PLUTO v. 4.3 (June 2018)

User's Guide

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

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

Contributors: C. Zanni[2] [zanni@oato.inaf.it]
B. Vaidya[3] [bvaidya@iiti.ac.in]
G. Muscianisi[4]
P. Tzeferacos[5]
O. Tesileanu[6]
T. Matsakos

1 Dipartimento di Fisica, 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 Indian Institute of Technology Indore, Indore, Khandwa Road, Simrol (Indore)
4 Consorzio Interuniversitario CINECA, via Magnanelli, 6/3, 40033 Casalecchio di Reno (Bologna), Italy
5 FLASH Center, University of Chicago, USA
6 Department of Physics, University of Bucharest, Str. Atomistilor nr. 405, RO-077125 Magurele, Ilfov, Romania

Terms & Conditions of Use

Contents

0. Quick Start

0.1 Downloading and unpacking PLUTO

PLUTO can be downloaded from http://plutocode.ph.unito.it. 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

```
~> export PLUTO_DIR=/home/user/PLUTO # If you're using the bash shell;
~> setenv PLUTO_DIR /home/user/PLUTO # If you're using the tcsh 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:

1. Change directory to any of the test problems under PLUTO/Test_Problems, e.g.

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

2. Copy the header and initialization files from a configuration of our choice (e.g. #01):

   ```
   ~/PLUTO/Test_Problems/HD/Sod> cp definitions_01.h definitions.h
   ~/PLUTO/Test_Problems/HD/Sod> cp pluto_01.ini pluto.ini
   ```

3. Run the Python script using

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

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

4. Once you return to the main menu, select "Change makefile", choose a suitable makefile (e.g. Linux.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.

5. Exit from the main menu ("Quit" or press 'q') and type

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

   to compile the code.

6. You can now run the code by typing

```
~/PLUTO/Test_Problems/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

1. Change directory to PLUTO/Test_Problems/MHD/Orszag_Tang.
2. Choose a configuration (e.g. #02) and copy the corresponding configuration files, i.e.,

```
~/PLUTO/Test_Problems/MHD/Orszag_Tang> cp definitions_02.h definitions.h
~/PLUTO/Test_Problems/MHD/Orszag_Tang> cp pluto_02.ini pluto.ini
```

3. Run the Python script:

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

select "Setup problem" and choose the default setting by pressing enter;

4. Once you return to the main menu, select "Change makefile" and choose a suitable makefile (e.g. Linux.gcc.defs) and press enter.
5. 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 *[Static Grid Output]* block, to `multiple_files`. Finally, edit definitions.h and change `PRINT_TO_FILE` from *YES* to *NO*.
6. Compile the code:

```
~/PLUTO/Test_Problems/MHD/Orszag_Tang> make
```

7. If compilation was successful, you can now run the code by typing

```
~/PLUTO/Test_Problems/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 and x velocity) 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 "rho.0001.dbl" bin array=200x200 format="%double" # density
gnuplot> splot "vx1.0001.dbl" bin array=200x200 format="%double" # density
```

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 §12.3.

[1]You need to include PLUTO/Tools/IDL into your IDL search path, §12.3.2

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.

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
   ```

2. 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.

3. 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.

4. Finally, we complete the python session by setting the architecture for the makefile. In the makefile menu choose your system configuration (e.g. Linux.gcc.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);

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

if (r <= 0.3) v[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

The official distribution of **PLUTO** comes with several examples and test problems that can be found under the Test_Problems/ folder. Documentation is extracted from comments at the beginning of init.c sources files using the Doxygen documentation system and an on-line documentation browser can be found in Doc/test_problems.html. Test problem documentation is still being added and more examples will be available in future releases.

0.6 Migrating from PLUTO 4.2 to PLUTO 4.3

PLUTO 4.2 provides several bug fixes and does not introduce major changes with respect to version 4.1 in the syntax of the basic functions defined in init.c. Some optimizations and improvements have been performed in the source distribution and a few minor changes have been introduced mainly for uniformity and efficiency reasons. The file CHANGES lists the most relevant ones:

```
Major Changes / Addition:
=========================

- The Grid structure has been changed from an array of structures to a structure
  of arrays.
  All arrays must be changed as

  grid[IDIR].x    must be chnaged to   grid->x[IDIR]

  and so on.
  Areas and volumes are now fully three-dimensional arrays containing the
  full expression rather than just the reduced one
  (i.e., Ar = R*dphi*dz and not just R).
- Log files are now always dumped to screen (in serial mode) or to different
  log files (in parallel, one per processor). It is possible to write log files
  into a separate directory using "log_dir" in your pluto.ini;
  In addition, print1 statement has been removed (only print is used).
- The "Frequently Used Options" (meaning the additional switches at the end of
  definitions.h) have been completely removed and reabsorbed into the more
  general "User-defined Constants", see the user-guide.
  Default values are used when not explicitly set.
- Added Particles modules (CR + LP), see documentation.
- Added forced turbulence module (see doc).
- Added RK_LEGENDRE option to integrate parabolic terms.
- The Asynchrounous I/O option has been removed.
- Python setup is compatible with Python 2.7.x and 3.x
- Magnetic field variable names have been changed
  from "bx1", "bx2", "bx3" to "Bx1", "Bx2", "Bx3"
- In the userdef_output.c, ChangeDumpVar() --> ChangeOutputVar()
                           SetDumpVar()    --> SetOutputVar()
- A new function, InitDomain(), has been introduced to assign initial conditions
  in a more global way. This does *NOT* replace the old Init() function.
- A more efficient InputData() module to assign initial conditions from
  external binary files;
- Added different pseudo random number generators (PRNG);
- The --show-dec options has been removed (now every processor
  log print its own domain);
- IDL can now read user-defined variables (see documentation);
- Time_Step structure name changed into timeStep (respect naming conventions!);



Bug Fixes:
```

1. Introduction

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

$$\frac{\partial \boldsymbol{U}}{\partial t} + \nabla\cdot\mathsf{T}_h = \nabla\cdot\mathsf{T}_p + \boldsymbol{S}(\boldsymbol{U}) \qquad (1.1)$$

where $\boldsymbol{U}$ represents a set of conservative quantities, $\mathsf{T}_h(\boldsymbol{U})$ is the (hyperbolic) flux tensor, T_p is the diffusion (parabolic) flux tensor and $\boldsymbol{S}(\boldsymbol{U})$ defines the source terms [MBM+07, MZT+12]. 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, §6.1;
- *MHD*: ideal/resistive magnetohydrodynamics, §6.2;
- *RHD*: special relativistic hydrodynamics, §6.3;
- *RMHD*: special (ideal) relativistic magnetohydrodynamics, §6.4;

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 [MBM+07] 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 13). A detailed description of the AMR implementation is given in [MZT+12].

Doxygen is used as the standard documentation system and the Application Programming Interface (API) reference guide can be found in Doc/API-ReferenceGuide.html.

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.

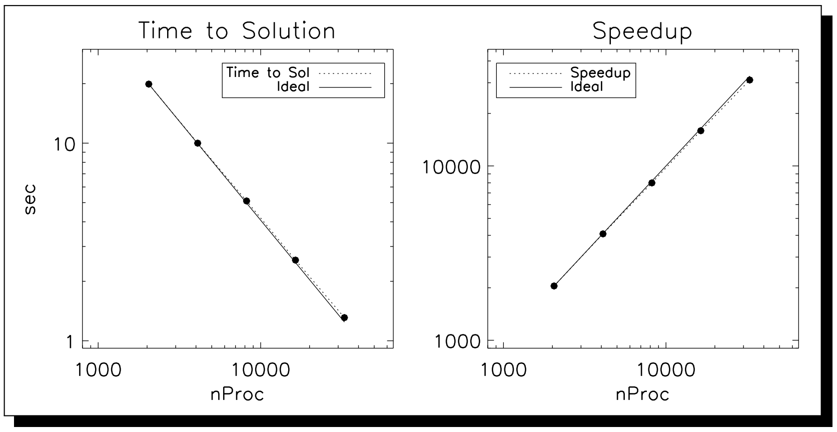


Figure 1.1: Strong scaling of PLUTO on a periodic domain problem with 512^3 grid zones. Left panel: average execution time (in seconds) per step vs. number of processors. Right panel: speedup factor computed as T_1/T_N where T_1 is the (inferred) execution time of the sequential algorithm and T_N is 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.

1.1 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, a `C` compiler and the `make` utility. These are usually installed by default on most Linux/Unix platforms. A comprehensive list is shown in Table 1.1.

	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 (v 3.2)	–	–	yes	yes
HDF5 library (v 1.6 or 1.8)	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.

Starting with **PLUTO** 4 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 13), 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.

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, `pyPLUTO`, etc...;
- Test_Problems/: a directory containing several test-problems 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 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.

Option	Description
`--with-chombo`	enables support for adaptive mesh refinement (AMR) using the Chombo library, Chapter 13;
`--with-fd`	enables support for finite difference schemes, §10.3
`--with-fargo`	enables support for the FARGO-MHD module, §10.2;
`--with-particles`	enables support for the particles module, see Chapter 11;
`--with-sb`	enables support for the shearing-box module, §10.1;
`--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.

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:

- definitions.h: header file containing all problem-dependent preprocessor directives required at compilation time (physics module, geometry, dimensions, etc.). This is the subject of Chapter 2.
- makefile: needed to compile **PLUTO** and it depends on your system architecture. This is described in Chapter 3.
- pluto.ini: startup initialization file containing run-time parameters (grid size, CFL,...). This is the subject of Chapter 4;
- init.c: implements initial, boundary conditions, etc.... See Chapter 5.

The Python script setup.py is used for the first two steps while the remaining files (pluto.ini and init.c) 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 the Test_Problems/ directory.

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

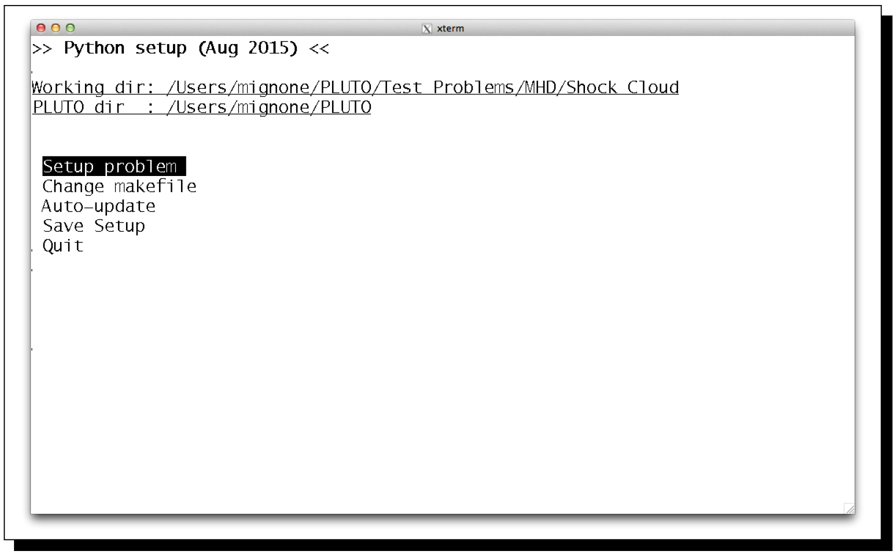
```
~> setenv PLUTO_DIR /home/user/PLUTO  # if you're using tcsh shell
~> export PLUTO_DIR=/home/user/PLUTO  # if you're using bash shell
```

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]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.



```
>> Python setup (Aug 2015) <<

Working dir: /Users/mignone/PLUTO/Test_Problems/MHD/Shock_Cloud
PLUTO dir  : /Users/mignone/PLUTO


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

Figure 1.2: Python script main menu.

1.4 Compiling & Running the Code

After the four basic configuration files (init.c, definitions.h, makefile and pluto.ini) have been created, **PLUTO** can be compiled from your local working directory by typing

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

It is important to remember that the makefile created by Python (Chapter 3) 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.

1.4.1 Output Log file

During execution, the integration log will look something like:

```
  ...
step:0; t = 0.0000e+00; dt = 1.0000e-04; 0.0 %
        [Mach = 1.236510, NRiemann = 10]
step:1; t = 1.0000e-04; dt = 1.1000e-04; 0.0 %
        [Mach = 1.431883, NRiemann = 6]
step:2; t = 2.1000e-04; dt = 1.2100e-04; 0.0 %
        [Mach = 1.389746, NRiemann = 6]
step:3; t = 3.3100e-04; dt = 1.3310e-04; 0.0 %
        [Mach = 1.326274, NRiemann = 5]
 ...
```

where `step` gives the current integration step, `t` is the current integration time, `dt` is the current time step, `x.x%` is the percentage of integration. The two numbers in square brackets are printed at the end of the step and are, respectively, the maximum Mach number and maximum number of iterations required by the Riemann solver (if an iterative one is used, e.g. `two_shock`) during the current step.

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.

Starting with **PLUTO** 4.3, the output log will be dumped on screen if the code has been compiled in serial mode. In parallel, each processors writes to a dedicated file pluto.n.log where n is the processor number. Since computations on many cores produce as many log files, it is possible to write them into a dedicated directory if desired (use the `log_dir` in pluto.ini, see §4.6). If a log file does not 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.

1.4.2 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, only for static grids). The input data files are read from the directory specified by the `output_dir` variables in pluto.ini (default is current working directory). With Chombo-AMR this switch is equivalent to `-restart`.	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, for static grid) or Chombo checkpoint file (chk.nnnn.hdf5 for Chombo-AMR). For the static grid, input data files are read from the directory specified by the `output_dir` variables in pluto.ini (default is current working directory).	Yes
`-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 to your local working directory and then edit it, 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.

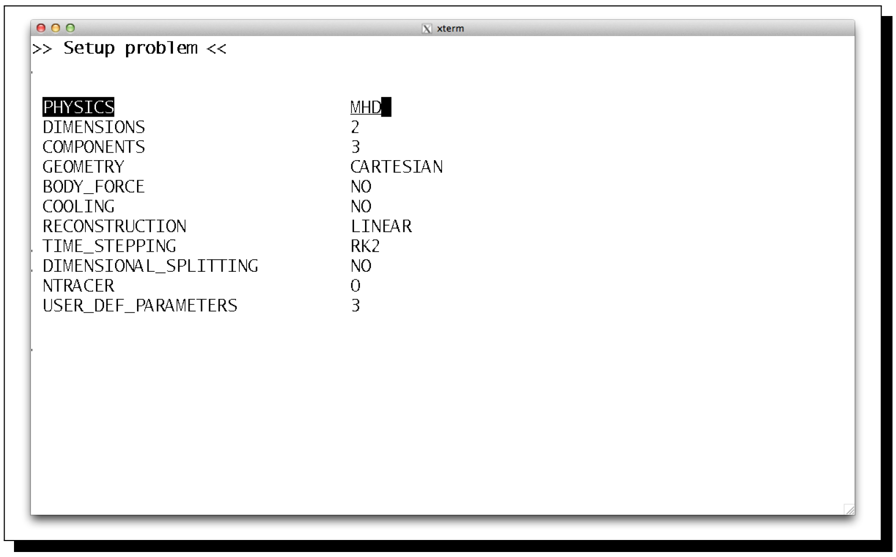
Note: Header files (*.h or *.H) do not follow the same convention and *must not* be copied to the local working directory. Modifications to header files must therefore be done in the original directory.

2. Problem Header File: definitions.h

This chapter explains how to create the configuration header file definitions.h for a specific problem.

2.1 Basic Options

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 the existing one.



```
>> Setup problem <<


PHYSICS                      MHD
DIMENSIONS                   2
COMPONENTS                   3
GEOMETRY                     CARTESIAN
BODY_FORCE                   NO
COOLING                      NO
RECONSTRUCTION               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 options (*user-defined constants*) that are not accessible via the Python script and should be changed manually (§2.3). 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, §6.1;
- *MHD*: single fluid, ideal/resistive magnetohydrodynamics, §6.2;
- *RHD*: special relativistic hydrodynamics, §6.3;
- *RMHD*: special relativistic magnetohydrodynamics, §6.4.

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 all cases 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 §5.4.

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 $\rho^2 T^\alpha$ (§9.1);
- *TABULATED*: radiative losses are computed as $n^2\Lambda(T)$, where $\Lambda(T)$ is a user-supplied tabulated function of temperature, see §9.2. Alternatively, this module can be used to provide user-defined cooling functions;
- *SNEq*: simplified non-equilibrium cooling function for atomic hydrogen. See §9.3 for more details;
- *H2_COOL*: optically thin cooling function for molecular and atomic hydrogen. See §9.4.
- *MINEq*: multi-ion non-equilibrium cooling model. It evolves the standard equations augmented with a chemical network of 29 ions, see §9.5 and the work by [TMM08].

2.1.6 RECONSTRUCTION

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 reconstruction is applied to primitive variables. It is 2nd order accurate in space. Stencil is 3 point wide.
- *WENO3*: provides 3rd order weighted essentially non-oscillatory reconstruction [YC09] inside a cell using is 3-point stencil.
- *LimO3*: provides 3rd order limiter function [ČT09] based on a 3-point stencil.
- *PARABOLIC*: piecewise parabolic method (PPM) as implemented by [Mig14]. 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 and curvilinear coordinates are handled more correctly with *LINEAR* and *PARABOLIC* using the approach presented in [Mig14].

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, §10.3.

2.1.7 TIME_STEPPING

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 and $\boldsymbol{\mathcal{L}}$ denotes the discretized spatial operator on the right hand side of Eq. (1.1), the possible options are:

- *EULER*: 1st (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*: 2nd or 3rd-order TVD Runge Kutta is used to advance the solution to the next time level:

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` to *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 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 linear reconstruction while *CHARACTERISTIC_TRACING* which does a more sophisticated characteristic limiting, can be combined with all reconstruction algorithms. The original PPM scheme of [CW84, MM96] is available for the `HD`, `MHD` and `RHD` modules by selecting `TIME_STEPPING` to *CHARACTERISTIC_TRACING*, together with `RECONSTRUCTION` to *PARABOLIC* and a *two-shock* Riemann solver (*Roe* or *hlld* alternatively).

Setting `DIMENSIONAL_SPLITTING` = *NO* yields the spatially unsplit fully corner-coupled method of [Col90, MPB05]. 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 [Bec92]. 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 (§8.5.1). On the contrary, when parabolic terms are included via Super-Time-Stepping integration (§8.5.2) the time step can be much larger being computed solely from the advection time scale (i.e. $\tau_d = 0$ in the table below).

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.

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

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 *HNCK* 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), [Bec92].

2.1.8 DIMENSIONAL_SPLITTING

Set this feature to *YES* if you intend to use Strang operator splitting [Str68] to solve multidimensional equations by a sequence of 1D problem. 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.

RECONST.	TIME_ST.	DIM._SPL.	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 [van79, Tor97]. Computationally more efficient than RK integrators, it is probably the fastest $2^{\rm nd}$ order algorithm. Works well for the HD/RHD modules and the MHD/RMHD modules (with *DIVERGENCE_CLEANING* or *EIGHT_WAVES*), 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 [van79, Col85].
PARABOLIC	*ChTr*	*YES*	$N_{\rm dim}$	Gives the original Piecewise-Parabolic-Method (PPM) of [CW84]. Suggested for HD, RHD or MHD (with *DIVERGENCE_CLEANING* or *EIGHT_WAVES*) in 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 [Col90, Sal94, MPB05] and [GS05] for the HD/RHD or MHD/RMHD modules (the RMHD version works only with *HANCOCK*). It is fully unsplit and stable up to 1 (in 2-D) and ~ 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*.

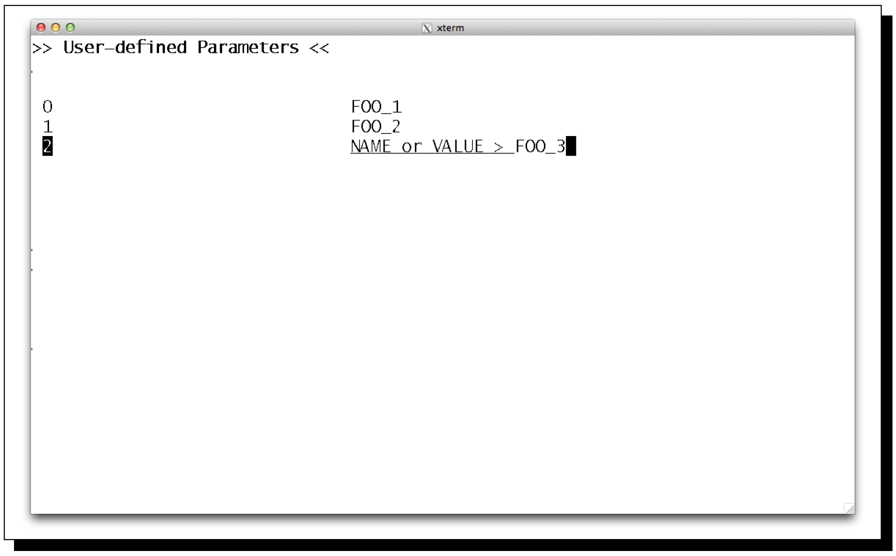
2.1.9 NTRACER

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

$$\frac{\partial Q_k}{\partial t} + \boldsymbol{v}\cdot\nabla Q_k = 0 \qquad\Longleftrightarrow\qquad \frac{\partial(\rho Q_k)}{\partial t} + \nabla\cdot\big(\rho Q_k \boldsymbol{v}\big) = 0 \tag{2.2}$$

The second form gives the conservative equation and it is the one actually being discretized by the code. The array index used to access tracer variables (§5.1,§5.3) is `TRC` for the first tracer, `TRC+1` for the second one and so on. The maximum number is 8.

2.1.10 USER_DEF_PARAMETERS



