\phi}{\partial\phi}+\frac{\partial F_z}{\partial z}
\end{aligned} \tag{A.8}$$

In the previous equations $\boldsymbol{v}=(v_R,v_\phi,v_z)$ and $\boldsymbol{B}=(B_R,B_\phi,B_z)$ are the velocity and magnetic field vectors, $(\mathcal{E}_R,\mathcal{E}_\phi,\mathcal{E}_z)$ are the components of the electromotive force $\boldsymbol{\mathcal{E}}=-\boldsymbol{v}\times\boldsymbol{B}$, $\boldsymbol{g}$ is the body force vector and Φ is the gravitational potential.

A.2.3 Spherical Coordinates

In spherical coordinates (r,θ,ϕ) the RMHD equations (3.14) are discretized using the following divergence form

$$\begin{aligned}
&\frac{\partial D}{\partial t}+\nabla\cdot(D\boldsymbol{v}) &&= 0 \\
&\frac{\partial m_r}{\partial t}+\nabla\cdot\big[(w+b^2)v_r\boldsymbol{v}-b_r\boldsymbol{b}\big]+\frac{\partial p_t}{\partial r} &&= \rho g_r+\frac{m_\theta v_\theta+m_\phi v_\phi}{r}+ \\
& &&\quad -\left(\frac{B_\theta}{\gamma^2}+(\boldsymbol{v}\cdot\boldsymbol{B})v_\theta\right)\frac{B_\theta}{r}-\left(\frac{B_\phi}{\gamma^2}+(\boldsymbol{v}\cdot\boldsymbol{B})v_\phi\right)\frac{B_\phi}{r} \\
&\frac{\partial m_\theta}{\partial t}+\nabla\cdot\big[(w+b^2)v_\theta\boldsymbol{v}-b_\theta\boldsymbol{b}\big]+\frac{1}{r}\frac{\partial p_t}{\partial\theta} &&= \rho g_\theta-\frac{m_\theta v_r-\cot\theta m_\phi v_\phi}{r} \\
& &&\quad +\left(\frac{B_\theta}{\gamma^2}+(\boldsymbol{v}\cdot\boldsymbol{B})v_\theta\right)\frac{B_r}{r}-\cot\theta\left(\frac{B_\phi}{\gamma^2}+(\boldsymbol{v}\cdot\boldsymbol{B})v_\phi\right)\frac{B_\phi}{r} \\
&\frac{\partial m_\phi}{\partial t}+\nabla^r\cdot\big[(w+b^2)v_\phi\boldsymbol{v}-b_\phi\boldsymbol{b}\big]+\frac{1}{r\sin\theta}\frac{\partial p_t}{\partial\phi} &&= \rho g_\phi \\
&\frac{\partial E}{\partial t}+\nabla\cdot(\boldsymbol{m}-D\boldsymbol{v}) &&= D\boldsymbol{v}\cdot\boldsymbol{g} \\
&\frac{\partial B_r}{\partial t}+\frac{1}{r\sin\theta}\frac{\partial(\sin\theta\mathcal{E}_\phi)}{\partial\theta}-\frac{1}{r\sin\theta}\frac{\partial\mathcal{E}_\theta}{\partial\phi} &&= 0 \\
&\frac{\partial B_\theta}{\partial t}+\frac{1}{r\sin\theta}\frac{\partial\mathcal{E}_r}{\partial\phi}-\frac{1}{r}\frac{\partial(r\mathcal{E}_\phi)}{\partial r} &&= 0 \\
&\frac{\partial B_\phi}{\partial t}+\frac{1}{r}\frac{\partial(r\mathcal{E}_\theta)}{\partial r}-\frac{1}{r}\frac{\partial\mathcal{E}_r}{\partial\theta} &&= 0
\end{aligned} \tag{A.9}$$

Note that curvature terms are present in the radial and meridional components while the azimuthal component is discretized in angular momentum conserving form. The corresponding divergence operators are

$$\begin{aligned}
\nabla\cdot\boldsymbol{F} &= \frac{1}{r^2}\frac{\partial(r^2F_r)}{\partial r}+\frac{1}{r\sin\theta}\frac{\partial(\sin\theta F_\theta)}{\partial\theta}+\frac{1}{r\sin\theta}\frac{\partial F_\phi}{\partial\phi} \\
\nabla^r\cdot\boldsymbol{F} &= \frac{1}{r^3}\frac{\partial(r^3F_r)}{\partial r}+\frac{1}{r\sin^2\theta}\frac{\partial(\sin^2\theta F_\theta)}{\partial\theta}+\frac{1}{r\sin\theta}\frac{\partial F_\phi}{\partial\phi}
\end{aligned} \tag{A.10}$$

Bibliography

[1] Alexiades, V., Amiez, A., & Gremaud E.-A. 1996, Com. Num. Meth. Eng., 12, 31

[2] Balbus, S. A. 1986, ApJ, 304, 787

[3] Balsara, D. S. & Spicer, S. D. 1999, J. Comput. Phys., 149, 270

[4] Balsara, D. S., Tilley, D. A., & Howk, J. C. 2008, MNRAS, 386, 627

[5] Beckers, J.M. 1992, SIAM J. Numer. Anal. 29, 701-713

[6] Borges R., Carmona M., Costa B., Don W.S. 2008, J. Comput. Phys. 227 3191-3211.