```
>> User-defined Parameters <<


 0                        FOO_1
 1                        FOO_2
 2                        NAME or VALUE > FOO_3
```

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

Sets the number of user-defined parameters that can be changed at runtime and accessed from anywhere in the code. 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 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 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.

The maximum number is 31 and parameter names should be chosen with care in order to avoid overlapping conflicts with names that are already defined in the code. Although there are no strict rules, we strongly advise to use capital letters, to avoid short labels such as "`V0`" or "`VX`" and to 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 after the execution of the python script. However, if you use a different initialization file than pluto.ini, you may have to set the parameter names together with their values manually.

2.2 Physics-Dependent Options

After the physics module has been selected, further options become available in a following menu. Depending on the selected physics module, different options may appear. They are described in the following.

2.2.1 BACKGROUND_FIELD

If set to *YES*, the magnetic field is split into a static curl-free contribution and a time dependent deviation. This option is only available for the *MHD* module and it is described in detail in §6.2.4.

2.2.2 DIVB_CONTROL

Numerical methods do not naturally preserve the condition $\nabla\cdot\boldsymbol{B} = 0$. For this reason, this option allows the user to select a control strategy to enforce the $\nabla\cdot\boldsymbol{B} = 0$ constraint. Possible values are

- *NO*
  divergence constraint is not enforced. 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 [Pow94] is used, see §6.2.3.1;
- *DIV_CLEANING*
  magnetic fields retain a cell average representation and the mixed hyperbolic/parabolic divergence cleaning technique of [DKK$^+$02, MTB10] is used, see §6.2.3.2. 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 §6.2.3.3.

2.2.3 EOS

Select the equation of state (EOS):

- *IDEAL*
  use the perfect gas law with constant specific heat ratio. This EOS is available for all physics module and it is described in §7.2.
- *ISOTHERMAL*
  use an isothermal equation of state (§7.1). In this case, the energy equation is not included.
- *PVTE_LAW*
  allows the user to employ / implement more complex equations of state by specifying the caloric EOS as $e = e(\rho, T)$. This EoS works for the HD and MHD module and is described in §7.3.
- *TAUB*
  use the Taub-Matthews equation of state (only for relativistic modules), see §7.4.

Please refer to Chapter 7 for a detailed description.

2.2.4 ENTROPY_SWITCH

By enabling this switch (only for the *IDEAL* or *TAUB* EOS), the equation of entropy is added to the system of conservation laws:

$$\frac{\partial\sigma_c}{\partial t} + \nabla\cdot(\sigma_c\boldsymbol{v}) = S_\sigma \tag{2.3}$$

where $\sigma_c = \rho\sigma$ (for *HD* or *MHD*) or $\sigma_c = D\sigma$ (for relativistic flows), σ is the entropy (e.g., $\sigma = p/\rho^\Gamma$ for the *IDEAL* EOS) and S_σ is a source term accounting for dissipative terms in the HD or MHD modules (see Eq. [3.27] of [BS03]):

$$S_\sigma = (\Gamma-1)\rho^{1-\Gamma}\left[\nabla\cdot(\kappa\nabla T) + \eta\boldsymbol{J}^2 + \mu\Pi_{ij}\frac{\partial v_i}{\partial x_j}\right] . \tag{2.4}$$

Equation (2.3) is solved in a conservative way and entropy is essentially treated as a passive scalar. However, starting with **PLUTO** 4.3, the source term (2.4) is never actually included since the total energy equation is *always* used when adding parabolic terms in a conservative fashion, see the documentation of **`ParabolicUpdate()`**. This allows the entropy switch to be used with any of the nonideal terms described in Chapter 8.

The entropy switch acts in a selective way allowing gas pressure to be recovered from either the total energy or the entropy:

$$\left\{\begin{array}{lll} p = p(E) & \text{if} & \mathcal{F}_\sigma = 0 \\ p = p(\sigma_c) & \text{if} & \mathcal{F}_\sigma = 1 \end{array}\right. \tag{2.5}$$

Here $\mathcal{F}_\sigma$ tells if a zone has been flagged with `FLAG_ENTROPY` at the beginning of the step and it depends on the value assigned to `ENTROPY_SWITCH`:

- *NO*
  the entropy equation is not included ($\mathcal{F}_\sigma = 0$ everywhere);

- *SELECTIVE*
  by enabling this option a selective update is employed. By default, all zones are flagged to be updated using the entropy equation unless they lie in proximity of a shock wave. Thus for each zone, we evaluate

  $$\mathcal{F}_\sigma = \left\{\begin{array}{lll} 0 & \text{if} & \tilde{\nabla}\cdot\boldsymbol{v} < 0 \quad \underline{\text{and}} \quad \|\tilde{\nabla}p\|/p > \epsilon_p \\ 1 & \text{otherwise,} & \end{array}\right. \tag{2.6}$$

  where $\tilde{\nabla}$ is a three-point undivided difference operator and ϵ_p sets the shock strength threshold (see `EPS_PSHOCK_ENTROPY` in Appendix B.3). The first criterion acts as shock detector. The implementation can be found in the source file Src/flag_shock.c.

  Note that by enabling this selective update, neither the total energy nor the entropy will generally be conserved at the numerical level.

- *ALWAYS*
  the entropy equation is used everywhere in the computational domain to update the solution array, i.e., $\mathcal{F}_\sigma = 1$ always in Eq. (2.5). This choice is consistent only with smooth flows.

- *CHOMBO_REGRID*
  the energy equation is used everywhere in the computational domain, that is, $\mathcal{F}_\sigma = 0$ in Eq. (2.5). However, pressure is still computed from entropy after projection, coarse-to-fine prolongation and restriction operations (AMR only). This option violates energy conservation but 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.

The `ENTROPY_SWITCH` is also compatible with super-time-stepping although it will only be used during the hydro steps.

2.2.5 HALL_MHD

This option is available only for the *MHD* module. It enables Hall MHD terms, see §8.1 and the available options are

- *NO*: Hall effects are not not included;
- *EXPLICIT*: Hall effects are included using explicit time stepping, see §8.5.1;

2.2.6 RESISTIVITY

Include resistive terms in the MHD equations, see §8.2. The available options are

- *NO*: resistivity is not included;
- *EXPLICIT*: resistivity is included explicitly, §8.5.1;
- *SUPER_TIME_STEPPING*: resistivity is treated using super-time-stepping, §8.5.2.
- *RK_LEGENDRE*: resistivity is treated using RK-Legendre, §8.5.3.

2.2.7 ROTATING_FRAME

When set to *YES*, it 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 [Kle98, MFS+12]. For example, in polar geometry, we solve

$$\begin{aligned}
&\frac{\partial}{\partial t}(\rho v_R) + \nabla\cdot(\rho v_R \boldsymbol{v}) + \frac{\partial p}{\partial R} &&= \frac{\rho(v_\phi + w_\phi)^2}{R} \\
&\frac{\partial}{\partial t}\left[\rho(v_\phi + w_\phi)\right] + \nabla^R\cdot\left[\rho(v_\phi + w_\phi)\boldsymbol{v}\right] + \frac{1}{R}\frac{\partial p}{\partial \phi} &&= 0 \\
&\frac{\partial}{\partial t}\left(E + \frac{w_\phi^2}{2}\rho + w_\phi\rho v_\phi\right) + \nabla\cdot\left[\boldsymbol{F}_E + pw\hat{\boldsymbol{\phi}} + \frac{w_\phi^2}{2}\rho\boldsymbol{v} + w_\phi\rho v_\phi\boldsymbol{v}\right] &&= 0
\end{aligned} \tag{2.7}$$

where $w_\phi = R\Omega_z$, 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. The additional terms under temporal derivatives are replaced, in the code, with the corresponding flux divergence terms only during the r sweep. The pressure term inside the divergence of the energy equation cancels with the $w\partial_\phi p/R$ term coming from the temporal derivative.

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

$$\frac{\rho(v_\phi + w_\phi)^2}{R} = \frac{\rho v_\phi^2}{R} + 2\rho v_\phi\Omega_z + \rho\Omega_z^2 R \tag{2.8}$$

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_Problems/HD/Disk_Planet.

2.2.8 THERMAL_CONDUCTION

Include thermal conduction effects, see §8.3. The available options are

- *NO*: thermal condution is not included;
- *EXPLICIT*: thermal conduction is treated explicitly, §8.5.1;
- *SUPER_TIME_STEPPING*: thermal conduction is treated using super-time-stepping, §8.5.2.
- *RK_LEGENDRE*: thermal conduction is treated using RK-Legendre, §8.5.3.

2.2.9 `VISCOSITY`

Include viscous terms, see §8.4. Options are

- *`NO`*: viscous terms are not included;
- *`EXPLICIT`*: viscosity is treated explicitly, §8.5.1;
- *`RK_LEGENDRE`*: viscosity is treated using RK-Legendre, §8.5.3.

See §8.4 for details.

2.3 User-defined Constants

In addition to the options described so far, the user can insert an arbitrary number of user-defined symbolic constants (macros) in the header file definitions.h. This should be done manually in the section delimited by

```
/* [Beg] user-defined constants (do not change this line) */


/* [End] user-defined constants (do not change this line) */
```

of this file. Only lines beginning with `#define` should appear in this section as they will not be changed by the python script. The value of a symbolic constant can be either a number or another symbolic constant previously defined by the code (e.g. `YES` or `NO`) and cannot be changed at runtime.

User-defined symbolic constants are useful in the following circumstances:

1. Conditional compilation: useful when your initial configuration contains computationally expensive code blocks that should be compiled separately. As an example, define (in your definitions.h) the symbolic constant name as `SETUP_VERSION` and give it the value of 0 or 1:

   ```
   /* [Beg] user-defined constants (do not change this line) */

   #define SETUP_VERSION    1

   /* [End] user-defined constants (do not change this line) */
   ```

   This symbolic macro name can then be used inside init.c (or any other source file) for conditional compilation:

   ```
   #if SETUP_VERSION == 0
    {
      /* implements version 0...  */
    }
   #elif SETUP_VERSION == 1
    {
      /* implements version 1... */
    }
   #endif
   ```

2. Advanced options (expert users): override the default value of special constant macros used throughout the code, a comprehensive list of which is given in Appendix B.3. This gives the user more control on the code and it avoids copying and modifying source files in the local working directory.

A simple example is given by configuration #04 in the Test_Problems/MHD/Rayleigh_Taylor/ problem where the symbolic constants

```
/* [Beg] user-defined constants (do not change this line) */

#define  USE_RANDOM_PERTURBATION   NO
#define  CHOMBO_REF_VAR            RHO

/* [End] user-defined constants (do not change this line) */
```

are be used in init.c to enable/disable a random perturbation and to tell **PLUTO**-Chombo that density must be used as the refinement variable. Another typical example is provided by the CGS physical units described §5.1.1.

3. Makefile Selection: makefile

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 parallel mode, support for HDF5 library and support for PNG library respectively 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, §13.2.

3.1 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)
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.

3.2 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`* inside your .defs file. If you do not have HDF5 installed, you may follow the installation guidelines given in §13.1.1. 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 §13.1.2.

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** uses the HDF5 1.6 API although it may be linked with HDF5 1.8.x without any problem since the `H5_USE_16_API` macro (defined in hdf5_io.c) forces the library to use HDF5 1.6 macro definitions.

3.3 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:

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

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

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

3.4 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.

4. Runtime 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 runtime information necessary for the computation. The initialization file controls several options used by the code at runtime, such as grid generation, CFL number, boundary conditions, output type and so forth. A template for this file can be found in the Src/Templates directory. The initialization file is divided into different "blocks" enclosed by a pair of square brackets $[\cdots]$. Each block contains a set of labels and corresponding mandatory or optional fields:

```
[Block]

label      ...  fields  ...
label      ...  fields  ...
label      ...  fields  ...
```

Tag labels on the left side identify appropriate field(s) following on the same line and must not be changed.

There're in total 9 blocks:

- *[Grid]*: controls grid generation;
- *[Chombo Refinement*: Chombo amr specific options;
- *[Time]*: time-stepping options;
- *[Solver]*: select the Riemann solver;
- *[Boundary]*: used to specify boundary conditions;
- *[Static Grid Output]*: controls output in the static grid version of **PLUTO** ;
- *[Chombo HDF5 output]*: controls output in the AMR version of **PLUTO** ;
- *[Particles]*: particle-related options (mainly output);
- *[Parameters]*: used to provide user-defined input parameters;

Block ordering is irrelevant. Runtime parameters can be accessed anywhere in the code through the members of the `Runtime` structure, (see Doc/Doxygen/html/structs_8h.html) using the function **`RuntimeGet()`**, e.g.

```
  cfl = RuntimeGet()->cfl;  /* Obtain the cfl number */
  char fname[64];
  sprintf (fname,"%s/mydata.dat",RuntimeGet()->output_dir);  /* Prepend output dir to file name */
```

The quantities (and related data-types) read from the file are now described.

4.1 The *[Grid]* block

```
[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
```

In the static version of **PLUTO** , it defines the physical domain and generates the whole computation mesh while in the AMR version is used to specify the base grid, corresponding to level 0.

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

For each dimension: the first (`int`) 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*) (*int*) (*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:

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

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

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

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.

In *CYLINDRICAL* or *SPHERICAL* coordinates, a radial logarithmic grid has the advantage of preserving the cell aspect ratio at any distance from the origin. In addition, the condition to obtain approximately squared cells (aspect ratio ≈ 1) is $\Delta r_1 \approx r_1 \Delta\phi$ where $\Delta r_1 = r_L\left(e^{\Delta\xi} - 1\right)$ is the radial spacing of the first active computational zone. This condition can be be used to determine either the number of points in the radial direction or the endpoint:

$$\log\frac{r_R}{r_L} = N_r \log\frac{2+\Delta\phi}{2-\Delta\phi} \,.$$

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

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 $0 \le x \le 3$. 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 $0 \le x \le 3$, and a stretched grid with 6 zones for $3 \le x \le 10$, see Fig.4.1

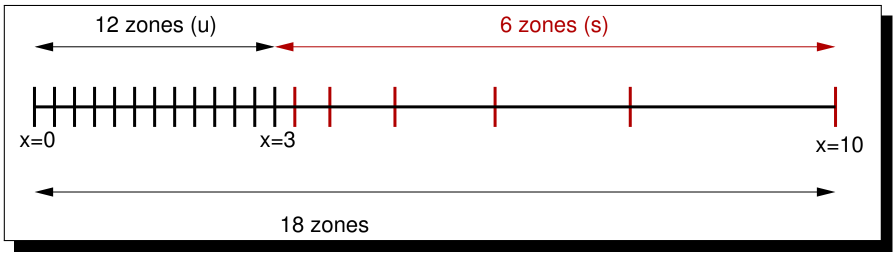


Figure 4.1: Example of one dimensional grid with a uniform (left) and stretched segment (right in red) covering the interval $[0, 10]$.

When the computational grid is generated, each processor owns a domain portion defined by the global integer variables `IBEG` $\le$ `i` $\le$ `IEND`, `JBEG` $\le$ `j` $\le$ `JEND` and `KBEG` $\le$ `k` $\le$ `KEND`. Ghost cells are added outside the local computational domain to complete the stencil at the boundaries, see Fig. 4.2. 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. 4.2.

However, the cell aspect ratio can be different from unity, that is, rectangular cells are allowed. The grid type *u*, *s* or *l±* is ignored and a uniform grid is always assumed unless the `CHOMBO_LOGR` switch is enabled to generate a logarithmic radial grid, see Appendix B.3. Cells must not necessarily have the same physical length in each direction (e.g., squares in 2D, cubes in 3D) and can have an aspect ratio different from 1. The refinement options are set in the `Chombo Refinement` section, §4.2.

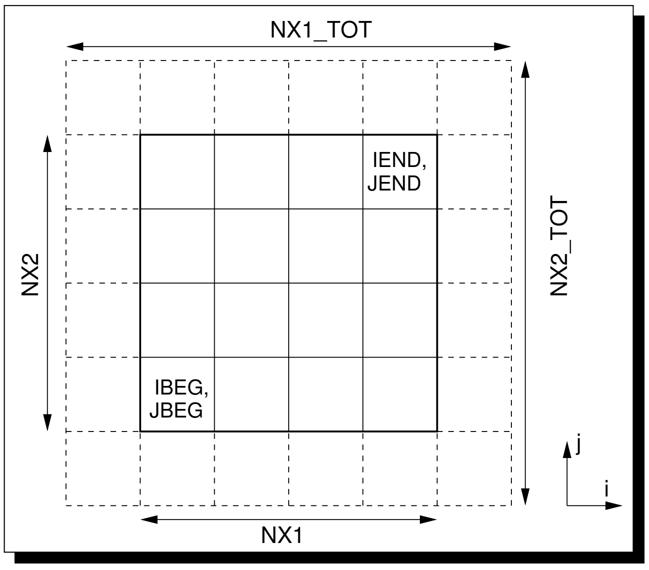


Figure 4.2: Computational grid in 2 dimensions with `NX1 = NX2 = 4` and 1 ghost zone. Internal zones (solid boxed) are spanned by IBEG $\leq i \leq$ IEND, JBEG $\leq j \leq$ JEND. Dashed boxes represent boundary ghost zones.

4.2 The *[Chombo Refinement]* Block

```
[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
```

Controls the grid refinement if **PLUTO** has been compiled with the Chombo library, §13. It is ignored otherwise. The relevant parameters for refinement are

- `Levels` (*int*)
  Sets 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` (*int*) (*int*) (...)
  Sets 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` (*int*) (*int*) (...)
  Sets 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*)
  Sets the threshold value χ_r above which cells are tagged for refinement during the grid generation process, see §13.3. When $\chi(U) > \chi_r =$ `Refine_thresh`, the cell is tagged to be refined.
- `Tag_buffer_size` (*int*)
  Sets the amount by which to grow tags (as a safety factor) before passing to MeshRefine.
- `Block_factor` (*int*)
  Sets the number of times that grids will be coarsenable by a factor of 2. A higher number produces ″blockier″ grids.
- `Max_grid_size` (*int*)
  Sets 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.

4.3 The *[Time]* Block

```
[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
```

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, §8.5.2), 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 (with STS), 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}.

	two_shock	roe	ausm+	hlld	hllc	hll	tvdlf
HD	√	√	√	-	√	√	√
MHD	-	√	-	√	√	√	√
RHD	√	-	-	-	√	√	√
RMHD	-	-	-	√	√	√	√

Table 4.1: Available Riemann solvers for the different physics module.

4.4 The *[Solver]* Block

```
[Solver]

Solver          tvdlf
```

- `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 [CW84] for the HD module, and [MB05] for the relativistic hydro equations;
  - *roe*: Linearized Roe Riemann solver based on characteristic decomposition of the Roe matrix, [Roe81].
  - *ausm+*: Advection Upstream Splitting Method of [Lio96] (only for the HD module);
  - *hlld*: The hlld approximate Riemann solver of [MK05] (for the adiabatic case), [Mig07] (for the isothermal case) and [MUB09] for the relativistic MHD equations;
  - *hllc*: Harten, Lax, Van Leer approximate Riemann Solver that restores with the middle contact discontinuity;
  - *hll*: Harten, Lax, Van Leer approximate Riemann Solver;
  - *tvdlf*: A simple Lax-Friedrichs scheme is used.

  Note that not all solvers are available for a given physics module, see Table 4.1. 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*, §B.3.

4.5 The *[Boundary]* Block

```
[Boundary]

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

Specifies the boundary conditions to be applied in the 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 the following:

- *`outflow`*
  Sets 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,

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

  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:

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

  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:

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

- *`periodic`*
  Sets periodic boundary conditions on both sides of the computational domain.

- *`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 §10.1) is enabled.

- *userdef*
  User-supplied boundary conditions (it requires coding your own boundary conditions in the function **UserDefBoundary()** in init.c, see §5.3).

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

4.6 The *[Static Grid Output]* Block

```
[Static Grid Output]

uservar     0
output_dir  ./                          # optional
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_dir     ./Log_Files                 # optional
log         1
analysis   -1.0  -1                     # optional
```

This block controls the output options in the static grid version of the code. Output files are written at specific times in a specific directory (local working directory by default) using different file formats, described in Chapter 12. The available fields are:

- `uservar` (*int*) (*string1 string2 ...*)
  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 12.

- `output_dir` (*string*)
  Sets the directory name for writing and reading simulation data. The directory must already exist at runtime or an error will occur. When writing, any of the data formats available below and the corresponding .out log files will be written in the directory specified by `output_dir`. In parallel, processor log files will also be written to the same directory unless `log_dir` (see below) is explicitly set to a different value. When reading (during restarts), .dbl or .dbl.h5 files and the corresponding *.out must be present in this directory or an error will occur.

  If `output_dir` is not specified, the directory from which pluto is executed is assumed.

- `dbl` (*double*) (*int/string*) (*string*)
  Assigns the output intervals for double precision (8 bytes) binary data. A negative value suppresses output with this format.

  – The first field (*double*) specifies the time interval (in code units) between consecutive outputs.
  – The second field can be an *int* 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).

  Double-precision format files can be used to restart the code using the `-restart n` command line argument.

- `flt` (*`double`*) (*`int/string`*) (*`string`*) [`cgs`]
  Like `dbl`, but for single-precision (4 bytes) data files. The last value (`cgs`) is optional and can be given to save datafiles directly in *cgs* physical units rather than in code units.
- `vtk` (*`double`*) (*`int/string`*) (*`string`*) [`cgs`]
  Like `flt`, but for VTK legacy file format, see §12.1.3.
- `dbl.h5` (*`double`*) (*`int/string`*)
  Like `dbl`, but for `hdf5` double-precision format §12.1.2. This format can also be used for restarting the code by supplying the `-h5restart n` command line argument.
- `flt.h5` (*`double`*) (*`int/string`*)
  Like `dbl` but for `hdf5` single-precision format §12.1.2;
- `tab` (*`double`*) (*`int/string`*)
  Sets the time and the number of steps interval for tabulated ascii format, §12.1.4;
- `ppm` (*`double`*) (*`int/string`*)
  Sets time and the number of step intervals for 2D color images in PPM format, §12.1.5;
- `png` (*`double`*) (*`int/string`*)
  Sets time and the number of step intervals for 2D color images in PNG format §12.1.5;
- `log_dir` (*`string`*)
  The name of the directory where output log files will be written. The directory must already be present at runtime. If not specified, the value of `output_dir` is used by default.
- `log` (*`int`*)
  Sets how often (in number of steps) the log file should be dumped to screen (in serial mode) or to disk (in parallel).
- `analysis` (*`double`*) (*`int`*)
  Sets time and number of steps interval between consecutive calls to the function **`Analysis()`**, see 5.5.

4.7 The *[Chombo HDF5 output]* Block

```
[Chombo HDF5 output]

Output_dir           ./            # optional
Checkpoint_interval  -1.0   0
Plot_interval         1.0   0
```

Relevant only for AMR-Pluto with the Chombo library (§13), this block controls how often restart and plot files are dumped to disk in the AMR version of the code. The relevant options are:

- `Output_dir` (*`string`*)
  Sets the output directory name for writing and reading simulation data. Both plotfiles and checkpoint files will be written to and read from this directory. If `Output_dir` is omitted, the current directory is used. Not to be confused with `output_dir` in §4.6.
- `Checkpoint_interval` (*`double`*) (*`int/string`*)
  Assigns the output interval(s) for checkpoint (restart) files.
  - The first field (*`double`*) can be used to set the time interval (or period) in code units.
  - The second field can be either an *`int`* or a *`string`*.
    An *`int`* value determines the number of steps between consecutive outputs.
    Alternatively, a *`string`* value can be used to set the wall-clock time between successive outputs: a value of *`3.55h`*, for instance, means that a checkpoint file is saved every 3 hours and 55 minutes.

Checkpoints files contain conservative variables.

- `Plot_interval` (*double*) (*int*)
  Sets the output frequency for plot (data) files. The meaning of the fields is the same used for `Checkpoint_interval` except that no wall-clock interval is permitted. Output files are stored using the HDF5 file format and numbered as data.nnnn.hdf5 where `n` is a zero-padded, sequentially increasing integer. Data files contain primitive variables.

Note that a negative number means that checkpoint- or plot-files are never written, 0 means that checkpoint files are written before the initial timestep and after the final one.

4.8 The *[Particles]* Block

```
[Particles]

Nparticles          8    -1
particles_dbl      -1.0  5.59h
particles_flt      -0.4  -1
particles_vtk       15.0  -1
particles_tab      -1.0  -1
```

The particles block is used to provide relevant information for the particle module (see Chapter 11) used at runtime. Additional information describing the datafile formats are given in §11.4. The available fields, mostly controlling output, are:

- `Nparticles` (*int*) (*int*)
  Specifies the number of particles used at initialization.
  - When positive, the first integer sets the total number of particles in the global computational domain;
  - When positive, the second integer specifies the number of particles per cell.

  More details can be found in §11.1.1. Note that only one of the two initialization is possible and thus one of the two integers must be always negative.

- `particles_dbl` (*double*) (*int/string*)
  Controls output intervals for double precision particle datafiles. The two fields have the same meaning used in the *[Static Grid Output]* block:
  - 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 5.50h tells PLUTO to write one .dbl file every five hours and 50 minutes.

  Negative values will suppress the option.

- `particles_flt` (*double*) (*int/string*)
  Same as `dbl`, but for single precision data files.

- `particles_vtk` (*double*) (*int/string*)
  Same as `dbl`, but for VTK legacy format.

- `particles_tab` (*double*) (*int/string*)
  Same as `dbl`, but for tabulated (ASCII) format (serial only).

4.9 The *[Parameters]* Block

```
[Parameters]

SCRH   0
```

• 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

5. Initial and Boundary Conditions

The source file init.c provides a set of functions that are used to set the fluid configuration for your own specific problem. These include:

- **`Init()`**: sets (fluid) initial conditions locally, as functions of the spatial coordinates x_1, x_2, x_3;
- **`InitDomain()`**: sets initial conditions by looping over the computational domain;
- **`UserDefBoundary()`**: sets user-defined boundary conditions at the physical boundary sides of your computational domain if necessary;
- **`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.

The init.c must be part of your local working directory and a template can be found in Src/Templates/init.c. Functions are described in the next sections. Note that, if you're using the particle module, initial conditions must be supplied using a different function, see Chapter 11.

5.1 Inital Conditions: the `Init()` function

The **`Init()`** function is used to assign the initial condition as a function of the spatial coordinates.

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` should 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 5.1.

  Temperature (in Kelvin) can also be used to initialize density or pressure see §5.1.2.

Note: PLUTO 3 Users: The old array indexing style using `DN`, `VX`, `VY` and `VZ`, etc... used in previous versions of the code is no longer supported.

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

Example #1:

The following code sets a disk with radius 1 centered around the origin in a 2D Cartesian domain. The flow is stationary and 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;
  }
}
```

With a small modification, the same initial condition can be written in a dimension-independent way (e.g. a line in 1D, a disk in 2D and a sphere in 3D):

```
void Init (double *v, double x1, double x2, double x3)
{
  double r2,r;

  r2 = D_EXPAND(x1*x1, + x2*x2, + x3*x3);
  r  = sqrt(r2);
  EXPAND(v[VX1] = 0.0;  ,
         v[VX2] = 0.0;  ,
         v[VX3] = 0.0;)

  if (r < 1.0){
    v[RHO] = 10.0;
    v[PRS] = 30.0;
  }else{
    v[RHO] = 1.0;
    v[PRS] = 1.0;
  }
}
```

The macro **`D_EXPAND(a,b,c)`** is used to write dimension-independent code by conditionally compiling one, two or three comma-separated arguments on the value taken by `DIMENSIONS`. Likewise, the macro **`EXPAND(a,b,c)`** is used to write component-independent code depending on the value taken by `COMPONENTS`. Function-like macro are documented in the file macro.h of the API reference guide, see ./Doc/Doxygen/html/macros_8h.html.

Index	Cylindrical	Polar	Spherical	Quantity	Physics Module
RHO	-	-	-	(rest-mass) 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 5.1: Array indices used for labeling primitive variables. Staggered components ("s" suffix) are used only for magnetic fields *in the boundary conditions*, see §6.2.3.3.

5.1.1 Units and Dimensions

In general, **PLUTO** works with non-dimensional (or "code") units so that flow 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), we strongly recommend to scale all quantities to non-dimensional units, in order to avoid occurrences of extremely small ($\lesssim 10^{-9}$) or large ($\gtrsim 10^{12}$) numbers that may be misinterpreted by numerical algorithms.

For simple adiabatic simulations involving no source term, the dimensionalization process can be avoided since the HD or MHD equations are scale invariant. However, dimensionalization is strictly necessary when specific length, time or energy scales are introduced in the problem and they must compare to the dynamical advection scales. For such problems, **PLUTO** requires three fundamental units to be specified through the definitions of the following symbolic constants:

`UNIT_DENSITY`	(ρ_0)	:	sets the reference density in gr/cm^3;
`UNIT_LENGTH`	(L_0)	:	sets the reference length in cm;
`UNIT_VELOCITY`	(v_0)	:	sets the reference velocity in cm/s.

All other units are derived from a combination of the previous three: time is measured in units of $t_0 = L_0/v_0$, pressure in units of $p_0 = \rho_0 v_0^2$, while magnetic field (for the `MHD` module only, see §6.2) in units of $B_0 = v_0\sqrt{4\pi\rho_0}$, i.e.:

$$\rho = \frac{\rho_{cgs}}{\rho_0} \quad , \quad \boldsymbol{v} = \frac{\boldsymbol{v}_{cgs}}{v_0} \quad , \quad p = \frac{p_{cgs}}{\rho_0 v_0^2} \quad , \quad \boldsymbol{B} = \frac{\boldsymbol{B}_{cgs}}{\sqrt{4\pi\rho_0 v_0^2}} \,. \tag{5.1}$$

where ρ, $\boldsymbol{v}$, p and $\boldsymbol{B}$ are now dimensionless.

Note: The dimensionless form of the equations can be derived by factoring out ρ_0, v_0, L_0 (and t_0) from the original equations written in c.g.s units. For instance, starting from the HD equation in presence of thermal conduction and cooling one has to write:

$$\begin{array}{llll}
\frac{\rho_0}{t_0}\frac{\partial\rho}{\partial t} & + & \frac{\rho_0 v_0}{L_0}\nabla\cdot(\rho\boldsymbol{v}) & = 0 \\
\frac{\rho_0 v_0}{t_0}\frac{\partial\boldsymbol{m}}{\partial t} & + & \frac{\rho_0 v_0^2}{L_0}\nabla\cdot(\boldsymbol{m}\boldsymbol{v}) + \frac{\rho_0 v_0^2}{L_0}\nabla p & = \frac{\rho_0}{L_0}\left(-\rho\nabla\Phi_{\rm cgs}\right) + \rho_0\rho\boldsymbol{g}_{\rm cgs} \\
\frac{\rho_0 v_0^2}{t_0}\frac{\partial(E+\rho\Phi)}{\partial t} & + & \frac{\rho_0 v_0^3}{L_0}\nabla\cdot\left[\left(\frac{\rho\boldsymbol{v}^2}{2}+\rho e + p + \rho\Phi\right)\boldsymbol{v}\right] - \frac{1}{L_0^2}\nabla\cdot\left(\kappa_{\rm cgs}\nabla T_{\rm cgs}\right) & = \rho_0 v_0\boldsymbol{m}\cdot\boldsymbol{g}_{\rm cgs} - \Lambda_{\rm cgs}
\end{array} \tag{5.2}$$

Since $t_0 = L_0/v_0$ and $T_{\rm cgs} = \mathcal{K}\mu p/\rho$ is the temperature express in Kelvin (see §5.1.2), straightforward manipulation yields the dimensionless form of the equations:

$$\begin{array}{llll}
\frac{\partial\rho}{\partial t} & + & \nabla\cdot(\rho\boldsymbol{v}) & = 0 \\
\frac{\partial\boldsymbol{m}}{\partial t} & + & \nabla\cdot(\boldsymbol{m}\boldsymbol{v}) + \nabla p & = -\rho\nabla\Phi + \rho\boldsymbol{g} \\
\frac{\partial(E+\rho\Phi)}{\partial t} & + & \nabla\cdot\left[\left(\frac{\rho\boldsymbol{v}^2}{2}+\rho e+p+\rho\Phi\right)\boldsymbol{v}\right] - \nabla\cdot\left[\kappa\nabla\left(\frac{\mu p}{\rho}\right)\right] & = \boldsymbol{m}\cdot\boldsymbol{g} - \frac{L_0}{\rho_0 v_0^3}\Lambda_{\rm cgs}
\end{array} \tag{5.3}$$

where $\Phi = \Phi_{\rm cgs}/v_0^2$, $\boldsymbol{g} = \boldsymbol{g}_{\rm cgs}L_0/v_0^2$, and

$$\kappa = \kappa_{\rm cgs}\frac{m_u}{\rho_0 v_0 L_0 k_B} \tag{5.4}$$

is the dimensionless form of the conduction coefficient.

If not specified, the default values of `UNIT_DENSITY`, `UNIT_LENGTH`, `UNIT_VELOCITY` are, respectively, $\rho_0 = 1\ m_p$ gr/cm^3, $L_0 = 1$ AU and $v_0 = 1$ Km/s. The values of the three fundamental units can be changed by redefining them in your definitions.h, e.g.,

```
/* [Beg] user-defined constants (do not change this line) */

#define  UNIT_DENSITY            1.67e-23
#define  UNIT_LENGTH             3.1e18
#define  UNIT_VELOCITY           1.e5

/* [End] user-defined constants (do not change this line) */
```

Note that, when the relativistic modules are used, v_0 must be the speed of light.
Output files can be directly saved in cgs units using the .flt or .vtk data format, see §4.6.

Example #2:

Consider a flow with typical number densities of the order of $n \approx 10\ \ \mathrm{cm}^{-3}$, temperatures $T \approx 10^4$ K (corresponding to typical sound speeds of $c_{s0} \approx 10$ Km/s). Suppose, also, that the flow propagates 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. Then one may choose

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

From the python script, this is done by including the following line in definitions.h:

```
/* [Beg] user-defined constants (do not change this line) */

#define  UNIT_DENSITY            (10.0*CONST_mp)
#define  UNIT_LENGTH             CONST_pc
#define  UNIT_VELOCITY           1.e5

/* [End] user-defined constants (do not change this line) */
```

where *CONST_mp* and *CONST_pc* are pre-defined symbolic constants (proton mass and parsec in c.g.s units) and are defined, together with several other constants, in Appendix B.1. Please remember using parenthesis around a macro expression to avoid incorrect expansion.

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

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

  /* -- define pressure to that sound speed = 1 * 1.e6 = 10 Km/sec  --  */

    v[PRS] = v[RHO]*cs*cs/g_gamma;  /* means 100/gmm * [\rho_0*v_0^2] */
```

where *CONST_PI* ($= \pi$) is another pre-defined constant. With this initialization, the sound speed is exactly $c_s = 10$ Km/s.

5.1.2 Specifying Temperature and Gas Composition.

In many applications, it may be more convenient to use a reference temperature to initialize pressure or velocity. In **PLUTO** , a direct relation between pressure and density (in "code", or non-dimensional units) and temperature (in Kelvin) is provided by

$$T = \frac{p}{\rho}\frac{\mu m_u v_0^2}{k_B} \equiv \frac{p}{\rho}\mathcal{K}\mu\,, \qquad (5.5)$$

where k_B is the Boltzmann constant, m_u is the atomic mass unit, μ is the mean molecular weight while p and ρ are in "code" (i.e. non-dimensional) units. The conversion factor $\mathcal{K}$ depends on UNIT_VELOCITY and it is provided by the macro **KELVIN** :

$$\mathcal{K} \equiv \frac{m_u v_0^2}{k_B} \qquad (5.6)$$

see also Chapter 7.
Eq. (5.5) can be easily used to determine pressure once the temperature is known, for instance:

```
v[RHO] = 0.5;          /* Density in code units  */
T      = 1.e3;         /* Temperature in KELVIN */
mu     = MeanMolecularWeight();
v[PRS] = v[RHO]*T/(KELVIN*mu);   /* Obtain pressure in code units */
```

where μ is the mean molecular weight:

$$\rho = \mu n_{\rm tot} m_u \tag{5.7}$$

while $n_{\rm tot}$ is the total number density of particle and it depends, in general, on the composition of the gas. The mean molecular weight may be computed:

- by calling the **MeanMolecularWeight()** function when the gas composition does not explicitly depend on temperature and density, e.g.

  ```
  mu = MeanMolecularWeight(v);
  ```

  where v is an array of primitive variables of which only ion fractions need to be defined (density and pressure are ignored). The mean molecular weight is implemented in the source file mean_mol_weight.c for a variety of different cooling/reaction network (Ch. 9) and also when no cooling is used.

  For the *SNEq* cooling module, for instance, the mean molecular weight is computed as

  $$\mu = \frac{A_H + A_{He}f_{He} + A_Z f_Z}{2 - f_{HI} + f_{He} + 2f_Z} \tag{5.8}$$

  where each metal contributes for one electron. In the previous equation, A_H (CONST_AH) and A_{He} (CONST_He) are the atomic mass numbers of hydrogen and helium whereas f_{HI}, f_{He} and f_Z are the number fractions of neutral hydrogen helium and metals with respect to *hydrogen*:

  $$f_{HI} = \frac{N_{HI}}{N_H}\,, \qquad f_{He} = \frac{N_{He}}{N_H} = \frac{Y}{A_{He}}\frac{A_H}{X}\,, \qquad f_Z = \frac{N_Z}{N_H} = \frac{Z}{A_Z}\frac{A_H}{X}\,,$$

  where $X = m_u n_H A_H/\rho$, $Y = m_u n_{He} A_{He}/\rho$ while $Z = 1 - X - Y$. Notice that while f_{He} and f_Z are fixed, f_{HI} is a time-dependent quantity that evolves with flow variables.

  Without any cooling / network, the mean molecular weight is computed from the given mass fractions assuming a fully ionized gas ($f_{HI} = 0$):

  $$\mu = \frac{A_H + A_{He}f_{He} + A_Z f_Z}{2 + f_{He} + f_Z(1 + A_Z/2)}\,,$$

  where $A_Z N_Z/2$ is the number of electrons due to metals.

  The value of X and Y can be specified through the user-defined constants H_MASS_FRAC and He_MASS_FRAC which, by default, are set to be equal to solar abundances ($X = 0.711$, $Y = 0.2741$, see also Appendix B.3).

- by calling the **GetMu()** function when the *PVTE_LAW* equation of state is adopted with equilibrum ionization, see §7.3. In this case temperature (in Kelvin) and density (in code units) must be supplied as input arguments:

  ```
  T   = 2.5e3;   /* In Kelvin */
  rho = 1.0;     /* In code units. Means rho*UNIT_DENSITY in cgs units */
  GetMu(T, rho, &mu);
  ```

Example #3:

Consider a flow with typical number densities of the order of $n \approx 4\times10^3$ cm^{-3}, temperature $T = 2.5\times10^3$ K and Mach number $M = v/c_s = 15$ while the typical length scale is $L_0 \approx 10$ AU. Suppose also that a magnetic field with strength of 10 μG is also present. Units can be set in definitions.h:

```
/* [Beg] user-defined constants (do not change this line) */

#define  UNIT_DENSITY            (1.e3*CONST_mp)
#define  UNIT_LENGTH             (10.0*CONST_au)
#define  UNIT_VELOCITY           1.e5

/* [End] user-defined constants (do not change this line) */
```

The sound speed c_s is defined, for an adiabatic equation of state, by the relation

$$c_s = \sqrt{\frac{\Gamma p}{\rho}} = \sqrt{\frac{\Gamma T}{\mathcal{K}\mu}} .$$

The initial condition is then implemented as follows:

```
    v[RHO] = 4.0;                  /* means  4 * [10^3 * 1.67e-24  gr/cm^3] */
    T      = 2.5e3;                /* Temperature in Kelvin */
    #if COOLING == SNEq
     CompEquil (n, T, v);          /* Compute ionization fraction at equilibrium */
    #endif
    mu = MeanMolecularWeight(v);
    v[PRS] = v[RHO]*T/(KELVIN*mu); /* Pressure in units of rho0*v0^2 */

  /* -- Define sound speed and velocity --  */

    cs     = sqrt(g_gamma*v[PRS]/v[RHO]); /* in units of [1 Km/s]  */
    v[VX1] = M*cs;                        /* in units of 15*cs*[1 Km/sec] */

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

    v[BX1] = 1.e-5/sqrt(4.0*CONST_PI*UNIT_DENSITY)/UNIT_VELOCITY;
```

The **CompEquil()** function is not strictly necessary but it has been introduced to compute ionization equilibrium values for a given reference temperature and number density. Its implementation may differ depending on the cooling module. In alternative, the fraction of neutrals could have been specified directly, .e.g, `v[X_HI] = 0.6;`.

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  */
v[BX1] = sqrt(2.0*v[PRS]/beta); /* in units of v_0\sqrt{4\pi\rho_0}  */
```

5.2 Initial Conditions: the `InitDomain()` function

The **`InitDomain()`** function has been introduced in **PLUTO** 4.3 to provide an alternative and more flexible way to provide initial conditions using a multidimensional loop. It is called after the **`Init()`** function to fill the values of

- the cell-centered primitive variable array `d->Vc[nv][k][j][i]`;
- the staggered magnetic field array `d->Vs[nv][k][j][i]`, at cell faces);
- the vector potential arrays `d->Ax1[k][j][i]`, `d->Ax2[k][j][i]` and `d->Ax3[nv][k][j][i]` at cell edges.

Location inside the cell (centered, face-centered or edge-centered) are shown in Fig. 5.1. Values given here will overwrite any previous assignment.

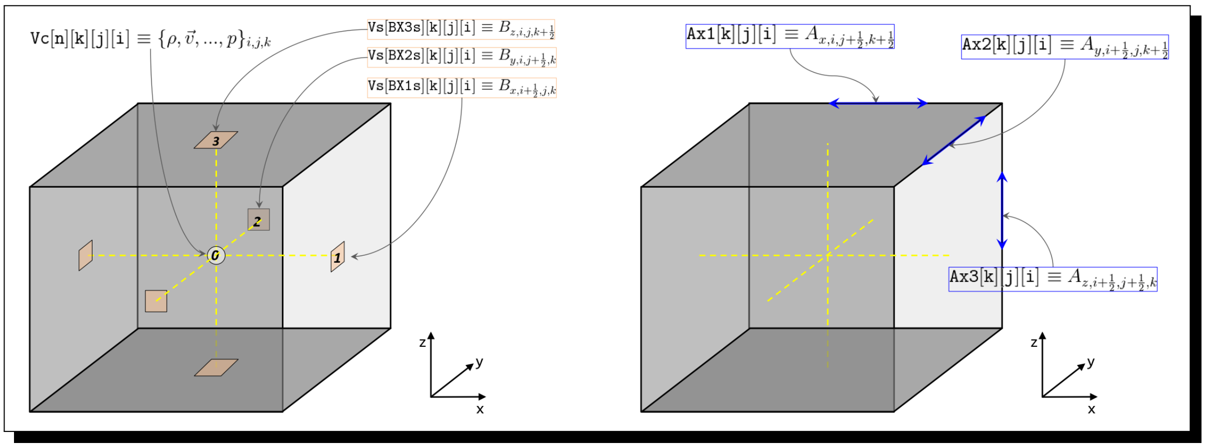

Vc[n][k][j][i] $\equiv \{\rho, \vec{v}, ..., p\}_{i,j,k}$
Vs[BX3s][k][j][i] $\equiv B_{z,i,j,k+\frac{1}{2}}$
Vs[BX2s][k][j][i] $\equiv B_{y,i,j+\frac{1}{2},k}$
Vs[BX1s][k][j][i] $\equiv B_{x,i+\frac{1}{2},j,k}$
3
2
0
1
z
y
x
Ax1[k][j][i] $\equiv A_{x,i,j+\frac{1}{2},k+\frac{1}{2}}$
Ax2[k][j][i] $\equiv A_{y,i+\frac{1}{2},j,k+\frac{1}{2}}$
Ax3[k][j][i] $\equiv A_{z,i+\frac{1}{2},j+\frac{1}{2},k}$
z
y
x

Figure 5.1: Schematic representation of cell- and face-centered (left panel) and edge-centered (right panel) collocation of physical variables inside a 3D cell i, j, k. Here `Vc`, `Vs` and `Ax1` are members of the **PLUTO** `data` structure.

The **`InitDomain()`** is preferable to **`Init()`** when the value of a primitive quantities needs to be assigned using neighbour values (a derivative, for example) or when a variable assignment depends on the total integral of a previous one or when initial conditions are specified from an external file (§5.2.1).

Syntax:

```
void InitDomain (const Data *d, 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 5.1), 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 5.2. 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 §6.2.3.3 for more details.
  - `d->Ax1[k][j][i]`, `d->Ax2[k][j][i]`, `d->Ax3[k][j][i]` (centered or staggered MHD): three-index arrays containing the three components of the vector potential defined at zone edges, see Fig 5.1.

    Note that, in the static grid version of **PLUTO** , the vector potential must be always defined at cell-edges regardless of the discretization of the $\nabla \cdot \boldsymbol{B} = 0$ condition: for staggered MHD,

B will be computed by differentiating the vector potential at cell faces, while for cell-centered MHD (eight wave of GLM) ***B*** is obtained by differentiating the same vector potential at cell centers.

Important: Face-centered (staggered) magnetic fields, cell-centered fluid variables and vector potential are defined on different zone stencils, see Figure 5.1. The zone-centering and the corresponding index range is encoded in the `box` structure (see below).

- `*grid`: a pointer to the `Grid` structure containing all of the relevant grid information. See the code documentation for more details on the members of the `Grid` structure.

A typical example is:

```
void InitDomain (Data *d, Grid *grid)
{
  int i, j, k;
  int id;
  double *x1 = grid->x[IDIR];
  double *x2 = grid->x[JDIR];
  double *x3 = grid->x[KDIR];

  TOT_LOOP(k,j,i){
    d->Vc[RHO][k][j][i] = 1.0 + exp(-x1[i]*x1[i]);
  }
}
```

__Note__: With **PLUTO** -Chombo, the **`InitDomain()`** function cannot be used to assign the vector potential and magnetic field components should be assigned directly.

5.2.1 Assigning Initial Conditions from Input Files

It is possible to assign initial conditions from user-supplied binary data by providing i) one or more grid data files and the corresponding ii) raw binary files containing the variable(s) 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 implemented. Even different input data files with different size and geomery may be used. 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 **`InputDataOpen()`** which reads and stores input grid information such as size, number of variables, geometry and dimensions. This function should be called once per variable from your **`InitDomain()`** function:

```
id = InputDataOpen (data_fname, grid_fname, endianity, pos);
```

where the arguments are:

1. `data_fname` is a string giving the name of the input binary file; The file name must end with the extension `.dbl` or `.flt` for double or single precision, respectively.
2. `grid_fname` specifies the name of the input grid file;
3. `endianity` is a string specifying the endianity of the input data file. Either `"big"`, `"little"` or and empty string `"  "` to leave things unchanged.
4. `pos` an integer giving the position of the variable inside the input binary file.

The return argument (`id`) is a file handle that can be used later.

The input binary data file(s) should be written in binary format using either single or double precision with extensions ".flt" (for the former) or ".dbl" (for the latter). Both single and/or multiple data files are supported. If a single file containing more variables is supplied, variables should be stored sequentially; otherwise you can provide one file per variable. You may provide only some of the variables used by **PLUTO** and not necessarily all of them. The input grid file should be written using the same format employed by **PLUTO** , see §12.1.6. Each field must contain as many points as specified by the input grid file.

After initialization, interpolation can be done using, e.g.,

```
d->Vc[RHO][k][j][i] = InputDataInterpolate (id, x1[i], x2[j], x3[k]);
```

which will read input data (in chunks) and interpolate them using bi- or tri-linear interpolation at the desired coordinate location (`x1[i], x2[j], x3[j]`). Input data are read from disk in chunks of size `nx1, nx2, ID_NZ_MAX` where `ID_NZ_MAX` is an integer number (default is 4) which can be changed in your definitions.h (see B.3).

The function **`InputDataClose()`** is finally used to free the memory structure when not needed.

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

```
void InitDomain (Data *d, Grid *grid)
{
  int i, j, k, id;
  double *x1 = grid->x[IDIR];
  double *x2 = grid->x[JDIR];
  double *x3 = grid->x[KDIR];


  id = InputDataOpen("tmp/data.0010.flt","tmp/grid0.out","␣",0);
  TOT_LOOP(k,j,i){
    d->Vc[RHO][k][j][i] = InputDataInterpolate (id,x1[i],x2[j],x3[k]);
  }
  InputDataClose(id);

  id = InputDataOpen("tmp/data.0010.flt","tmp/grid0.out","␣",1);
  TOT_LOOP(k,j,i){
    d->Vc[VX1][k][j][i] = InputDataInterpolate (id,x1[i],x2[j],x3[k]);
  }
  InputDataClose(id);
}
```

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.