[7] Cada P.& Torrilhon M. 2009, J. Comput. Phys. 228, 4118.

[8] Colella, P. & Woodward, P. R. 1984, J. Comput. Phys., 54, 174

[9] Colella, P. 1985, SIAM J. Sci. Stat. Comput. 6, 104-117.

[10] Colella, P. 1990, J. Comput. Phys., 87, 171

[11] Cowie, L. L. & McKee, C. F. 1977, APJ, 211, 135

[12] Courant, R., Friedrichs, K. O. & Lewy, H. 1928, Math. Ann., 100, 32

[13] Dedner, A., Kemm, F., Kröner, D., Munz, C.-D., Schnitzer T., and Wesenberg, M. 2002, J. Comput. Phys., 175, 645

[14] Gardiner, T. A., & Stone, J. M. 2005, J. Comp. Phys., 205, 509

[15] Harten A., Engquist B., Osher S., Chakravarthy S. 1987, J. Comput. Phys. 71, 231

[16] Jiang, G. & Shu, C.-W. 1996, J. Comput. Phys. 126, 202

[17] Jiang, G. & Wu, C.-C. 1999, J. Comput. Phys. 150, 561

[18] Landau, L. D., & Lifshitz, E. M. 1987, Fluid Mechanics, 2nd edition, Pergamon Press, Oxford .

[19] Liou, M.-S. 1996, J. Comp. Phys., 129, 364

[20] Londrillo, P., & Del Zanna, L., 2004, J. Comp. Phys, **195**, 17

[21] Löhner, R. 1987, Computer Methods in Applied Mechanics and Engineering, **61**, 323

[22] Kley, W. 1998, A&A, 338, L37

[23] Masset, F. 2000, A&A, 141, 165

[24] Martí, J. M. & Müller, E. 1996, J. Comput. Phys., 123, 1

[25] Mignone, A., & Bodo, G. 2005, MNRAS, 364, 126

[26] Mignone, A., Plewa, T., & Bodo, G. 2005, Astrophysical Journal Supplement, 160, 199

[27] Mignone, A. 2007, J. Comp. Phys.,

[28] Mignone, A., Bodo, G., Massaglia, S., Matsakos, T., Tesileanu, O., Zanni, C., & Ferrari, A. 2007, Astrophysical Journal Supplement, 170, 228

[29] Mignone, A., Zanni, C., Tzeferacos, P., van Straalen, B., Colella, P., and Bodo, G., 2012, Astrophysical Journal Supplement, 198, 7

[30] Mignone, A., Flock, M., Stute, M., Kolb, S. M., & Muscianisi, G. 2012, A&A, 545, A152

[31] Mignone, A., & McKinney, J. C. 2007, MNRAS, 378, 1118

[32] Mignone, A., Ugliano, M., & Bodo, G. 2009, MNRAS, 393, 1141

[33] Mignone, A., & Tzeferacos, P. 2010, Journal of Computational Physics, 229, 2117

[34] Mignone, A., Tzeferacos, P., Bodo, G. 2010, Journal of Computational Physics, 229, 5896

[35] Mignone, A., Flock, M., Stute, M., Kolb, S.M. and Muscianisi, G. 2012, Astronomy & Astrophysics, in press

[36] Miyoshi, T., & Kusano, K. 2005, Journal of Computational Physics, 208, 315

[37] Orlando, S., Bocchino, F., Reale, F., Peres, G., & Pagano, P. 2008, ApJ, 678, 274

[38] Kenneth G. Powell, NASA CR-194902 ICASE Report No. 94-24, April 1994, pp. 15.

[39] Liu X.-D., Osher S., Chan T. 1994, J. Comput. Phys. 115, 200

[40] Powell, K. G., Roe, P.L., Linde, T., Gombosi, T.I. & De Zeeuw, D.L 1999, Journal of Computational Physics, 154,284

[41] P. L. Roe. 1981, Journal of Computational Physics, 43:357-372, 1981.

[42] Saltzman, J. 1994, J. Comp. Phys., 115, 153

[43] Shu C.-W., Osher S. 1989, J. Comput. Phys. 83, 32

[44] Spitzer, L. 1962, Physics of fully ionized gases (New York: Interscience, 1962)

[45] Strang, G., 1968, SIAM J. Num. Anal., 5, 506

[46] Suresh A., Huynh H.T., 1997, J. Comput. Phys. 136, 83-99

[47] Synge, J. L. 1957, The relativistic Gas, North-Holland Publishing Company

[48] Taub, A. H. 1948, Physical Review, 74, 328

[49] Teşileanu, O., Mignone, A.,& Massaglia, S. 2008, A & A, 488, 429

[50] Toro, E. F. 1997, Riemann Solvers and Numerical Methods for Fluid Dynamics, Springer-Verlag, Berlin

[51] B. van Leer 1979, J. Comput. Phys. 32, 101-136

[52] Yamaleev, N.K. & Carpenter, M.H. 2009, J. Comput. Phys. 228, 3025-3047

[53] L. Del Zanna & N. Bucciantini 2002, A & A, 390, 1177

[54] Del Zanna, L., Bucciantini, N., & Londrillo, P. 2003, Astronomy & Astrophysics, 400, 397