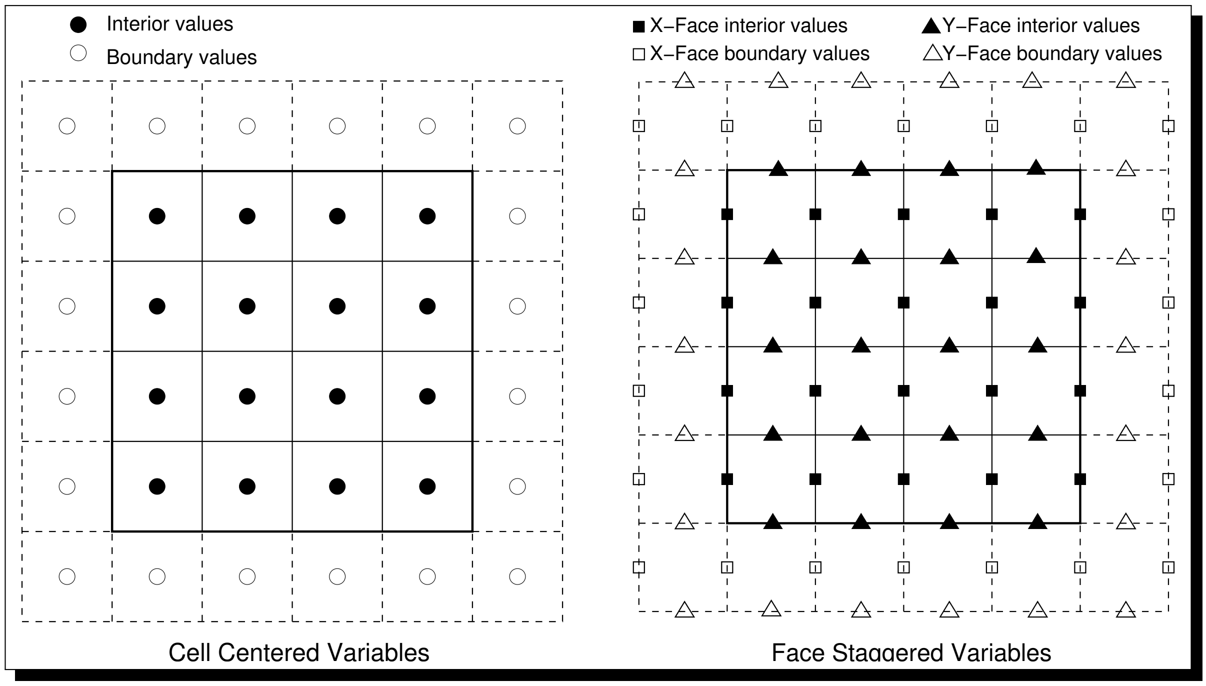


Figure 5.2: Schematic representation of cell-centered (left panel) and face-centered (right panel) collocation of physical variables on a 2D grid. X and Y-face centered staggered quantities are shown by squares and triangles, respectively. Filled symbols (circles, boxes and triangles) are considered 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**.

5.3 User-defined Boundary Conditions

The **`UserDefBoundary()`** function is used to assign user-defined boundary conditions to a particular physical boundary (see Fig 5.2) if this has 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...) by first enabling `INTERNAL_BOUNDARY` to *`YES`* inside definitions.h, see §5.3.1.

Syntax:

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

Arguments:

- `*d`: a pointer to the **PLUTO** data structure (see the description in **`InitDomain()`**

  All variables must be assigned at a user-defined boundary with the exception of the staggered component of magnetic field normal to the interface if you are using the Constrained Transport (CT) method, see §6.2.3.3.

- `*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 should be used 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 §5.3.1.

  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 the `Grid` structure containing all of the relevant grid information. See the code documentation for more details on the members of the `Grid` structure.

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 $z = 0$ boundary for $R \leq 1$ while reflective boundary conditions are assigned for $R > 1$. The inflow values are specified as

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

where M is the Mach number.

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

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

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 position (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.

Function-like macros are described in the code documentation: ./Doc/Doxygen/html/macros_8h.html

5.3.1 Internal Boundary

When **`UserDefBoundary()`** is called with `side==0` and `INTERNAL_BOUNDARY` has been turned to *`YES`* inside your definitions.h, the user has full control over the solution array. This feature can be used to adjust the value of selected 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[RHO][k][j][i] < 1.e-3) {
         d->Vc[RHO][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->xgc[IDIR];
  x2 = grid->xgc[JDIR];
  x3 = grid->xgc[KDIR];

  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.

5.4 Body Forces

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

$$\boldsymbol{a} = -\nabla\Phi + \boldsymbol{g}\,, \tag{5.9}$$

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. In this case, $\boldsymbol{g} = 0$ and 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. (6.1) and Eq. (6.4). The gravitational potential, however, must not change in time.

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

```
double BodyForcePotential(double x1, double x2, double x3)
{
  #if GEOMETRY == CARTESIAN
   return -1.0/sqrt(x1*x1 + x2*x2 + x3*x3);
  #elif GEOMETRY == CYLINDRICAL
   return -1.0/sqrt(x1*x1 + x2*x2);
  #elif GEOMETRY == SPHERICAL
   return -1.0/x1;
  #endif
}
```

for the three coordinate systems.

- The acceleration vector can be employed when the `BODY_FORCE` flag is set to *`VECTOR`* and the three components of ***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 ***g*** components along the coordinate directions;
  - `x1`, `x2`, `x3`: local zone coordinates.

  As an example, let's consider again a point-mass source located at the origin of coordinates. Then one needs to define, depending on the geometry (= *`CARTESIAN`*, *`CYLINDRICAL`* or *`SPHERICAL`*),

```
  void BodyForceVector(double *v, double *g, double x1, double x2, double x3)
  {
    double gs, rs;

    #if GEOMETRY == CARTESIAN
     rs = sqrt(x1*x1 + x2*x2 + x3*x3);  /* spherical radius in cart. coords */
    #elif GEOMETRY == CYLINDRICAL
     rs = sqrt(x1*x1 + x2*x2);          /* spherical radius in cyl. coords */
    #elif GEOMETRY == SPHERICAL
     rs = x1;                           /* spherical radius in sph. coords */
    #endif

    gs = -1.0/rs/rs;  /* spherical gravity  */

    #if GEOMETRY == CARTESIAN
     g[IDIR] = gs*x1/rs;
     g[JDIR] = gs*x2/rs;
     g[KDIR] = gs*x3/rs;
    #elif GEOMETRY == CYLINDRICAL
     g[IDIR] = gs*x1/rs;
     g[JDIR] = gs*x2/rs;
     g[KDIR] = 0.0;
    #elif GEOMETRY == SPHERICAL
     g[IDIR] = gs;
     g[JDIR] = 0.0;
     g[KDIR] = 0.0;
    #endif
  }
```

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)`*.

Beware that 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 §2.2.7.

A word of caution about using reflective, equatorial symmetric (or similar) boundary conditions: strictly speaking, gravity should be defined consistently with the antisymmmetric behavior of the velocity component normal to the given boundary plane. More precisely, the normal component of ***g*** should be antisymmetric while the potential should be an even function about the boundary plane. Consider, for instance, a reflective (or equatorial symmetric) conditions at the lower and upper boundaries z_b and z_e in the z direction. Then one should have:

$$\left\{\begin{array}{lcl} g_z(x,y,z_b-z) & = & -g_z(x,y,z_b+z) \\ \Phi(z,y,z_b-z) & = & \Phi(x,y,z_b+z) \end{array}\right. , \qquad \left\{\begin{array}{lcl} g_z(x,y,z_e+z) & = & -g_z(x,y,z_e-z) \\ \Phi(z,y,z_e+z) & = & \Phi(x,y,z_e-z) \end{array}\right.$$

If gravity does not satisfy this property then it must be imposed manually. As an example you can look at the Test_Problems/MHD/Rayleigh_Taylor or Test_Problems/MHD/Shearing_Box setups where reflective and equatorial symmetric boundaries are used in the y- and z- directions.

Note: **Relativistic flows**: Body forces are marginally compatible with the relativistic modules. Only *VECTOR* may be used.

5.5 The `Analysis()` function

The **`Analysis()`** function can be used to perform run-time data analysis/reduction in order to save intensive I/O for data post-processing. This function call frequency is set in pluto.ini, see §4.6.

Syntax:

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

Arguments:

- `*d` a pointer to the **PLUTO** data structure as in §5.3
- `*grid`: a pointer to the `Grid` structure containing all the relevant grid information.

Example:
In the next example we show how to compute, at run-time, the total integrated kinetic energy and the maximum internal energy:

$$\langle E_{\rm kin}\rangle \equiv \frac{1}{\Delta\mathcal{V}}\int\frac{1}{2}\rho\boldsymbol{v}^2dx\,dy\,dz = \frac{1}{\Delta\mathcal{V}}\sum_{i,j,k}\frac{1}{2}\rho_{i,j,k}\boldsymbol{v}^2_{i,j,k}\Delta\mathcal{V}_{i,j,k} \qquad (\rho e)_{\max} = \max_{i,j,k}\left(\frac{p}{\Gamma-1}\right)$$

where $\Delta\mathcal{V}$ is the total volume of the computational domain, $\Delta\mathcal{V}_{i,j,k} = \Delta x_i\Delta y_j\Delta z_k$ is the volume of a single cell, (i,j,k) extend over the entire computational domain (a Cartesian 3D domain is used for simplicity). The output file name is averages.dat and it is written as a 4-column tabulated ascii file containing the current integration time, the time step, the volume-integrated kinetic energy and maximum internal energy for the required time level. The example works also for parallel computations and can be safely used at restart since the last position of the file is automatically searched for and subsequent writing is appended starting from the correct row.

```
void Analysis (const Data *d, Grid *grid)
{
  int    i, j, k;
  double dV, vol, scrh;
  double Ekin, Eth_max, vx2, vy2, vz2;
  double *dx, *dy, *dz;

/* ---- Set pointer shortcuts ---- */

  dx = grid->dx[IDIR];
  dy = grid->dx[JDIR];
  dz = grid->dx[KDIR];

/* ---- Main loop ---- */

  Ekin = Eth_max = 0.0;
  DOM_LOOP(k,j,i){
    dV = dx[i]*dy[j]*dz[k];  /* Cell volume (Cartesian coordinates) */

    vx2 = d->Vc[VX1][k][j][i]*d->Vc[VX1][k][j][i];  /* x-velocity squared */
    vy2 = d->Vc[VX2][k][j][i]*d->Vc[VX2][k][j][i];  /* y-velocity squared */
    vz2 = d->Vc[VX3][k][j][i]*d->Vc[VX3][k][j][i];  /* z-velocity squared */

    scrh     = 0.5*d->Vc[RHO][k][j][i]*(vx2 + vy2 + vz2); /* cell kinetic energy */
    Ekin    += scrh*dV;

    scrh    = d->Vc[PRS][k][j][i]/(g_gamma - 1.0); /* cell internal energy */
    Eth_max = MAX(Eth_max, scrh);
  }

  vol  = g_domEnd[IDIR] - g_domBeg[IDIR];  /* Compute total domain volume */
  vol *= g_domEnd[JDIR] - g_domBeg[JDIR];
  vol *= g_domEnd[KDIR] - g_domBeg[KDIR];

  Ekin /= vol;   /* Compute kinetic energy average */

/* ---- Parallel data reduction ---- */

  #ifdef PARALLEL
   MPI_Allreduce (&Ekin, &scrh, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
   Ekin = scrh;

   MPI_Allreduce (&Eth_max, &scrh, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
   Eth_max = scrh;

   MPI_Barrier (MPI_COMM_WORLD);
  #endif

/* ---- Write ascii file "averages.dat" to disk ---- */

  if (prank == 0){
    char  fname[512];
    static double tpos = -1.0;
    FILE *fp;

    sprintf (fname, "%s/averages.dat",RuntimeGet()->output_dir);
    if (g_stepNumber == 0){  /* Open for writing only when we're starting */
      fp = fopen(fname,"w"); /*   from beginning */
      fprintf (fp,"#␣%7s␣␣%12s␣␣%12s␣␣%12s\n", "t", "dt", "<Ekin>","Max(Eth)");
    }else{                 /* Append if this is not step 0  */
      if (tpos < 0.0){  /* Obtain time coordinate of to last written row */
        char   sline[512];
        fp = fopen(fname,"r");
        while (fgets(sline, 512, fp))  {}
        sscanf(sline, "%lf\n",&tpos); /* tpos = time of the last written row */
        fclose(fp);
      }
      fp = fopen(fname,"a");
    }
    if (g_time > tpos){      /* Write if current time if > tpos */
      fprintf (fp, "%12.6e␣␣%12.6e␣␣%12.6e␣␣%12.6e␣\n",g_time, g_dt,Ekin, Eth_max);
    }
    fclose(fp);
  }
}
```

6. 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. 8 for a comprehensive description of the diffusion terms (thermal conduction, viscosity and magnetic resistivity) and cooling.

6.1 The HD Module

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

6.1.1 Equations

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

$$\frac{\partial}{\partial t}\left(\begin{array}{c}\rho\\ \boldsymbol{m}\\ E_t+\rho\Phi\end{array}\right)+\nabla\cdot\left(\begin{array}{c}\rho\boldsymbol{v}\\ \boldsymbol{m}\boldsymbol{v}+p\mathsf{I}\\ (E_t+p+\rho\Phi)\boldsymbol{v}\end{array}\right)^T=\left(\begin{array}{c}0\\ -\rho\nabla\Phi+\rho\boldsymbol{g}\\ \boldsymbol{m}\cdot\boldsymbol{g}\end{array}\right) \tag{6.1}$$

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_t is the total energy density:

$$E_t=\rho e+\frac{\boldsymbol{m}^2}{2\rho}\,. \tag{6.2}$$

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

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 $\boldsymbol{g}$ (§5.4).

The right hand side of the system of Eqns (6.1) is implemented in the **`RightHandSide()`** function inside Src/MHD/rhs.c[1] 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 $\boldsymbol{v}=\boldsymbol{m}/\rho$ while $p=p(\rho,\rho e)$ depends on the equation of state, see Chapter 7. 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:

$$\begin{array}{rcl}\dfrac{\partial\rho}{\partial t}+\boldsymbol{v}\cdot\nabla\rho+\rho\nabla\cdot\boldsymbol{v} &=& 0\\[2mm] \dfrac{\partial\boldsymbol{v}}{\partial t}+\boldsymbol{v}\cdot\nabla\boldsymbol{v}+\dfrac{\nabla p}{\rho} &=& -\nabla\Phi+\boldsymbol{g}\\[2mm] \dfrac{\partial p}{\partial t}+\boldsymbol{v}\cdot\nabla p+\rho c_s^2\nabla\cdot\boldsymbol{v} &=& 0\,,\end{array} \tag{6.3}$$

[1]HD and MHD share the same **`RightHandSide()`** function.

where $c_s = \sqrt{\Gamma p/\rho}$ is the adiabatic speed of sound for an ideal EOS. The quasi-linear form (6.3) is implemented in the Src/HD/prim_eqn.c source file and it is required during the predictor stages of the *HANCOCK* or *CHARACTERISTIC_TRACING* time-stepping schemes.

6.2 The MHD Module

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

6.2.1 Equations

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

$$\left\{\begin{array}{lcll}
\dfrac{\partial\rho}{\partial t} & + & \nabla\cdot(\rho\boldsymbol{v}) & = 0 \\[2ex]
\dfrac{\partial\boldsymbol{m}}{\partial t} & + & \nabla\cdot\left[\boldsymbol{m}\boldsymbol{v} - \boldsymbol{B}\boldsymbol{B} + \mathsf{I}\left(p + \dfrac{\boldsymbol{B}^2}{2}\right)\right]^T & = -\rho\nabla\Phi + \rho\boldsymbol{g} \\[2ex]
\dfrac{\partial\boldsymbol{B}}{\partial t} & + & \nabla\times(c\boldsymbol{E}) & = 0 \\[2ex]
\dfrac{\partial(E_t+\rho\Phi)}{\partial t} & + & \nabla\cdot\left[\left(\dfrac{\rho\boldsymbol{v}^2}{2} + \rho e + p + \rho\Phi\right)\boldsymbol{v} + c\boldsymbol{E}\times\boldsymbol{B}\right] & = \boldsymbol{m}\cdot\boldsymbol{g}
\end{array}\right. \tag{6.4}$$

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

$$E_t = \rho e + \frac{\boldsymbol{m}^2}{2\rho} + \frac{\boldsymbol{B}^2}{2}\,. \tag{6.5}$$

where an additional equation of state provides the closure $\rho e = \rho e(p,\rho)$ (see Chapter 7). 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 $\boldsymbol{g}$ (see §5.4).

In the third of Eq. (6.4), $\boldsymbol{E}$ is the electric field defined by the expression

$$c\boldsymbol{E} = -\boldsymbol{v}\times\boldsymbol{B} + \frac{\eta}{c}\cdot\boldsymbol{J} + \frac{\boldsymbol{J}}{ne}\times\boldsymbol{B} \qquad \Big(\boldsymbol{J} = c\nabla\times\boldsymbol{B}\Big) \tag{6.6}$$

where the first term is the convective term, the second term is the resistive term (η denotes the resistivity tensor, see §8.2) while the third term is the Hall term (§8.1). Note that the speed of light c never enters in the equations but we keep it for the sake of completeness. In the ideal case (only the first term in Eq. 6.6) is retained), the energy flux takes the form:

$$\frac{\partial(E_t+\rho\Phi)}{\partial t} + \nabla\cdot\left[(E_t + p_t + \rho\Phi)\boldsymbol{v} - \boldsymbol{B}\,(\boldsymbol{v}\cdot\boldsymbol{B})\right] = \boldsymbol{m}\cdot\boldsymbol{g}\,. \tag{6.7}$$

The hyperbolic contributions from the divergences terms on the left hans side of Eqns (6.4) are 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:

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

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

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

The primitive form of the equations, neglecting diffusion terms, is

$$
\begin{array}{rcl}
\dfrac{\partial\rho}{\partial t} + \boldsymbol{v}\cdot\nabla\rho + \rho\nabla\cdot\boldsymbol{v} & = & 0 \\
\dfrac{\partial\boldsymbol{v}}{\partial t} + \boldsymbol{v}\cdot\nabla\boldsymbol{v} + \dfrac{1}{\rho}\boldsymbol{B}\times(\nabla\times\boldsymbol{B}) + \dfrac{1}{\rho}\nabla p & = & -\nabla\Phi + \boldsymbol{g} \\
\dfrac{\partial\boldsymbol{B}}{\partial t} + \boldsymbol{B}(\nabla\cdot\boldsymbol{v}) - (\boldsymbol{B}\cdot\nabla)\boldsymbol{v} + (\boldsymbol{v}\cdot\nabla)\boldsymbol{B} & = & \boldsymbol{v}\,(\nabla\cdot\boldsymbol{B}) \\
\dfrac{\partial p}{\partial t} + \boldsymbol{v}\cdot\nabla p + \rho c_s^2\nabla\cdot\boldsymbol{v} & = & 0\,,
\end{array}
\tag{6.8}
$$

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.

6.2.2 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* (default) 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, §6.2.3).

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 and must be prescribed in the standard way, see the third example in Table 6.1.

Table 6.1 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	`v[BX1] = 0.0;` `v[BX2] = 5.0;` `v[BX3] = 0.0;`	`v[AX1] = 0.0;` `v[AX2] = 0.0;` `v[AX3] = -x1*5.0;`
$\boldsymbol{B} = (0,5,0)$ Cylindrical, 2D	`v[BX1] = 0.0;` `v[BX2] = 5.0;` `v[BX3] = 0.0;`	`v[AX1] = 0.0;` `v[AX2] = 0.0;` `v[AX3] = 0.5*x1*5.0;`
$\boldsymbol{B} = (-\sin y, \sin 2x, 2)$ Cartesian, 2.5D	`v[BX1] = -sin(x2);` `v[BX2] = sin(2.0*x1);` `v[BX3] = 2.0;`	`v[AX1] = 0.0;` `v[AX2] = 0.0;` `v[AX3] = cos(x2)+0.5*cos(2.0*x1);` `v[BX3] = 2.0;`

Table 6.1: 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).

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

6.2.3.1 Eight-Wave Formulation

In the eight-wave formalism [Pow94, PRL+99] the magnetic field has a cell-centered representation. In the ideal case, additional source terms are added on the right hand side of Eqns (6.4):

$$\frac{\partial}{\partial t}\left(\begin{array}{c}\rho\\ \boldsymbol{m}\\ E_t\\ \boldsymbol{B}\end{array}\right) + \nabla\cdot(...) = -\nabla\cdot\boldsymbol{B}\left(\begin{array}{c}0\\ \boldsymbol{B}\\ \boldsymbol{v}\cdot\boldsymbol{B}\\ \boldsymbol{v}\end{array}\right)$$

Contributions to $\nabla \cdot \boldsymbol{B}$ are taken direction by direction. Note that the 8-wave method 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 algorithm should be used in conjunction any Riemann solver with the exception of *hlld*.

6.2.3.2 Hyperbolic Divergence Cleaning

In [DKK+02] (see also [MT10, MTB10] 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

$$\left\{\begin{array}{lcl}\dfrac{\partial\boldsymbol{B}}{\partial t} + \nabla\times(c\boldsymbol{E}) + \nabla\psi & = & 0\\[2ex] \dfrac{\partial\psi}{\partial t} + c_h^2\nabla\cdot\boldsymbol{B} & = & -\dfrac{c_h^2}{c_p^2}\psi\end{array}\right. \tag{6.9}$$

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 controls the rate at which monopole are damped [MT10] and its value is set by the user-defined constant `GLM_ALPHA` (default value 0.1). 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, §12, 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 version, called the *extended GLM* formulation, that breaks momentum and energy conservation, has been found to be more robust in problems involving strongly magnetized media (see, for example, configuration # 11 in Test_Problems/MHD/Blast). The extended form of the equations [DKK+02, MT10] can be enabled by adding the user-defined constant `GLM_EXTENDED` to definitions.h and setting its value to *`YES`* from the Python script. For a complete description of the GLM- and Extended GLM-MHD formulation and its implementation in **PLUTO** refer to [MT10, MTB10].

6.2.3.3 Constrained Transport (CT)

In this formulation [BS99, Ld04, GS05], 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. 6.1. 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. 5.2. *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, §6.2.2.

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

$$\int\left(\frac{\partial \boldsymbol{b}}{\partial t} + \nabla\times\boldsymbol{E}\right)\cdot d\boldsymbol{S}_d = 0 \quad\Longrightarrow\quad \frac{db_{x_d}}{dt} + \frac{1}{S_d}\oint \boldsymbol{E}\cdot d\boldsymbol{l} = 0 \tag{6.10}$$

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

$$b_{x_1,i+\frac{1}{2},j,k} : \left\{\begin{array}{ll} \mathrm{IBEG}-1 & \le i \le \mathrm{IEND} \\ \mathrm{JBEG} & \le j \le \mathrm{JEND} \\ \mathrm{KBEG} & \le k \le \mathrm{KEND} \end{array}\right.$$

$$b_{x_2,i,j+\frac{1}{2},k} : \left\{\begin{array}{ll} \mathrm{IBEG} & \le i \le \mathrm{IEND} \\ \mathrm{JBEG}-1 & \le j \le \mathrm{JEND} \\ \mathrm{KBEG} & \le k \le \mathrm{KEND} \end{array}\right.$$

$$b_{x_3,i,j,k+\frac{1}{2}} : \left\{\begin{array}{ll} \mathrm{IBEG} & \le i \le \mathrm{IEND} \\ \mathrm{JBEG} & \le j \le \mathrm{JEND} \\ \mathrm{KBEG}-1 & \le k \le \mathrm{KEND} \end{array}\right.$$

Thus $b_{x_1,i+\frac{1}{2},j,k}$ is NOT a boundary value for $i = \mathrm{IBEG}-1$, $\mathrm{JBEG} \le j \le \mathrm{JEND}$, $\mathrm{KBEG} \le k \le \mathrm{KEND}$ 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) E is computed at zone edges, see Fig. 6.1 by a proper averaging/reconstruction scheme (set by `CT_EMF_AVERAGE` inside your definitions.h). Options are:

- `CT_EMF_AVERAGE` = *`ARITHMETIC`* yields a simple arithmetic averaging [BS99] of the fluxes computed during the upwind steps. In this case, one has available

$$\left(\begin{array}{c} 0 \\ -E_{x_3} \\ E_{x_2} \end{array}\right)_{i+\frac{1}{2},j,k} , \quad \left(\begin{array}{c} E_{x_3} \\ 0 \\ -E_{x_1} \end{array}\right)_{i,j+\frac{1}{2},k} , \quad \left(\begin{array}{c} -E_{x_2} \\ E_{x_1} \\ 0 \end{array}\right)_{i,j,k+\frac{1}{2}}$$

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

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

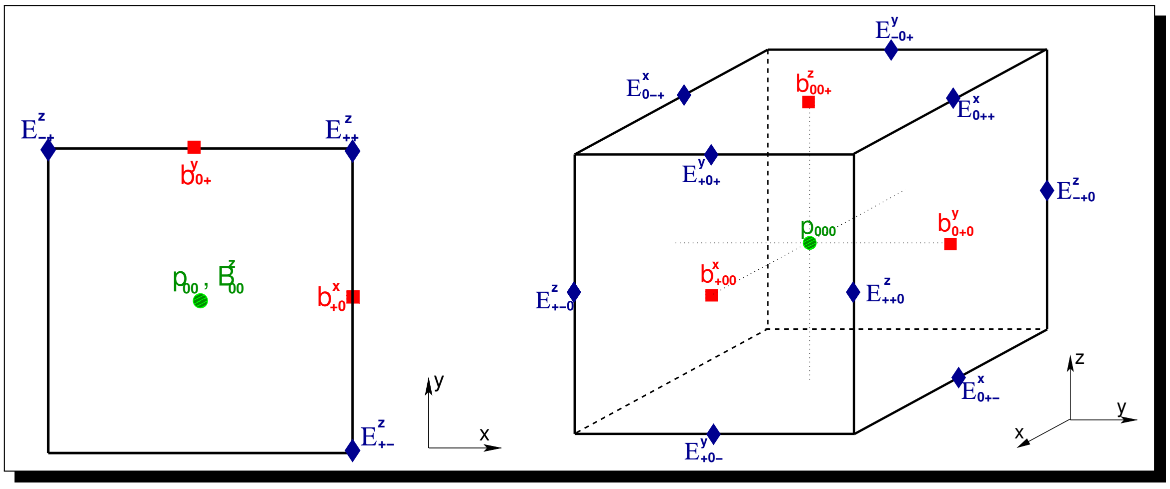


Figure 6.1: Collocation points in 2 D (left) and in 3D (right). Cell-centered quantities are given as green circles, face-centered quantities (magnetic field) as red squares and edge-centered values (electric field) as blue diamonds.

$$E_{x2,i+\frac{1}{2},j,k+\frac{1}{2}} = \frac{1}{4}\left(E_{x2,i+\frac{1}{2},j,k} + E_{x2,i+\frac{1}{2},j,k+1} + E_{x2,i,j,k+\frac{1}{2}} + E_{x2,i+1,j,k+\frac{1}{2}}\right)$$

$$E_{x3,i+\frac{1}{2},j+\frac{1}{2},k} = \frac{1}{4}\left(E_{x3,i+\frac{1}{2},j,k} + E_{x3,i+\frac{1}{2},j+1,k} + E_{x3,i,j+\frac{1}{2},k} + E_{x3,i+1,j+\frac{1}{2},k}\right)$$

Although being the simplest one, this average procedure may suffer from insufficient dissipation in some circumstances ([GS05, Ld04]) 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 [DBL03, Ld04]. 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 [GS05], 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:

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

Beware that the three staggered components have *different spatial locations* and the **`BOX_LOOP()`** macro introduced in §5.3 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 a MHD jet problem 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 $R \le 1$ while a symmetric counter-jet is assumed for $R > 1$:

```
if (side == X2_BEG){

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

6.2.4 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:

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

In this case one can show [Pow94] that the MHD equations reduce to:

$$\begin{array}{rcl} \dfrac{\partial\rho}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}) & = & 0 \\ \dfrac{\partial\boldsymbol{m}}{\partial t} + \nabla\cdot\Big(\boldsymbol{m}\boldsymbol{v} - \boldsymbol{B}_1\boldsymbol{B} - \boldsymbol{B}_0\boldsymbol{B}_1\Big) + \nabla p_t & = & \rho(-\nabla\Phi + \boldsymbol{g}) \\ \dfrac{\partial(E_{t1} + \rho\Phi)}{\partial t} + \nabla\cdot\Big[(E_{t1} + p_t + \rho\Phi)\boldsymbol{v} - (\boldsymbol{v}\cdot\boldsymbol{B}_1)\boldsymbol{B}\Big] & = & \boldsymbol{m}\cdot\boldsymbol{g} \\ \dfrac{\partial\boldsymbol{B}_1}{\partial t} - \nabla\times(\boldsymbol{v}\times\boldsymbol{B}) & = & 0 \end{array}$$

where

$$p_t = p + \frac{\boldsymbol{B}_1^2}{2} + \boldsymbol{B}_1\cdot\boldsymbol{B}_0\,, \quad E_{t1} = \frac{p}{\Gamma - 1} + \frac{1}{2}\left(\rho\boldsymbol{v}^2 + \boldsymbol{B}_1^2\right)$$

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 $\boldsymbol{B} \to \boldsymbol{B}_1$, $E \to E_{t1}$.

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)
```

Note that when writing output datafiles to disk, only the deviation $\boldsymbol{B}_1$ is written.

Examples can be found in the 4^{th} configuration of Test_Problems/MHD/Rotor/ and in the 4^{th} or 5^{th} configurations of Test_Problems/MHD/Blast/.

Note: Background field splitting works, at present, with the CT and GLM divergence cleaning techniques, with most Riemann solvers but only with RK-type integrators.

6.3 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 in Newtonian approximation can also be incorporated. The relevant source files and definitions for this module can be found in the Src/RHD directory.

6.3.1 Equations

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

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

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

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

where $\boldsymbol{v}$ is the velocity, p is the thermal pressure. Primitive variables $\boldsymbol{V}$ always include the rest-mass density ρ, three-velocity $\boldsymbol{v} = (v_{x1}, v_{x2}, v_{x3})$ and pressure p. The relation between $\boldsymbol{U}$ and $\boldsymbol{V}$ is more complicated and is expressed by

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

where h is the specific enthalpy (see Chapter 7 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:

$$\begin{array}{lcl}
\displaystyle\frac{\partial\rho}{\partial t} + \boldsymbol{v}\cdot\nabla\rho - \frac{1}{c_s^2 h}\boldsymbol{v}\cdot\nabla p & = & \displaystyle\frac{1}{c_s^2 h}\frac{\partial p}{\partial t} \\[2ex]
\displaystyle\frac{\partial\boldsymbol{v}}{\partial t} + \boldsymbol{v}\cdot\nabla\boldsymbol{v} + \frac{1}{\rho h\gamma^2}\nabla p & = & \displaystyle -\frac{\boldsymbol{v}}{\rho h\gamma^2}\frac{\partial p}{\partial t} + \boldsymbol{a} \\[2ex]
\displaystyle\frac{\partial p}{\partial t} + \frac{1}{1-\boldsymbol{v}^2c_s^2}\Big[c_s^2\rho h\nabla\cdot\boldsymbol{v} + (1-c_s^2)\boldsymbol{v}\cdot\nabla p\Big] & = & 0\,.
\end{array} \qquad (6.11)$$

For more detailed expressions and the characteristic decomposition, see [MPB05].

Spatial reconstruction may be performed on the four-velocity rather than on the three-velocity by enabling the macro `RECONSTRUCT_4VEL` to *YES* manually in your definitions.h (see also Appendix B.3). 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 $|\boldsymbol{v}| < 1$.

6.4 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.

6.4.1 Equations

The RMHD module solves the following system of conservation laws:

$$\frac{\partial}{\partial t}\begin{pmatrix} D \\ \boldsymbol{m} \\ E_t \\ \boldsymbol{B} \end{pmatrix} + \nabla\cdot\begin{pmatrix} D\boldsymbol{v} \\ w_t\gamma^2\boldsymbol{v}\boldsymbol{v} - \boldsymbol{b}\boldsymbol{b} + \mathsf{I}p_t \\ \boldsymbol{m} \\ \boldsymbol{v}\boldsymbol{B} - \boldsymbol{B}\boldsymbol{v} \end{pmatrix}^T = \boldsymbol{0} \tag{6.12}$$

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

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

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

$$M^{ij} = w_tu^iu^j - b^ib^j = m^iv^j - \frac{b^iB^j}{\gamma} = m^iv^j - \left(\frac{B^i}{\gamma^2} + v^i\boldsymbol{v}\cdot\boldsymbol{B}\right)B^j$$

Primitive variables are similar to the RHD module but they also contain the magnetic field, $\boldsymbol{V} = (\rho, \boldsymbol{v}, p, \boldsymbol{B})$. 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 *IDEAL* and *TAUB* already introduced for the RHD module (see [MM07] for the extension of this EOS to the RMHD equations).

The RMHD sub-menu offers some of the switches already discussed in the MHD module (§6.2) or in the RHD (§6.3) module. Divergence control is achieved using the same algorithms introduced for MHD, namely: 8-wave (§6.2.3.1), divergence cleaning (§6.2.3.2) and the constrained transport (§6.2.3.3).

Computation of the fast characteristic speeds can be perfomed by replacing the numerical solution of a quartic equation (see [MM07]) with the analytical solution of an approximate quadratic equtions thus making computation faster. This is achieved by setting `RMHD_FAST_EIGENVALUES` to *YES* (as in [DZBL07]), see the Appendix B.3.

7. Equation of State

In the current implementation, **PLUTO** describes a thermally ideal gas obeying the *thermal* Equation of State (EOS)

$$p = nk_BT = \frac{\rho}{m_u\mu}k_BT \tag{7.1}$$

where p is the pressure, n is the total particle number density, k_B is the Boltzmann constant, T is the temperature, ρ is the density, m_u is the atomic mass unit and μ is the mean molecular weight. The thermal EOS describes the thermodynamic state of a plasma in terms of its pressure p, density ρ, temperature T and chemical composition μ. Eq. (7.1) is written in CGS physical units. Using code units for p and ρ while leaving temperature in Kelvin, the thermal EOS is conveniently re-expressed as

$$p = \frac{\rho T}{\mathcal{K}\mu} \qquad \Longleftrightarrow \qquad T = \frac{p}{\rho}\mathcal{K}\mu \qquad \left(\text{where}\quad \mathcal{K} = \frac{m_u v_0^2}{k_B}\right) \tag{7.2}$$

where $\mathcal{K}$ is the **KELVIN** macro which depends explictly on the value of UNIT_VELOCITY.

Another fundamental quantity is the (specific) internal energy e whose rate of change under a physical process is regulated by the first law of thermodynamics:

$$de = dQ - pd\left(\frac{1}{\rho}\right) . \tag{7.3}$$

where Q represents the heat absorbed or released. The internal energy is a state function of the system and can also be related to temperature and density via the *caloric* equation of state [Tor97]

$$e = e(T,\rho) . \tag{7.4}$$

The *thermal* and *caloric* equations of state given by Eq. (7.2) and (7.4) constitutes the basis for the consideration discussed in the next sections.

7.1 The *ISOTHERMAL* Equation of State

In an isothermal gas, the temperature is constant and the pressure is readily obtained as

$$p = \rho c_{\rm iso}^2 \tag{7.5}$$

where $c_{\rm iso}$ (the isothermal sound speed) can be either a constant value or a spatially-varying quantity. This EOS is available only in the HD and MHD modules. No energy equation is present and the labels ENG and PRS are undefined.

The value of $c_{\rm iso}$ can be set using the global variable g_isoSoundSpeed in your init.c, e.g.

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

If not set, the default value is $c_{\rm iso} = 1$.

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.

7.2 The *IDEAL* Equation of State

For a calorically ideal gas, the ratio of specific heats Γ is constant and the internal energy can be written

$$\rho e = \frac{p}{\Gamma - 1} . \tag{7.6}$$

The value of Γ is stored in the global variable `g_gamma` and can be modified in your **`Init()`** function (default value $5/3$).

For a relativistic flow, the constant-Γ EOS is more conveniently expressed through the specific enthalpy:

$$\rho h = \rho + \frac{\Gamma}{\Gamma - 1} p\,. \tag{7.7}$$

The ideal EOS is compatible with all physics modules, algorithms and Riemann solvers in the code.

7.3 The *PVTE_LAW* Equation of State

The PVTE (Pressure-Volume-Temperature-Energy) EOS allows the user to specify the internal energy as a general function of the temperature T and chemical fractions (or concentrations) $\boldsymbol{X}$ as described in the paper by [VMBM15].

The thermal EOS (7.1) together with the caloric EOS (7.4) link the five quantities p, ρ, T, $\boldsymbol{X}$ and e and are used by the code to compute two of them given the remaining three:

$$\left\{\begin{array}{lcl} p & = & \dfrac{\rho}{\mu(\boldsymbol{X}) m_u} k_B T \\[2ex] e & = & e(T, \boldsymbol{X}) \end{array}\right. \tag{7.8}$$

where m_u is the atomic mass unit and $\mu(\boldsymbol{X})$, the mean molecular weight, depends on the gas composition. The *PVTE_LAW* EOS allows the user to provide explicit definitions for $\mu(\boldsymbol{X})$ and $e(T, \boldsymbol{X})$ in a thermodynamically consistent way[1].

The implementation of this EOS depends on how chemical fractions are computed and a major distinction should be made between non-equilibrium and equilibrium cases:

- *Non equilibrium case:* a chemical network is used to evolve $\boldsymbol{X}(t)$ through rate equations under non-equilibrium conditions, see §9. This occurs, for example, when this EOS is used in conjunction with a cooling module that includes a time-dependent reaction network. In this case, species are evolved independently and their value is at disposal when performing conversion between pressure, temperature and internal energy. In particular, recovering temperature from internal energy, $T = T(e, \boldsymbol{X})$ requires inverting a nonlinear equation by means of an iterative root finder.
- *Equilibrium case:* there's no chemical network and fractions are not evolved independently but are computed when necessary using some sort of equilibrium assumptions such as Saha (LTE, valid in the high density limit) or collisional-ionization equilibrium (CIE, valid at low densities). This corresponds to express fractions as $\boldsymbol{X} = \boldsymbol{X}(T, \rho)$ so that quantities depending on $\boldsymbol{X}$ become functions of (T, ρ). For example, the thermal EOS becomes $p = p(\rho, T)$ while internal energy becomes a function of two variables, $e = e(T, \rho)$. In this case, the inverse functions $T = T(p, \rho)$ and $T = T(e, \rho)$ are computed by finding the roots of nonlinear equations.

The implementation of the *PVTE_LAW* EOS can be found in the Src/EOS/PVTE directory. The source file pvte_law.c (or pvte_law_template.c if you are starting from scratch) provides the interface between the user implementation and the module through the following functions:

- **`InternalEnergyFunc()`**: compute and return the internal energy density ρe where $e \equiv e(T, \boldsymbol{X})$ in non-equilibrium chemistry or $e \equiv e(T, \rho)$ in the equilibrium case;
- **`GetMu()`**: compute the mean molecular weight $\mu = \mu(\boldsymbol{X})$ or $\mu(T, \rho)$.
- **`Gamma1()`**: compute the value of the first adiabatic index,

$$\Gamma_1 = \frac{1}{c_V}\frac{p}{\rho T}\chi_T^2 + \chi_\rho^2 \qquad \text{where} \qquad \left\{\begin{array}{l} \chi_T = \left(\dfrac{\partial \log p}{\partial \log T}\right)_\rho = 1 - \dfrac{\partial \log \mu}{\partial \log T} \\[3ex] \chi_\rho = \left(\dfrac{\partial \log p}{\partial \log \rho}\right)_T = 1 - \dfrac{\partial \log \mu}{\partial \log \rho} \end{array}\right. \tag{7.9}$$

[1]For a thermally ideal gas, it can be shown that the specific internal energy e is a function of the temperature T and chemical composition $\boldsymbol{X}$. Also, $e(T)$ must be monotonically increasing.

needed to evaluate the sound speed, $c_s = \sqrt{\Gamma_1 p/\rho}$. Note that Γ_1 has the upper bound of $5/3$ and may not be straightforward to compute. Fortunately, its value is only needed to estimate the wave propagation speed during the Riemann solver and an approximate value should suffice.

Two different implementations are provided with the current distribution: pvte_law_H+.c is suitable for a partially hydrogen gas in LTE (described in the next section) while pvte_law_dAngelo.c can be used for molecular and atomic hydrogen cooling as in D'Angelo, G. et al ApJ (2013) 778. More technical details can be found under the Src/EOS/PVTE folder in the API reference guide or following this link.

Note: The *PVTE_LAW* EOS is not compatible with algorithms requiring characteristic decomposition and cannot be used with the `ENTROPY_SWITCH`. We suggest to use RK time-stepping and the *tvdlf*, *hll* or *hllc* Riemann solvers. This EOS is, at present, available for the HD and MHD modules only.

7.3.1 Example: EOS for a Partially Ionized Hydrogen Gas in LTE

As a simple non-trivial example, consider a partially ionized hydrogen gas in Local Thermodynamic Equilibrium (LTE, no cooling), see also §2.4 of [VMBM15]. Let the particle number densities be

$$n_0\,(\text{neutrals})\,,\qquad n_p = n_e\,(\text{charge neutrality})\qquad \Longrightarrow\qquad n = n_0 + n_p + n_e = n_0 + 2n_p$$

Density and pressure can then be written as

$$\left\{\begin{array}{lcl} \rho &=& m_p n_p + (m_p + m_e)n_0 + m_e n_e \approx m_p(n_p + n_0) = \mu(2n_p + n_0)m_p \\ p &=& (n_e + n_p + n_0)k_B T = (1 + x)(n_p + n_0)kT = \dfrac{\rho k_B T}{\mu m_p} \end{array}\right.$$

where $\mu = 1/(1+x)$ is the mean molecular weight, $x = n_p/(n_p+n_0)$ is the degree of ionization computed from Saha equation:

$$\frac{x^2}{1-x} = \frac{(2\pi m_e k_B T)^{3/2}}{h^3(n_p + n_0)} e^{-\chi/(k_B T)}\,, \tag{7.10}$$

where $\chi = 13.6$ eV and $n_p + n_0 = \rho/m_p$.

The (specific) internal energy includes two contributions:

$$e = \frac{3}{2}\frac{k_B T}{\mu m_p} + \frac{\chi}{m_p}x = \frac{3}{2}\frac{p}{\rho} + \frac{\chi}{m_p}x \tag{7.11}$$

where the first one represents the standard kinetic energy while the second one corresponds to the ionization energy (neutral atoms have a potential energy that is lower than that of ions). Note that the latter introduces a temperature, or equivalently, a velocity scale in the problem so that computations are no longer scale-invariant but depend on the value of `UNIT_VELOCITY` (used to obtain the temperature in Kelvin) and `UNIT_DENSITY` (used in Saha equation) that must be defined in your **`Init()`** function, see §5.1.1. Fig. 7.1 shows the classical Sod shock tube solution at $t = 0.2$ obtained with the *IDEAL* equation of state and the *PVTE_LAW* with `UNIT_DENSITY`$= 10^5\,m_p$ and `UNIT_VELOCITY`$= 10$ Km/s. The equivalent Γ, defined as

$$\Gamma_{\rm eq} = \frac{p}{\rho e} + 1\,, \tag{7.12}$$

is no longer a constant but a function of the temperature, see Fig. 7.2.

The implementation of this particular EOS can be found in Src/EOS/PVTE/pvte_law_H+.c (simply copy it to your working directory as pvte_law.c). Eq. (7.11) is implemented by the **`InternalEnergyFunc()`** function while the mean molecular weights μ is defined by the **`GetMu()`** function.

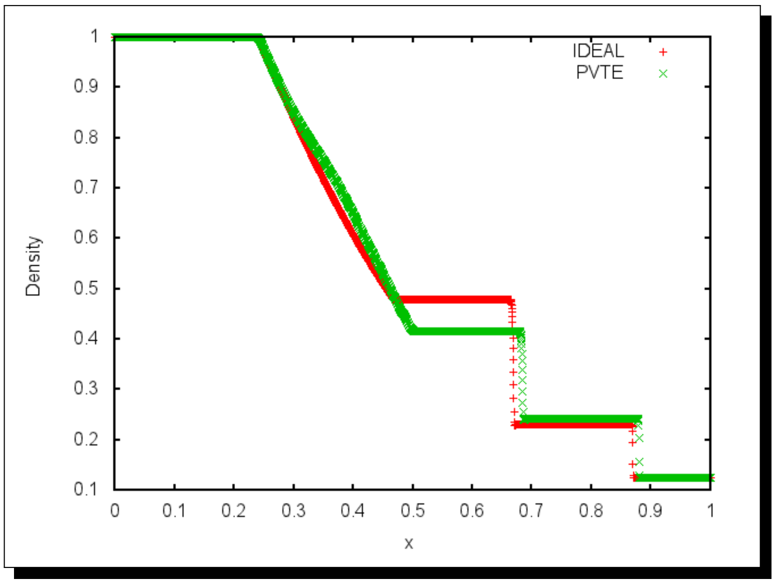


Figure 7.1: Density plot for the Sod shock tube test at $t = 0.2$ obtained with the *IDEAL* EOS (red) and the *PVTE_LAW* EOS (green) with reference density $10^5\,m_p$ and reference velocity 10^6 cm/s.

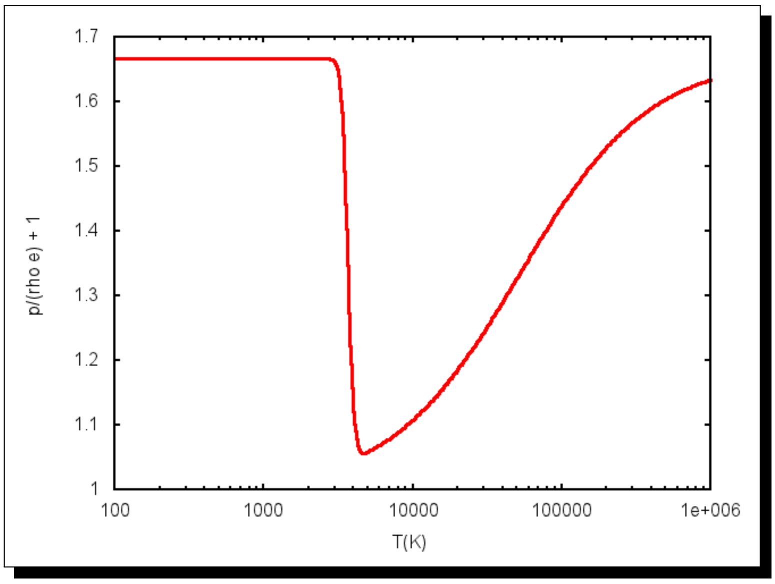


Figure 7.2: Equivalent $\Gamma = p/(\rho e) + 1$ for the *PVTE_LAW* EOS of a partially ionized hydrogen gas.

7.3.2 Analytic vs. tabulated approach

As **PLUTO** performs conversions between primitive (e.g. density and pressure) and conservative variables (e.g. total energy and momentum) during a single update step, the *PVTE_LAW* Eqns. (7.8) must often be inverted to obtain the temperature from pressure or internal energy. Since Eqns (7.8), specially the second one, can be nonlinear functions of T, the inversion must be taken numerically using a root finder and this can be an expensive task.

In the equilibrium case, however, a faster and often more convenient approach is to have **PLUTO** pre-compute tabulated versions of the EOS so as to replace expensive function evaluations with tables. In this case no root finder is used and computations involving EOS require (direct or inverse) simpler lookup table operations and cubic/linear interpolation (see §3.2.2 of [VMBM15]). This feature is always turned on by default but can be overridden through the user-defined constants `PV_TEMPERATURE_TABLE` and/or `TV_ENERGY_TABLE` in your definitions.h (see §2.3). For example, the following definitions

```
#define PV_TEMPERATURE_TABLE   YES
#define TV_ENERGY_TABLE        NO
```

tell **PLUTO** to replace the thermal EOS with a temperature table $T = T(p_i, \rho_j)$ while still using the analytical approach (i.e. direct function evaluation or root finder) for the caloric EOS.

The tables $T(p_i, \rho_j)$ and $\rho e(T_i, \rho_j)$ are initialized at runtime and used throughout the integration. The number of points needed to construct such tables is fixed by the constants {`PV_TEMPERATURE_TABLE_NX`, `PV_TEMPERATURE_TABLE_NY`} for the first table and {`TV_ENERGY_TABLE_NX`, `TV_ENERGY_TABLE_NY`} for the second one. To avoid the occurrence of spurious waves in the solution of the Riemann problem, a monotone cubic spline is always used in the temperature grid, see Appendix C of [VMBM15].

7.4 The *TAUB* Equation of state

The Taub-Matthews (TM) equation of state is available to describe a relativistic perfect gas, for which the adiabatic exponent is a function of the temperature. The actual expression for the Synge gas [Syn57] is rather complex and **PLUTO** employs a 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 [MPB05, MM07]:

$$\left(h - \frac{p}{\rho}\right)\left(h - 4\frac{p}{\rho}\right) = 1\,, \tag{7.13}$$

where h is the specific enthalpy related to the internal energy and pressure through

$$h = 1 + e + \frac{p}{\rho}\,.$$

8. Nonideal Effects

In this chapter we give an overview of the code capabilities for treating dissipative (or diffusion) terms which, at present, include

- Hall MHD (MHD), described in §8.1;
- Resistivity (MHD), described in §8.2;
- Thermal conduction (HD, MHD), described in §8.3.
- Viscosity (HD, MHD), described in §8.4;

Each modules can be individually turned on from 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 §8.5. Depending on the time step restriction, you may include diffusion processes by setting the corresponding sub-menu choice(s) to *`EXPLICIT`* or to *`SUPER_TIME_STEPPING`*, respectively.

8.1 Hall MHD

The Hall MHD term can be enabled from the python menu by setting `HALL_MHD` to *`EXPLICIT`*. In Hall MHD the electric field includes the convective term and the Hall term (see Eq. 6.6):

$$c\boldsymbol{E} = -(\boldsymbol{v} + \boldsymbol{v}_H) \times \boldsymbol{B} \,,$$

where

$$\boldsymbol{v}_H = -\frac{\boldsymbol{J}}{en_e} \tag{8.1}$$

is the Hall velocity, n_e is the electron number density and e is the (absolute value) of the electron charge. Physically, the Hall term decouples ion and electron motion on ion inertial length scales: $L \ll c/\omega_{pi}$ where $\omega_{pi} = 4\pi n_i e^2/m_i$ [Hub05, TMG08].

The induction and total energy equations in the MHD system now take the form

$$\begin{array}{lcl} \dfrac{\partial \boldsymbol{B}}{\partial t} + \nabla \times \left[-\left(\boldsymbol{v} + \boldsymbol{v}_H\right) \times \boldsymbol{B}\right] & = & 0 \\ \dfrac{\partial E_t}{\partial t} + \nabla \cdot \left[(E_t + p_t)\boldsymbol{v} - \boldsymbol{B}\left(\boldsymbol{v} \cdot \boldsymbol{B}\right) - \left(\boldsymbol{v}_H \times \boldsymbol{B}\right) \times \boldsymbol{B}\right] & = & 0 \end{array} \tag{8.2}$$

The factor en_e can be defined by editing a local copy of PLUTO/Src/MHD/Hall_MHD/hall_ne.c which includes the function **`HallMHD_ne()`**:

Syntax:

```
double HallMHD_ne(double *v)
```

Arguments:

- `v`: a pointer to a vector of primitive variables;

The function return the desired value of en_e, function of primitive variables.

Note: *Units.* The value of en_e should be specified in dimensionless form as

$$en_e = (en_e)_{\rm cgs}\frac{L_0}{c\sqrt{4\pi\rho_0}} \tag{8.3}$$

where $(en_e)_{\rm cgs}$ is the c.g.s value, L_0 and ρ_0 are the unit length and the unit density, respectively. By choosing $L_0 = c/\omega_{pi}$ (ion skin depth) where $\omega_{pi} = \sqrt{4\pi e^2 n_i/m_i}$ is the ion plasma frequency, Eq. (8.3) takes the simpler form

$$en_e = \frac{1}{4\pi}\frac{(n_e)_{\rm cgs}}{n_0} \tag{8.4}$$

where $n_0 = \rho_0/m_i$ is the reference number density.

Test problems can be found in the PLUTO/Test_Problems/MHD/Hall_MHD/ directory.

At present, the Hall MHD module works only with the *`hll`* Riemann solver, *`DIVERGENCE_CLEANING`* . Time-integration is explicit and carried out in fully unsplit form by including all fluxes (convective + Hall) simultaneously. However, due to its dispersive nature the time-step is restricted by the condition

$$\Delta t \lesssim C_a \frac{\Delta x^2}{B/ne}$$

with $C_a \lesssim 0.25$ for an explicit scheme. Future versions of the code will address these issues.

8.2 Resistivity

The resistive module is enabled by setting `RESISTIVITY` to either *`EXPLICIT`* (for time-explicit computations) or to *`SUPER_TIME_STEPPING`* (to accelerate explicit computations) from the Python menu.

From Eq. (6.6), electric resistivity is modeled by introducing the resistivity tensor η so that the electric field becomes $c\boldsymbol{E} = -\boldsymbol{v}\times\boldsymbol{B} + \eta\cdot\boldsymbol{J}/c$, where $\boldsymbol{J} \equiv c\nabla\times\boldsymbol{B}$ is the current density. The induction and energy equations are modified as:

$$\begin{aligned}\frac{\partial \boldsymbol{B}}{\partial t} + \nabla\times(-\boldsymbol{v}\times\boldsymbol{B}) &= -\nabla\times\left(\eta\cdot\frac{\boldsymbol{J}}{c}\right)\\ \frac{\partial E_t}{\partial t} + \nabla\cdot\left[(E_t+p_t)\boldsymbol{v} - \boldsymbol{B}\left(\boldsymbol{v}\cdot\boldsymbol{B}\right)\right] &= -\nabla\cdot\left[(\eta\cdot\boldsymbol{J})\times\boldsymbol{B}\right]\,.\end{aligned} \tag{8.5}$$

Similarly, the internal energy equation modifies to

$$\frac{\partial p}{\partial t} + \boldsymbol{v}\cdot\nabla p + \rho c_s^2\nabla\cdot\boldsymbol{v} = \frac{1}{c^2}(\Gamma-1)\left(\eta\cdot\boldsymbol{J}\right)\cdot\boldsymbol{J}\,. \tag{8.6}$$

The resistive tensor η is assumed to be diagonal with components

$$\eta \equiv \mathrm{diag}\left(\eta_{x1},\eta_{x2},\eta_{x3}\right)\,. \tag{8.7}$$

The module is implemented in the Src/MHD/Resistivity directory and the functional form of η can be specified by editing your local copy of PLUTO/Src/MHD/Resistivity/res_eta.c which includes the function **`Resistive_eta()`**:

Syntax:

```
void Resistive_eta(double *v, double x1, double x2, double x3,
                   double *J, double *eta)
```

Arguments:

- `v`: a pointer to a vector of primitive variables;
- `x1,x2,x3`: local spatial coordinates;
- `J`: a pointer to the electric current vector;
- `eta`: a pointer to an array containing the three components of the resistive diagonal tensor.

The resistive module works in 1, 2 and 3 dimensions in all systems of coordinates on both uniform and non-uniform grid, although higher accuracy can be achieved on uniform grid spacing. Both cell-centered and staggered MHD are supported using either *`EXPLICIT`*, or *`SUPER_TIME_STEPPING`* integration. *`RK_LEGENDRE`* methods work, at present, only with cell-centered formulation (see §8.5).

The current ($\nabla\times\boldsymbol{B}$ computed in the function **`GetCurrent()`**) is located at *cell edges* and stored in the array `d->J`. For cell-centered MHD, the three components of the current are later calculated at the zone interfaces by averaging edge values (**`ResistiveFlux()`**) normal to the sweep integration direction.

8.3 Thermal Conduction

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

$$\frac{\partial E_t}{\partial t} + \nabla \cdot \left[(E_t + p_t)\boldsymbol{v} - \boldsymbol{B}\left(\boldsymbol{v} \cdot \boldsymbol{B}\right)\right] = \nabla \cdot \boldsymbol{F}_c\,, \tag{8.8}$$

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:

$$\boldsymbol{F}_c = \frac{F_{\rm sat}}{F_{\rm sat} + |\boldsymbol{F}_{\rm class}|}\boldsymbol{F}_{\rm class}\,, \tag{8.9}$$

see [Spi62, OBR+08].

In the MHD case, thermal conductivity is highly anisotropic being largely suppressed in the direction transverse to the magnetic 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 [Bal86]:

$$\boldsymbol{F}_{\rm class} = \kappa_{\parallel}\hat{\boldsymbol{b}}\Big(\hat{\boldsymbol{b}} \cdot \nabla T\Big) + \kappa_{\perp}\left[\nabla T - \hat{\boldsymbol{b}}\Big(\hat{\boldsymbol{b}} \cdot \nabla T\Big)\right]\,, \tag{8.10}$$

where the subscripts $\parallel$ and $\perp$ denote, respectively, the parallel and normal components to the magnetic field, T is the temperature, $\kappa_{\parallel}$ and $\kappa_{\perp}$ are the thermal conduction coefficients along and across the field. In the purely hydrodynamical limit (no magnetic field), Eq. (8.10) reduces to $\boldsymbol{F}_c = \kappa_{\parallel}\nabla T$.

Saturated effects are accounted for by making the flux independent of ∇T for very large temperature gradients [Spi62, CM77]. 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. Note, however, that it is possible to suppress saturation effects by turning the macro `TC_SATURATED_FLUX` to *NO*, see also Appendix B.3: in this case $\boldsymbol{F}_c = \boldsymbol{F}_{\rm class}$.

The coefficients appearing in Eq. (8.10), (8.9) 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 $\kappa_{\parallel} \to *{\rm kpar}$, $\kappa_{\perp} \to *{\rm knor}$ and $\phi \to *{\rm phi}$. The variable $*{\rm knor}$ can be ignored in the HD case, where $\kappa = \kappa_{\parallel}$. Proper setting of units and dimensions is briefly discussed in §8.3.1.

The thermal conduction module is implemented inside Src/Thermal_Conduction and works in 1, 2 and 3 dimensions in all systems of coordinates. 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 §8.5.

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, [BTH08, MZT+12].
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 §8.5) and a more tight coupling between the hydrodynamical and thermal conduction scale. The latter condition may be accomplished by lowering the `rmax_par` parameter (§4.3) which controls the ratio between the current time step and the diffusion time scale, see also §8.5. An example problem can be found in Test_Problems/MHD/Thermal_conduction/Blast.

8.3.1 Dimensions

Equations (8.8)-(8.10) 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 $T = (p/\rho)\mathcal{K}\mu$, where p and ρ are in code units and μ is the mean molecular weight. Here ρ_0, v_0, L_0 are the unit density, velocity, length while $\mathcal{K}$ is the **KELVIN** macro, see §5.1.2. The thermal conduction coefficients must be properly defined by re-absorbing the correct normalization constants in the **TC_kappa()** function as follows

$$\kappa \to \kappa_{cgs} \frac{\mu m_u}{\rho_0 v_0 L_0 k_B} \tag{8.11}$$

where, for instance, one may use $\kappa_{cgs,\parallel} = 5.6 \cdot 10^{-7} T^{5/2}$ and $\kappa_{cgs,\perp} = 3.3 \cdot 10^{-16} n_H^2 / (\sqrt{T} \boldsymbol{B}_{\rm cgs}^2)$, 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.

8.4 Viscosity

The viscous stresses enter the HD and MHD equations with two parabolic diffusion terms in the momentum and energy equations. 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:

$$\begin{aligned} \frac{\partial \boldsymbol{m}}{\partial t} + \nabla\cdot\mathsf{T}_h &= \nabla\cdot\Pi \\ \frac{\partial E_t}{\partial t} + \nabla\cdot\boldsymbol{F}_E &= \nabla\cdot(\boldsymbol{v}\cdot\Pi) \end{aligned} \tag{8.12}$$

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

$$\Pi_{ij} = 2\frac{\nu_1}{h_i h_j}\left(\frac{v_{i;j}+v_{j;i}}{2}\right) + \left(\nu_2 - \frac{2}{3}\nu_1\right)\nabla\cdot\boldsymbol{v}\delta_{ij}\,. \tag{8.13}$$

Coefficients ν_1 and ν_2 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 diffusion fluxes on the right hand side are added (parabolic_update.c) after the computation of the hyperbolic right hand side (if *EXPLICIT* is chosen). In curvilinear geometries, additional geometrical source terms coming from the tensor's divergence are added to the right hand side of the equations. On the other hand, if VISCOSITY is set to *STS* or *RKL*, advection and diffusion terms are treated separately using operator splitting, still retaining conservation form. 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.

When ROTATING_FRAME is enabled (§2.2.7), the right hand side of the momentum equation is computed using the total velocity $\boldsymbol{v}_t = \boldsymbol{v} + w_\phi\hat{\boldsymbol{\phi}}$ while the viscous energy flux is augmented - see the function **ViscousRHS()** in Viscosity/viscous_rhs.c - as

$$\boldsymbol{F}_E^{\rm visc} \leftarrow \boldsymbol{v}\cdot\Pi + w_\phi \boldsymbol{F}_\phi^{\rm visc} \tag{8.14}$$

where $w_\phi = R\Omega_z$.

The viscous transport coefficients ν_1 (shear) and ν_2 (bulk) are defined in the function **Visc_nu()** in the source file PLUTO/Src/Viscosity/visc_nu.c. This file should be copied from its original folder to the actual working directory before doing any modification.

The **Visc_nu()** function has the following syntax:

Syntax:

```
void Visc_nu(double *v, double x1, double x2, double x3,
                        double *nu1, double *nu2)
```

Arguments:

- v: a pointer to a vector of primitive variables;
- x1,x2,x3: local spatial coordinates;
- *nu1: a pointer to the 1st viscous coefficient (shear);
- *nu2: a pointer to the 2nd viscous coefficient (bulk);

Even though the behaviour of these coefficients is arbitrary, according to the user's needs, for monoatomic gases Molecular Theory gives $\nu_2 = 0$. The coefficient of shear viscosity ν_1, 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 [LL87] and [Tor97]. It should be noted, nonetheless, that both transport coefficients must have dimensions of $\rho\times{\rm length}^2/{\rm time}$, for the correct control of the timestep, according to the stability condition discussed at the beginning of this chapter.

8.5 Numerical Integration of Diffusion Terms

8.5.1 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:

$$\boldsymbol{F} \to \boldsymbol{F}_{\rm hyp} + \boldsymbol{F}_{\rm par} \tag{8.15}$$

where "hyp" and "par" are, respectively, the hyperbolic and parabolic fluxes (see also §3.1 of [MZT+12]).

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.

8.5.2 Super-Time-Stepping (STS)

STS, [AAG96], 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`):

$$\Delta t^n = \Delta t_{\rm par}\frac{N}{2\sqrt{\nu}}\frac{(1+\sqrt{\nu})^{2N}-(1-\sqrt{\nu})^{2N}}{(1+\sqrt{\nu})^{2N}+(1-\sqrt{\nu})^{2N}}\,, \qquad \text{with} \qquad \Delta t_{\rm par} = \frac{C_p}{\displaystyle\frac{2}{N_{\rm dim}}\max_{ijk}\left(\sum_d \frac{\mathcal{D}_d}{\Delta l_d^2}\right)}\,. \tag{8.16}$$

Here $\Delta t_{\rm par}$ is the explicit parabolic time step computed in terms of the diffusion coefficient $\mathcal{D}$ and physical grid size 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 8.1. To change the default value of $\nu =$ `STS_NU`, redefine it in the user-defined symbolic constant section of definitions.h, see §2.3.

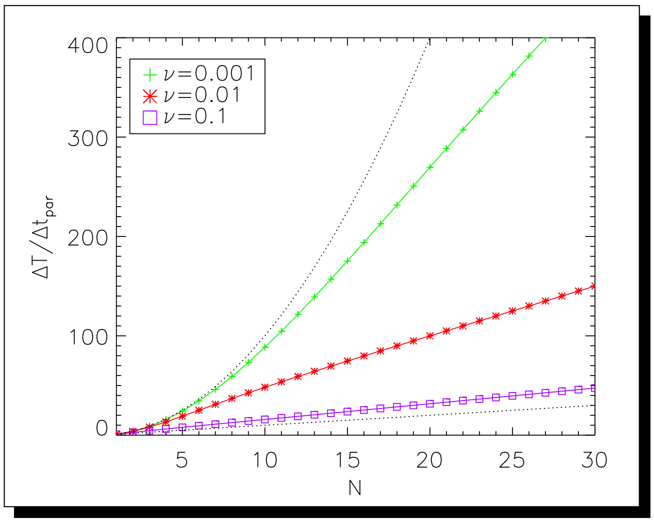


Figure 8.1: Length of a super-step (in units of the explicit one, $\Delta T/\Delta t_{\rm par}$) as function of the number of sub-steps N using different values of $\nu = 10^{-3}$ (green, plus sign), $\nu = 10^{-2}$ (red, asterisk - default), $\nu = 10^{-1}$ (purple, square). The upper dotted lines gives 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.

Stability analysis for the constant coefficient diffusion equation, [Bec92], indicates that the value of C_p (parabolic Courant number) should be $\leq 1/N_{\rm dim}$ ($N_{\rm dim}$ 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 §4.3.

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 $r_{\max}\Delta t_{\rm par}$, with $\Delta t_{\rm par}$ defined by Eq. (8.16) and $r_{\max}$ a free parameter, see §4.3. In this cases, $r_{\max}$ 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.

During the STS step, momentum, magnetic field or total energy are evolved in time even if the `ENTROPY_SWITCH` has been enabled.

8.5.3 Runge-Kutta Legendre (RKL)

RKL is yet another explicit multistage time-stepping scheme that has extended stability properties and improves over the standard first-order STS method in terms of both accuracy and robustness. RKL is based on the Legendre polynomial and has been introduced by [MBA12] and a method paper for the PLUTO code has been presented by [VPM+17] in the context of thermal conduction.

If Δt_h ($\propto \Delta x$) and Δt_p ($\propto \Delta x^2$) are, respectively, the hyperbolic and parabolic time steps, the number of sub-stages s is given by

$$
\begin{array}{rclll}
\Delta t_h &=& \Delta t_p\dfrac{s^2+s}{2} & \text{for} & RKL1 \\[2ex]
\Delta t_h &=& \Delta t_p\dfrac{s^2+s-2}{4} & \text{for} & RKL2
\end{array}
\qquad (8.17)
$$

Here RKL1 and RKL2 denotes the first and second-order accurate RKL method. By default, the second order method is employed (`RKL_ORDER` = 2, see Appendix B.3).

Several numerical benchmarks indicate that RKL yields more accurate, oscillation-free and, in some cases, larger time steps can be taken when compared to standard STS. In addition, RKL methods do not depend on any tunable parameter (such as ν). Using Eq. [17] and [18] of [VPM+17], we plot in Fig. 8.2 the effective parabolic CFL number $C_p = \Delta t_h/2\Delta t_p$ as a function of the number of stages s. Note that STS does not offer substantial gain with respect to an explicit scheme ($C_p \propto s$) when $s \gtrsim 1/2\sqrt{\nu}$.

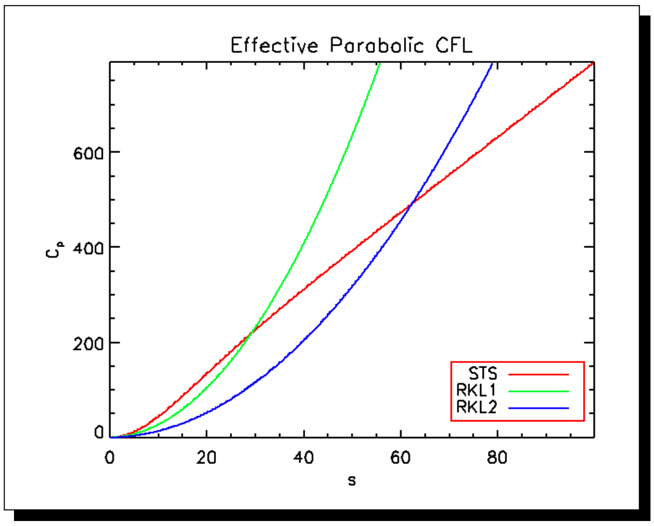


Figure 8.2: Effective parabolic CFL as function of the number of sub-steps s for STS (using $\nu = 10^{-3}$, red line) RKL1 (green) and RKL2 (blue).

On the CPU side, the actual implementation of RKL methods is a factor of $20 - 40\%$ more intensive than STS, depending on the temporal accuracy. We refer the reader to the paper by [VPM^{+}17] for a thorough comparison between RKL and STS.

9. Optically Thin Cooling

PLUTO can include time-dependent optically thin radiative losses in a fractional step formalism in which the hydrodynamical evolution and the source step are solved separately using operator splitting. This preserves 2^{nd} order accuracy in time if both the advection and source steps are at least 2^{nd} order accurate. During the cooling source step, specifically, **PLUTO** solves the internal energy and chemical reaction network equations

$$\left\{\begin{array}{lcl} \dfrac{\partial(\rho e)}{\partial t} & = & -\Lambda(n, T, \boldsymbol{X}) \\ \dfrac{\partial \boldsymbol{X}}{\partial t} & = & \boldsymbol{S}(\boldsymbol{X}, T) \end{array}\right. \tag{9.1}$$

where Λ is a cooling (or heating) term, $\boldsymbol{X}$ is an array of fractional abundances (typically ion or molecule number fractions) and $\boldsymbol{S}$ is a reaction source term. The right-hand side of Equations (9.1) is implemented in the function Src/Cooling/<COOLING>/radiat.c of each corresponding cooling module, except for the *POWER_LAW* cooling where integration is performed analytically. The user can select one among several different cooling module by setting the `COOLING` flag during the python script:

- *POWER_LAW*: power-law cooling, see §9.1;
- *TABULATED*: only the equation for the internal energy with a tabulated cooling function $\Lambda(T)$ is provided. No chemical network, see §9.2;
- *SNEq*: cooling function for atomic hydrogen, $\boldsymbol{X} = \{X_{HI}\}$, including ionization, recombination and collisionally excited emission lines, §9.3;
- *H2_COOL*: cooling function for atomic and molecular and atomic hydrogen, $\boldsymbol{X} = \{X_{H2}, X_{HI}, X_{HII}\}$, including ionization, recombination and collisionally excited emission lines, §9.4;
- *MINEq*: cooling function for atomic and molecular and atomic hydrogen treating the time-dependent ionization state of the plasma, $\boldsymbol{X} = \{X_H, X_{He}, X_C, X_N, X_Ne, X_O, X_S\}$, see §9.5.

Cooling modules are implemented inside the Src/Cooling directory and require three dimensional constants to be correctly initialized. Dimensional constants are essential to scale data values to cgs physical units as explained in §5.1.1.

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`: sets the cut-off temperature below which cooling is artificially set to 0.

9.1 Power Law Cooling

Power law cooling is the most simple form of cooling, where the loss term in the internal energy equation becomes:

$$\Lambda = a_r\rho^2T^\alpha \tag{9.2}$$

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

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

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

$$p^{n+1} = \left\{\begin{array}{cl} \Big[(p^n)^{1-\alpha} - \Delta t C(1-\alpha)\Big]^{\frac{1}{1-\alpha}} & \text{for} \quad \alpha \neq 1 \\ p^n\exp\left(-C\Delta t\right) & \text{for} \quad \alpha = 1 \end{array}\right. \tag{9.3}$$

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

The default power law accounts for bremsstrahlung cooling by solving

$$\frac{dp_{\rm cgs}}{dt_{\rm cgs}} = -(\Gamma-1)\frac{a_{\rm br}}{\mu^2m_H^2}\rho^2_{\rm cgs}\sqrt{T(K)} \qquad \Longrightarrow \qquad \frac{dp}{dt} = -C\rho^2\sqrt{\frac{p}{\rho}} \tag{9.4}$$

with p, t and ρ given in code units and

$$C = a_{\rm br}\frac{\Gamma-1}{(k_B\mu m_H)^{3/2}}\frac{\rho_0L_0}{v_0^2}$$

where ρ_0, v_0 and L_0 are the reference density, velocity and length defined in §5.1.1 and $a_{\rm br} = 2\cdot10^{-27}$ in expressed in c.g.s. units. The implementation of this cooling step, with $\alpha = 1/2$, can be found under Src/Cooling/Power_Law/cooling.c.

9.2 Tabulated Cooling

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

$$\Lambda = n^2\tilde{\Lambda}(T) \ , \qquad \text{with} \qquad n = \frac{\rho}{\mu m_u} \tag{9.5}$$

when the cooling/heating function $\tilde{\Lambda}(T)$ is not known analytically but rather is available as a table sampled at discrete (not necessarily equidistant) points, i.e., $\tilde{\Lambda}_j \equiv \tilde{\Lambda}(T_j)$. 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 $\tilde{\Lambda}$ in `ergs`·`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 `UNIT_DENSITY`, `UNIT_LENGTH` and `UNIT_VELOCITY` have been defined in **`Init()`**.

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

$$\Lambda = \Lambda(\boldsymbol{V}) \,, \tag{9.6}$$

where $\boldsymbol{V}$ is a vector 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.

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

9.3 Simplified Non-Equilibrium Cooling: SNEq

This module is implemented in the Src/Cooling/SNEq directory and introduces a new variable, with index X_HI used to label the fraction of neutrals x_{HI}:

$$x_{HI} = \frac{n_{H_I}}{n_H} \,. \tag{9.7}$$

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

```
v[X_HI] = 0.2;    /*  for example  */
```

in your **Init()** function. The fraction of neutrals is treated as a passive scalar during the hydro step while it is governed by the following ODE during the cooling step:

$$\frac{\partial x_{HI}}{\partial t} = S = n_e\Big[-(c_r + c_i)\,x_{HI} + c_r\Big] \tag{9.8}$$

together with the energy equation

$$\frac{\partial(\rho e)}{\partial t} = -\Lambda = -n_e n_H\left(\sum_{k=1}^{k=16} j_k + w_{i/r}\right) \tag{9.9}$$

where $n_e = n_H(1 - x_{HI} + x_Z)$ is the electron number density while x_Z ($= 10^{-3}$ by default) is the fractional number density of metals. In Eq. (9.9) 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).

The coefficient j_k in (9.9) has dimensions of 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. (9.9) $w_{i/r}$ represents the thermal energy lost by ionization and recombination:

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

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

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

Table 9.1: Summary of the chemistry reaction set. T is the temperature in Kelvin, $T_{\rm eV}$ is the temperature in electron-volts and T_2 = T/100

No.	Reaction	Rate Coefficient (cm^{3}s^{-1})	Reference [a]
1.	$H + e^- \to H^+ + 2e^-$	$k_1 = 5.84 \times 10^{-11} T^{0.5} \exp(-157,809.0/T)$	1
2.	$H^+ + e^- \to H + h\nu$	$k_2 = 2.6 \times 10^{-11} T^{-0.5}$	1
3.	$H_2 + e^- \to 2H + e^-$	$k_3 = 4.4 \times 10^{-10} T^{0.35} \exp(-102,000.0/T)$	2
4.	$H_2 + H \to 3H$	$k_4 = 1.067 \times 10^{-10} T_{\rm eV}^{2.012}$	
		$\exp[(-4.463/T_{\rm eV})(1 + 0.2472 T_{\rm eV})^{3.512}]$	3
5.	$H_2 + H_2 \to H_2 + 2H$	$k_5 = 1.0 \times 10^{-8} \exp(-84,100/T)$	4
6.	$H + H \xrightarrow{\rm dust} H_2$	$k_6 = 3.0 \times 10^{-17} \sqrt{T_2}(1.0 + 0.4\sqrt{T_2} + 0.2 T_2 + 0.08 T_2^2)$	5

[a]REFERENCES – (1) [RBMF97] [Eq. 1e] (2) [GP98] [Eq. H17]; (3) [AAZN97] [Tab. 3 Eq. 13]; (4) [WAMM07] [UMIST Database] (5) [HM79] [Eq. 3.8]

9.4 Molecular Hydrogen Non-Equilibrium Cooling: *H2_COOL*

This module is implemented in the Src/Cooling/H2_COOL directory and introduces three new variables, with index `X_HI`, `X_H2` and `X_HII` used to label the fraction of atomic hydrogen, molecular hydrogen and ionized hydrogen respectively as follows:

$$x_{H2} = \frac{n_{H_2}}{n_H}, \qquad x_{HI} = \frac{n_{H_I}}{n_H}, \qquad x_{HII} = \frac{n_{H_{II}}}{n_H},$$

where, the total hydrogen number density $n_H = n_{H_I} + n_{H_{II}} + 2n_{H_2}$.

You can assign these hydrogen fractions, in a similar manner like the *SNEq* module,

```
/*  for example  */
v[X_HI] = 0.2;
v[X_H2] = 0.4;
v[X_HII] = 1.0 - v[X_HI] - 2.0*v[X_H2];
```

in your **`Init()`** function. Note, the value of `v[X_H2]` should be between 0.0 and 0.5, while the remaining two hydrogen fractions can have values ranging from 0.0 to 1.0, such that their sum is conserved.

The chemical evolution of molecular, atomic and ionized hydrogen is governed by equations listed in Table 9.1. The number density of various hydrogen forms are determined by solving the chemical rate equations, which have a general form as,

$$\frac{dn_i}{dt} = \sum_{j,k} k_{j,k} n_j n_k - n_i \sum_j k_{i,j} n_j, \tag{9.10}$$

where, n is the number density, $k_{j,k}$ is the rate of formation of ith specie from all j and k species and $k_{i,j}$is the rate of destruction of ith specie due to all j species.

The code integrates the three hydrogen fractions defined above using the advection equation of the form:

$$\frac{\partial X_i}{\partial t} = -\boldsymbol{v} \cdot \nabla X_i + S_i\,, \tag{9.11}$$

where the first term on the rhs is treated during the hydro step while only the second is integrated during the cooling step. The source terms S$_i$ is essentially the difference between formation and destruction rate of a particular specie (see eq.9.10). Additionally, the internal energy losses take into accounts various hydrogen cooling processes,

$$\Lambda = \Lambda_{\rm CI} + \Lambda_{\rm RR} + \Lambda_{\rm rotvib} + \Lambda_{\rm H2diss} + \Lambda_{\rm grain}, \tag{9.12}$$

where, $\Lambda_{\rm CI}$ and $\Lambda_{\rm RR}$ are losses due to collisional ionization and radiative recombination respectively. The remaining terms, $\Lambda_{\rm rotvib}$, $\Lambda_{\rm H2diss}$ and $\Lambda_{\rm grain}$ are associated with molecular hydrogen and represent

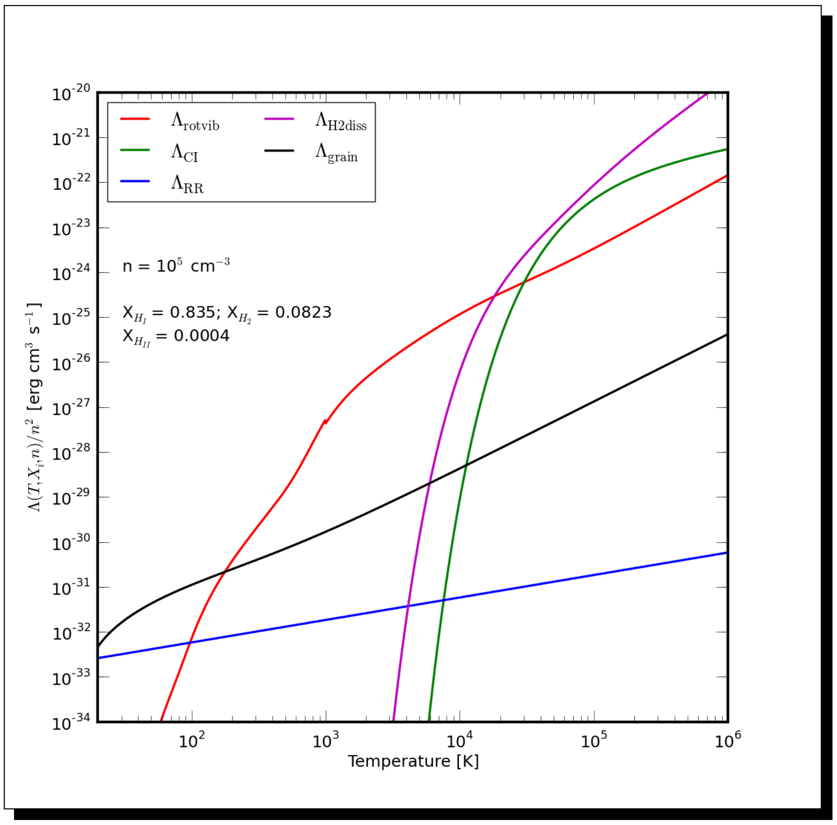


Figure 9.1: Variation of the radiative cooling functions Λ_i with temperature due to various processes that can affect the total energy of a gas comprising of atomic, molecular and ionized hydrogen. Here the total number density n is set to be 10^5cm^{-3}, while the fractions of different hydrogen species are : `X_HI` = 0.835, `X_H2` = 0.0823 and `X_HII` = 0.0004

losses due to rotational-vibrational cooling, dissociation and gas-grain processes. Their variation with temperature for a particular set ofm hydrogen fractions is shown in fig.9.1. Depending on the requirement, the user can add more components to the cooling function, for e.g., cooling due to fixed fractions of standard molecules like CO, OH, H_2O etc or contributions from collisional excitation of lines as indicated in the *SNEq* module.

9.5 Multi-Ion Non-Equilibrium Cooling: ***MINEq***

This module computes the dynamical evolution and ionization state of the plasma using the multi-ion model of [TMM08] 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: `X_HI`, `HeI`, `HeII`, `X_CI`, `X_CII`, `X_CIII`, `X_CIV`, `X_CV`, `X_NI`, `X_NII`, `X_NIII`, `X_NIV`, `X_NV`, `X_OI`, `X_OII`, `X_OIII`, `X_OIV`, `X_OV`, `X_NeI`, `X_NeII`, `X_NeIII`, `X_NeIV`, `X_NeV`, `X_SI`, `X_SII`, `X_SIII`, `X_SIV`, `X_SV`. Ionized hydrogen is simply $1 - v[X\_HI]$. You can assign the fraction of any ion specie by setting, in the usual fashion

```
v[X_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.

<u>**Note**</u>: The number of ions for C, N, O, Ne and S may be reduced from 5 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 current default value is 3.

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:

$$\frac{\partial X_i}{\partial t} + \boldsymbol{v}\cdot\nabla X_i = S_i\,, \tag{9.13}$$

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 [TMM08]. Similarly, the energy loss term is

$$\Lambda = \Big[n_{\rm at} n_{\rm el}\Lambda_1\left(T,\boldsymbol{X}\right) + L_{\rm FF} + L_{\rm I-R}\Big]\,,\qquad \Lambda_1(T,\boldsymbol{X}) = \sum_k X_k\mathcal{L}_k(n_{\rm el},T)B_k\,, \tag{9.14}$$

where B_k is the fractional abundance of the element, and

$$\mathcal{L}_k = \sum_i N_i \sum_{j<i} A_{ij} h\nu_{ij}\,, \tag{9.15}$$

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.

In Eq. (9.14), $L_{\rm FF}$ and $L_{\rm I-R}$ represent 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.

`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.

10. Additional Modules

10.1 The ShearingBox Module

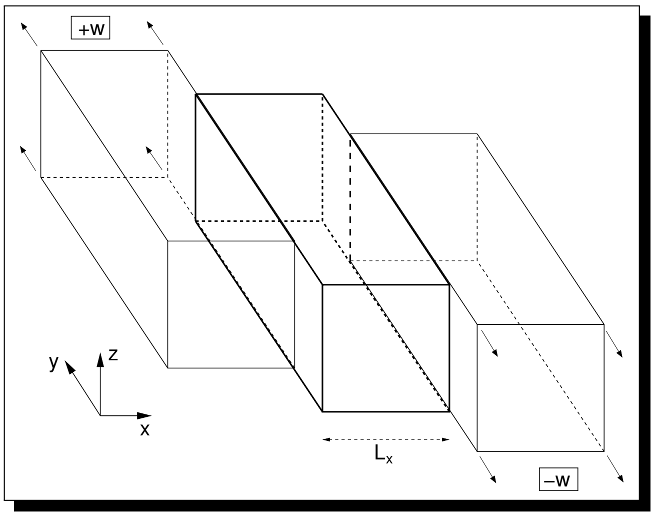


Figure 10.1: Schematic representation of the shearing boundary condition. The computational domain (central box) is assumed to be surrounded by identical boxes sliding with constant velocity $w = |q\Omega_0 L_x|$ with respect to one another.

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,

$$v_y = -q\Omega_0 x\,, \qquad \text{with} \quad q = -\left.\frac{d\log\Omega(R)}{d\log R}\right|_{R=R_0} \tag{10.1}$$

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 HD or MHD equations in a non-inertial frame so that the momentum and energy equations become

$$\begin{array}{rcl} \dfrac{\partial(\rho\boldsymbol{v})}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}\boldsymbol{v} - \boldsymbol{B}\boldsymbol{B}) + \nabla p_t & = & \rho\boldsymbol{g}_s - 2\Omega_0\hat{\boldsymbol{z}}\times\rho\boldsymbol{v} \\ \dfrac{\partial E}{\partial t} + \nabla\cdot\left[(E+p_t)\,\boldsymbol{v} - (\boldsymbol{v}\cdot\boldsymbol{B})\,\boldsymbol{B}\right] & = & \rho\boldsymbol{v}\cdot\boldsymbol{g}_s\,, \end{array} \tag{10.2}$$

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. (10.2) represents the Coriolis force. The continuity and induction equations retain the same form as the original system.

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 10.1. In other words, the boundary conditions at the left/right x-boundaries are

$$\left\{\begin{array}{rcl} q(x,y,z,t) & = & q\left(x\pm L_x, y\mp wt, z, t\right) \\ v_y(x,y,z,t) & = & v_y\left(x\pm L_x, y\mp wt, z, t\right) \pm w\,, \end{array}\right. \tag{10.3}$$

where q is any other flow quantities except v_y.

The ShearingBox module is implemented inside Src/MHD/ShearingBox and works, at present, with the HD equations or with *CONSTRAINED_TRANSPORT* MHD. Parallelization can be performed in all three spatial dimensions.

10.1.1 Using the module

The shearingbox module is enabled by invoking the Python setup script with the `--with-sb` option. It is compatible with the *ISOTHERMAL* or *IDEAL* equations of state.

Initial conditions are specified, as usual, in the **`Init()`** function where the orbital speed must be set to $v_y = -q\Omega_0 x$. A simple example corresponding to $\rho = 1$, $p = c_s^2\rho$ and plasma $\beta = 10^3$ is given below:

```
  cs     = 1.0; /* Isothermal sound speed */
  v[RHO] = 1.0;
  v[VX1] = 0.0;
  v[VX2] = -SB_Q*SB_OMEGA*x1;  /* Orbital velocity */
  v[VX3] = 0.0;
#if EOS == IDEAL
  v[PRS] = v[RHO]*cs*cs;
#elif EOS == ISOTHERMAL
  g_isoSoundSpeed = cs;
#endif
#if PHYSICS == MHD
  beta   = 1.e3;
  v[BX1] = 0.0;
  v[BX2] = 0.0;
  v[BX3] = cs*sqrt(2.0/beta);  /* Vertical field (net flux) */
#endif
```

The numerical value of q and Ω_0 is prescribed (starting with **PLUTO** 4.2) using the macros `SB_Q` and `SB_OMEGA` which, by default, take the value of $3/2$ and 1, respectively. Should you change the default value, add them as user-defined constants as explained in §2.3.

Only gravitational forces must be given through the **`BodyForceVector()`** function since Coriolis term are separately included by **PLUTO** . An example containing several configurations can be found in the Test_Problems/MHD/Shearing_Box/ directory.

Boundary conditions must be prescribed as *shearingbox* at the `X1_BEG` and `X1_END` boundaries, periodic in the azimuthal (y) direction but can be freely assigned in the vertical direction z.

Compatibility with the `FARGO` module. The shearingbox module is fully compatible with the `FARGO` algorithm and a significant gain may be obtained for boxes with large aspect ratio ($L_x \gg L_z$). To enable both modules, you must invoke the python script with the `--with-sb` and the `--with-fargo` options. In this case the macro `FARGO_AVERAGE_VELOCITY` (§10.2.1) is automatically turned to *NO* so that the background orbital velocity is prescribed analytically with the **`FARGO_SetVelocity()`** function.

With `FARGO`, however, the source terms in the momentum and energy equations are slightly different [MFS+12, SG10]:

$$\begin{aligned}
\boldsymbol{S_m} &= \Big[2\Omega_0\rho v'_y\Big]\hat{\boldsymbol{i}} + \Big[(q-2)\Omega_0\rho v_x\Big]\hat{\boldsymbol{j}} + \Big[-\rho\Omega_0^2 z\Big]\hat{\boldsymbol{k}} \\
S_E &= (\rho v'_y v_x - B_y B_x)q\Omega_0 + \rho v_z(-\Omega_0^2 z)
\end{aligned}$$

One can see that radial gravity disappears and, therefore, only the vertical component of gravity must be included in the **`BodyForceVector()`** or **`BodyForcePotential()`** functions.
The additional term in the energy equation represents the work done by Reynolds and magnetic stresses because of the radial shear [SG10]. This term is accounted separately for during the `FARGO` transport step.

10.2 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, see [MFS+12].

The algorithm decomposes the total velocity into an average azimuthal contribution and a residual term,

$$\boldsymbol{v} = \boldsymbol{v}' + \boldsymbol{w} \tag{10.4}$$

where $\boldsymbol{v}'$ is called the residual velocity while $\boldsymbol{w}$ is a background velocity field that must be solenoidal. The MHD or HD equations are solved in two steps: i) a linear transport operator corresponding to the velocity $\boldsymbol{w}$ in the direction of orbital motion and ii) a standard nonlinear solver applied to the original equations written in terms of the residual velocity $\boldsymbol{v}'$.

The Courant condition is then computed only from the residual velocity, leading to substantially larger time steps. In [MFS+12] it has been shown that if the characteristic velocity of fluctuations are comparable in magnitude than the expected gain in polar coordinates is, roughly,

$$\frac{\Delta t_F}{\Delta t_s} \approx \frac{\displaystyle\max_{ijk}\left[\frac{1}{\Delta R} + \frac{M+1}{R\Delta\phi} + \frac{1}{\Delta z}\right]}{\displaystyle\max_{ijk}\left[\frac{1}{\Delta R} + \frac{1}{R\Delta\phi} + \frac{1}{\Delta z}\right]}\,, \tag{10.5}$$

where Δt_F and Δt_s are the FARGO time step and the standard time step, respectively, whereas $M = |w|/\lambda'$ and $\lambda' = |v'_d| + c_{f,d}$ is the characteristic speed in the $\hat{e}_d$ direction.

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

10.2.1 Using the Module

The FARGO-MHD module is implemented in the directory 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`:

- *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 using the **`FARGO_SetVelocity()`** function that can be implemented in your init.c. This must be the default if FARGO is used together with the shearing box module.

Initial and boundary conditions are assigned as usual by prescribing the *total* velocity and not the residual. Likewise, output files are always written using the total velocity and not the residual.

The order of reconstruction used during the linear transport step is set by the constant `FARGO_ORDER` which, by default, is 3 (third-order PPM). The default value of the three switches `FARGO_ORDER`, `FARGO_NSTEP_AVERAGE` and `FARGO_AVERAGE_VELOCITY` can be changed inside your definitions.h, see §2.3.

The FARGO-MHD is typically used to model supersonic accretion disks and test problems can be found in the directory Test_Problems/HD/Disk_Planet/ (configurations #2, #4 and #6) as well as in Test_Problems/MHD/FARGO/Spherical_Disk/. For more information see the test problem documentation at Doc/test_problems.html).

10.2.2 A Note on Parallelization

The FARGO-MHD algorithm is 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 denoted with m.

With a large number of processors ($\gtrsim 2048$), the resulting auto-decomposition mode may result in sub-domains that violate this condition and an error message is issued. To avoid this problem you can specify the parallel decomposition manually using the `-dec n1 [n2] [n3]` command line argument (§1.4.2) 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
```

10.3 High-order Finite Difference Schemes

An alternative to the Finite Volume (FV) methodology presented in the previous chapters and to the reconstruction algorithms described in Chapter 2 is the employment 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 `RECONSTRUCTION` will now be

- *LIMO3_FD*: third-order reconstruction of [ČT09];
- *WENO3_FD*: an improved version of the classical third-order WENO scheme of [JS96] based on new weight functions designed to improve accuracy near critical points [YC09];
- *WENOZ_FD*: improved WENO5 scheme proposed by [BCCD08];
- *MP5_FD* : the monotonicity preserving scheme of [SH97] 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* (special relativistic counterparts are not yet implemented).
- In the case of the *MHD* module, only cell centered magnetic fields are 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 to 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 [MTB10] to outperform traditional second-order TVD schemes in terms of reduced numerical dissipation and faster convergence rates for problem involving smooth flows. Figure 10.2 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.

10.3.1 WENO schemes

The WENO schemes are based on the essentially non-oscillatory (ENO) schemes, originally developed by [HEOC87] using a finite volume formulation and later improved by [SO89] 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 [LOC94] for 1-D finite volume formulation, WENO schemes were then implemented in multi-dimensional FD by [JS96], optimizing the original weighing for accuracy.

Currently, the available WENO schemes in **PLUTO** are the 5th order WENOZ of [BCCD08] which improves over the original one [JS96] 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 [YC09]. More details can be found in the paper by Mignone, Tzeferacos & Bodo [MTB10].

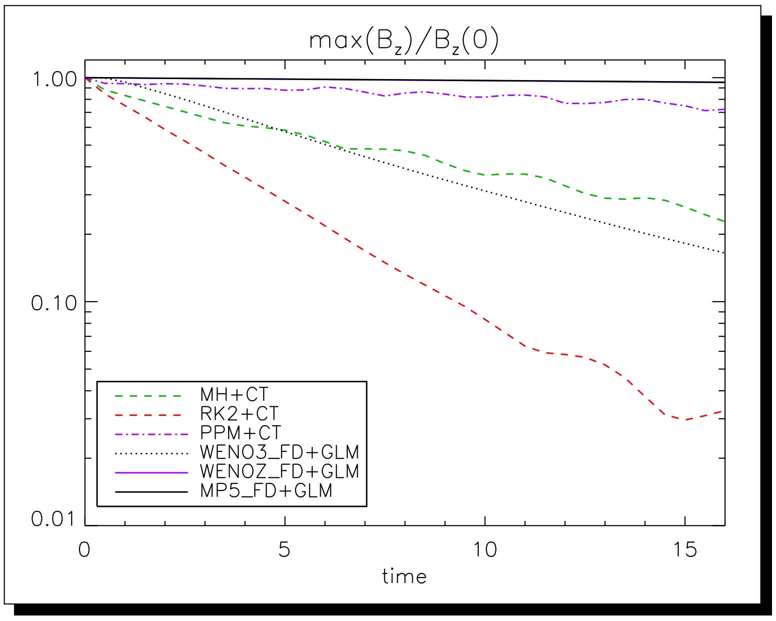


Figure 10.2: Long term (numerical) decay of a circularly polarized Alfven wave on a 2D periodic domain with [120 $\times$ 20] zones. The different curves plot the maximum value of B_z as a function of time and thus give a measure of the intrinsic numerical dissipation. Selected finite volume schemes 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.

10.3.2 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 [ČT09]. 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 [SH97], 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 [MTB10].

10.4 The Forced Turbulence Module

The continuous driving force term is now included with both the hydro-dynamical and magneto-hydro-dynamical module of the **PLUTO** code. Such a driving force term helps to understand the statistical properties of turbulence (i.e., the Probability Distribution Function (PDF), power spectrum, structure functions etc). The driving force denoted by $\boldsymbol{F}^{\rm turb}(\boldsymbol{x},t)$ is added to both the total momentum and total energy equation of the standard HD and MHD module equations. For example, in case of HD module, these equations are given by -

$$\begin{aligned}
\frac{\partial \boldsymbol{m}}{\partial t} + \nabla\cdot\left[\boldsymbol{m}\boldsymbol{v} + \mathsf{I}p\right]^T &= \rho\boldsymbol{F}^{\rm turb} && (10.6)\\
\frac{\partial E_t}{\partial t} + \nabla\cdot\left[(E_t+p)\boldsymbol{v}\right]^T &= \boldsymbol{m}\cdot\boldsymbol{F}^{\rm turb}\,, && (10.7)
\end{aligned}$$

where $\boldsymbol{m} = \rho\boldsymbol{v}$ is the momentum flux, E_t is the total energy i.e., sum of kinetic and internal energy and p is the total gas pressure.

10.4.1 Time And Space varying Random Forcing

We model $\boldsymbol{F}^{\rm turb}$ as a time and space varying function to achieve a smoothly varying pattern that resembles the flow of kinetic energy from scales larger than the simulation box scale. For this purpose, we have used the Ornstein-Uhlenbeck (OU) process. The OU process is essentially a well-defined stochastic process with a finite auto-correlation timescale. It can be used to excite turbulent motions in all dimensions. The time-correlation is important for modeling realistic driving forces as it will lead to coherent structures in the simulation that will be absent with white-noise forcing. For instance, large-scale driving forces like turbulent stirring from larger scales will be correlated on timescales related to the lifetime of an eddy on the scale of the simulation domain.

The OU process is a stochastic differential equation governing the evolution of the driving force, $\boldsymbol{F}^{\rm turb}$ in Fourier domain -

$$d\boldsymbol{F}^{\rm turb}(\boldsymbol{k},t) = \boldsymbol{F}_0^{\rm turb}(\boldsymbol{k})\mathcal{P}^{\zeta}(\boldsymbol{k})d\mathcal{W}(t) - \boldsymbol{F}^{\rm turb}(\boldsymbol{k},t)\frac{dt}{T}. \tag{10.8}$$

The first term on the right hand side of the Eq. 10.8 represents the diffusion term which is modeled using the Wiener process $\mathcal{W}(t)$, which is a random process and defined as

$$\mathcal{W}(t) - \mathcal{W}(t-dt) = \boldsymbol{\mathcal{N}}(0,dt) \tag{10.9}$$

where, $\boldsymbol{\mathcal{N}}$ is a dimension dependent normal distribution with zero mean and standard deviation given by dt. Quantitatively this process adds a Gaussian random increment to the vector force field of the previous time step dt. The projection tensor $\mathcal{P}^{\zeta}(\boldsymbol{k})$ in Fourier space is given by -

$$\mathcal{P}^{\zeta}_{ij}(\boldsymbol{k}) = \zeta\mathcal{P}^{\perp}_{ij}(\boldsymbol{k}) + (1-\zeta)\mathcal{P}^{\parallel}_{ij}(\boldsymbol{k}) = \zeta\delta_{ij} + (1-2\zeta)\frac{k_i k_j}{k^2} \tag{10.10}$$

where δ_{ij} is the Kronecker delta function and $\mathcal{P}^{\perp}_{ij}(\boldsymbol{k}) = \delta_{ij} - k_i k_j/k^2$ and $\mathcal{P}^{\parallel}_{ij}(\boldsymbol{k}) = k_i k_j/k^2$ are the solenoidal (divergence-free) and compressive (curl-free) projection operators respectively. The parameter $0 \le \zeta \le 1$ governs the type of forcing to be incorporated (see Federath et. al. 2010).

The second term on the right-hand side of Eq 10.8 is a drift term describing the exponential decay of the auto-correlation of $\boldsymbol{F}^{\rm turb}$. Typically, the auto-correlation timescale is set equal to the turbulent crossing time at the scale of peak energy injection i.e., $T = L_{\rm peak}/V$. This models the kinetic energy input from large-scale turbulent fluctuations and its breaking up into smaller and smaller structures.

10.4.2 Setting up Forced Turbulence

The user can set the forced turbulence module by setting the `FORCED_TURB` flag to `YES` during the python setup script. The different user-defined constants for this module that would be required should be specified in definitions.h.

Based on the energy injection scales chosen by the setting FORCED_TURB_KMIN and FORCED_TURB_KMAX as the minimum and maximum energy input wavevectors, total number of modes N_k based on dimension are calculated initially. Note the wave-vectors are in units of $1/L_{\rm box}$, so to inject between the scales equivalent to $L_{\rm box}/3$ and $L_{\rm box}$, the values of FORCED_TURB_KMIN and FORCED_VAR_KMAX has to be set to 2π and $3 \times 2\pi = 6\pi$ respectively.

Subsequently for each mode, at regular intervals in number of steps governed by FORCED_TURB_FREQ, six separate phases (real and imaginary in each of the three spatial dimensions) are evolved by an Ornstein-Uhlenbeck (OU) random process. As described above, this random process decays the previous value of the force by an exponential factor given by $\exp(-dt/T)$ and then adds a Gaussian random variable with a given variance and weighted by a *driving* factor - $\sqrt{1 - \exp(-dt/T)\exp(-dt/T)}$. The variance is chosen to be the square root of the specific energy input rate (set by FORCED_TURB_ENERGY) divided by the decay time (FORCED_TURB_DECAY). By evolving these 6 phases of each of the modes for a given value of ζ (set by FORCED_TURB_WEIGHT), one can obtain the acceleration in Fourier space which are further converted to physical space by a direct Fourier transform - i.e., adding explicitly the sine and cosine terms.

11. Particles

The static version of **PLUTO** comes with a fully parallel module supporting different kinds of particles. The module can be enabled by invoking the python script with the `--with-particles` option while the particle type can be selected by adding, in the user-defined constants section of definitions.h, the following line:

```
#define PARTICLES_TYPE   <type>
```

where *<type>* is one among:

- *COSMIC_RAYS* (see §11.2)
- *LAGRANGIAN* (or tracer particles, see §11.3, Default)
- *DUST* (under development)

Particles are described by the `Particle` structure and stored into memory using a doubly linked list, consisting of sequentially linked node structures as in Fig. 11.1. Each node contains the particle itself and pointers to the previous and to the next node in the sequence. In a linked list, elements can be inserted or removed in a straightforward way and shuffling operations can be easily performed by changing pointers.

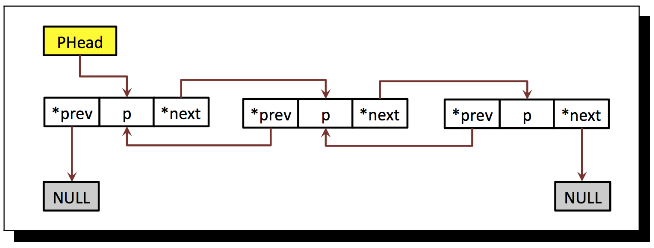


Figure 11.1: Doubly linked list structure. Here `p` represents a `Particle` structure.

As an example, in order to count the number of particles in the list, we use the following piece of code:

```
  particleNode* CurNode = d->PHead;

  int cnt = 0;
  while(CurNode != NULL){
    cnt++;
    CurNode = CurNode->next;
  }
```

where `d->PHead` contains the starting node.

A particle structure may contain several fields, depending on the selected particle type. While different particle types have different structure members, some of these are common to all types. Table 11.1 described the most important fields.

The *[Particles]* block in pluto.ini (see §4.8) controls both the number of particles to be initialized and the output frequency.

Field	Particle	Description
`p->coord[]`	All	A 3 element array (in double precision) giving the particle coordinates. Supplied by the user.
`p->speed[]`	All	A 3 element array (in double precision) giving the particle velocities. Supplied by the user (except when `PARTICLES_TYPE` == *`LAGRANGIAN`*).
`p->tinj`	All	The creation (or injection) time. Set by the code.
`p->color`	All	A free scalar typically used to label particles (e.g. depending on the region they were created). Supplied by the user.
`p->id`	All	An integer to uniquely identify the particle once it is created. Identities increase sequentially as particles are created. Set by the code.
`p->rho`	*CR*	The mass density of a single CR particle. Se by the user.
`p->ca`	*LP+S*	A parameter that controls losses to adiabatic expansion (*`double`*).
`p->cr`	*LP+S*	A parameter that controls losses to synchrotron/IC process (*`double`*).
`p->cmp_ratio`	*LP+S*	The compression ratio at shocks (*`double`*).
`p->nmicro`	*LP+S*	Total number of micro-particles per unit volume (*`double`*).
`p->shkflag`	*LP+S*	A flag that determines if the particle is inside the shock or not (*`int`*).
`p->shk_gradp`	*LP+S*	Gradient of pressure while the particle is crossing a shock (*`double`*).
`p->shk_vL[]`	*LP+S*	An array of primitive variables computed just before the particle enters the shocked zones.
`p->shk_vR[]`	*LP+S*	An array of primitive variables computed just after the particle enters the shocked zones.
`p->eng[]`	*LP+S*	An double-precision array containing values of spectral energies in code units. The number of array elements is set by the constant `PARTICLES_LP_NEBINS` sets the total number of bins (= 100 by default). To obtain the values of energies in physical units each energy bin has to be multiplied with `PARTICLES_LP_SPEC_ENERGY` specified in ergs. Its default value is 0.01 ergs.
`p->chi[]`	*LP+S*	An array that represents the number of micro-particles (leptons) per unit volume distributed (normalized to the fluid density) in the energy bins (*`double`*). The number of elements if set by the constant `PARTICLES_LP_NEBINS` (100 by default). For example, for a power-law spectra its an array of $\mathcal{N}(E) \propto E^{-s}$, where E is the spectral energy.

Table 11.1: Particle structure members. Here `p` is intended as a pointer to a particle structure while "All", "CR" and "LP+S" refers to fields that are available, respectively, for all particles, *`COSMIC_RAYS`* and *`LAGRANGIAN`* with spectra.

11.1 Initial, Boundary and Injection conditions

Particles initial and boundary conditions as well as injection criteria can be specified using the functions available in PLUTO/Src/Particles/particls_init.c (a local copy of the file is strongly recommended).

11.1.1 Initial Condition

The function **Particles_Init()** allows the user to specify the particles' initial coordinates and velocities inside the computational domain. Note that *LAGRANGIAN* particles do not require velocity to be set.

There are two possible initialization methods: *global* or *cell-by-cell*, which can be enabled from your pluto.ini initialization file (§4.8).

1. **Global Initialization.** In a global initialization N_{glob} is the global number of particles to be distributed in the computational domain. The following example provides a simple uniform initialization using the *global* method:

```
void Particles_Init(Data *d, Grid *grid)
{
  int i,j,k, np;
  int np_glob = RuntimeGet()->Nparticles_glob;
  static int first_call = 1;
  Particle p;

  if (first_call) RandomSeed(time(NULL),0);  /* Seed random number */
  first_call = 0;

  if (np_glob > 0){

    for (np = 0; np < np_glob; np++){

    /* -- Spatial distribution -- */
      Particles_LoadUniform(np, np_glob, grid->xbeg_glob, grid->xend_glob, p.coord);

    /* -- LAGRANGIAN particles -- */
      #if (PARTICLES_TYPE == LAGRANGIAN) && (PARTICLES_LP_SPECTRA == YES)
      Particles_LP_InitSpectra(&p);
      #endif

    /* -- COSMIC_RAY particles -- */
      #if PARTICLES_TYPE == COSMIC_RAYS
      p.speed[IDIR] = RandomNumber(-1,1);
      p.speed[JDIR] = RandomNumber(-1,1);
      p.speed[KDIR] = RandomNumber(-1,1);
      p.rho         = 2.e-3;
      #endif
      p.color = 0.0;
      Particles_Insert (&p, d, PARTICLES_CREATE, grid);
    }
  }
  Particles_SetID(d->PHead);
}
```

The function **Particles_LoadUniform()** attempts to place uniformly `np_glob` particles inside the computational domain. This variable is read from the initialization file pluto.ini and it is recalled using the function **RuntimeGet()**. The function **Particles_LP_InitSpectra()** is used to initialize, only in the case of *LAGRANGIAN* particles, the initial spectral distribution. The function **RandomNumber()** generates uniform deviates in the range specified by the arguments and it is used, in the above example, to initialize the velocity of CR particles[1]. When all of the attributes have been set, the particle is finally inserted into the linked list using **Particles_Insert()**. The last function call to **Particles_SetID()** is optional and serves to set a unique id to newly created particles.

Spatial and/or velocity distributions may also be initialized to follow an assigned probability distribution function. This is achieved using the **Particles_LoadRandom()** function which is based on a simple acceptance-rejection method.

[1] The prng sequence is seeded through the call to **RandomSeed()**. **PLUTO** provides some pseudo random-number functionality in Src/Math_Tools/math_random.c, plase consult the documentation in Doc/math_tools.html.

As an example, consider placing N particles following a 2D Gaussian distribution on a Cartesian grid. You may then call **`Particles_LoadRandom()`** by passing, as an argument, a reference to the function **`Particles_SpaceDistrib()`** which implements the desired Gaussian distribution:

```
void Particles_Init(...)
{
  ...

    for (np = 0; np < np_glob; np++){
      ...
      Particles_LoadRandom (xbeg, xend, Particles_SpaceDistrib, p.coord);
      ...
    }
  ...
}

double Particles_SpaceDistrib(double x, double y, double z)
{

/* -- Gaussian distribution -- */

  double mu_x = 2.5;
  double mu_y = 1.0;
  double sigma_x  = 0.5;
  double sigma_y  = 0.25;
  double exp_fact =   0.5*(x-mu_x)*(x-mu_x)/(sigma_x*sigma_x)
                    + 0.5*(y-mu_y)*(y-mu_y)/(sigma_y*sigma_y);
  return 1.0*exp(-exp_fact);
}
```

This example is taken from Test_Problems/Particles/LP/Planar_Shock.

2. **Cell-by-Cell Initialization** Conversely, in a cell-by-cell initialization, $N_{\rm cell}$ particles are assigned to each cell, resulting in a total of $N_{\rm cell} \times N_x \times N_y \times N_z$ particles. An example suitable for CR particles:

```
void Particles_Init(Data *d, Grid *grid)
{
  int i,j,k, np, dir;
  int np_cell = RuntimeGet()->Nparticles_cell;
  static int first_call = 1;
  double xbeg[3], xend[3];
  Particle p;

  if (first_call) RandomSeed(time(NULL),0);
  first_call = 0;

  if (np_cell > 0){
    DOM_LOOP(k,j,i){

    /* -- Get cell size -- */

      xbeg[IDIR] = grid->xl[IDIR][i]; xend[IDIR] = grid->xr[IDIR][i];
      xbeg[JDIR] = grid->xl[JDIR][j]; xend[JDIR] = grid->xr[JDIR][j];
      xbeg[KDIR] = grid->xl[KDIR][k]; xend[KDIR] = grid->xr[KDIR][k];

    /* -- Loop on particles -- */

      for (np = 0; np < np_cell; np++){

        Particles_LoadUniform(np, np_cell, xbeg, xend, p.coord);

        double T0 = ...; /* Plasma temperature in code units (p/rho) */
        p.speed[0] = GaussianRandomNumber(0.0, sqrt(T0));
        p.speed[1] = GaussianRandomNumber(0.0, sqrt(T0));
        p.speed[2] = GaussianRandomNumber(0.0, sqrt(T0));
        p.rho      = 1.e-3/np_cell;
        p.color = (p.coord[0] > 0.0 ? 1.0:-1.0);
        Particles_Insert (&p, d, PARTICLES_CREATE, grid);
      }
    }
  }
  Particles_SetID(d->PHead);
}
```

Again, the function **`Particles_LoadUniform()`** is used to place `np_cell` particles uniformly inside the cell (rather than the whole computational domain).

The function **`GaussianRandomNumber()`** generates random deviates from a Gaussian distribution with mean μ and standard deviation σ (first and second argument, respectively), see the documentation in Src/Math_Tools/math_random.c. The final result is a Maxwell-Boltzmann distribution seen as the product of three independent normally distributed variables v_x, v_y and v_z with variance k_BT/m_p, where m_p is the CR particle mass. If m_f is the mass of the particles composing the fluid, a convenient normalization from c.g.s to code units can be easily recovered as

$$f(v_x)dv_x = \sqrt{\frac{m_p}{2\pi k_B T}} \exp\left(-\frac{m_p v_x^2}{2k_B T}\right) dv_x = \sqrt{\frac{1}{2\pi\sigma^2}} \exp\left(-\frac{(v_x/v_0)^2}{2\sigma^2}\right) d\left(\frac{v_x}{v_0}\right) \tag{11.1}$$

where $\sigma^2 = m_f T_0/m_p$ is the variance, T_0 is some average plasma temperature computed as p/ρ (p and ρ are in code units) and v_x/v_0 is the fluid velocity in code units ($v_0 \equiv$`UNIT_VELOCITY`).

A similar, although slower, initialization could have been done using **`Particles_LoadRandom()`**:

```
void Particles_Init(Data *d, Grid *grid)
{
   ...
  if (np_cell > 0){
    DOM_LOOP(k,j,i){
      Particles_LoadUniform(np, np_cell, xbeg, xend, p.coord);
      Particles_LoadRandom(vbeg, vend, Particles_VelocityDistrib, p.speed);
      p.rho = 1.e-3/np_cell
      ...
    }
  }
}

double Particles_VelocityDistrib(double vx, double vy, double vz)
{
  double s2 = T0 = ...;
  double fvx, fvy, fvz;
  double fact = 1.0/sqrt(2.0*CONST_PI*s2);

  fvx = fact*exp(- (vx*vx)/(2.0*s2));
  fvy = fact*exp(- (vy*vy)/(2.0*s2));
  fvz = fact*exp(- (vz*vz)/(2.0*s2));

  return fvx*fvy*fvz;
}
```

The acceptance-rejection method is usually slower than using directly **`GaussianRandomNumber()`** to generate Gaussian distributed deviates, but it is more general and may be useful for different distributions.

11.1.2 Boundary Conditions

At present, boundary conditions used for the fluid also apply to particles. If a *`userdef`* value is given in the standard boundary block (§4.5), particles may be created or destroyed at the specified boundary and the user must code his/her boundary condition using the function **`Particles_UserDefBoundary()`** located in the file particles_init.c. This function is also called by **PLUTO** in the case of an *`outflow`* boundary which requires particles to be deleted once they cross an *`outflow`* boundary.

11.1.3 Injection

The function **`Particles_Inject()`** is used to create (inject) new particles inside a particular area of the computational domain at specific times during your simulation. Injection requires the user to specify the field members used during the initialization process.

In the example below, injection takes place every $\Delta t_{\rm inj}$ (which is assigned through a user-defined parameter `g_inputParam[INJECT_FREQ]`) but only in a specified region of the domain, selected through a user supplied function **`InjectZone()`** which returns either 1 or 0. Particle velocity is initialized using a Gaussian ditribution:

```
void Particles_Inject(Data *data, Grid *grid)
{
  int     i,j,k, status, np;
  int     n0, n1;
  int     np_cell = RuntimeGet()->Nparticles_cell;
  double  t0, t1, t_freq;
  double *x1 = grid->xgc[IDIR];  /* -- array pointer to x1 coordinate -- */
  double *x2 = grid->xgc[JDIR];  /* -- array pointer to x2 coordinate -- */
  double *x3 = grid->xgc[KDIR];  /* -- array pointer to x3 coordinate -- */
  Particle p;

  t0      = g_time;
  t1      = g_time + g_dt;
  t_freq  = g_inputParam[INJECT_FREQ];
  n0      = (int)(t0/t_freq);
  n1      = (int)(t1/t_freq);

  if (n0 != n1 || g_stepNumber == 0){
    print (">>␣Injection\n");
    DOM_LOOP(k,j,i){
      if ( InjectZone(x1[i],x2[j],x3[k],grid) ){

        for (np = 0; np < np_cell; np++){  /* Loop on particles/cell */

          double sigma = sqrt(1.0/g_gamma);
          p.coord[0] = x1[i];
          p.coord[1] = x2[j];
          p.coord[2] = x3[k];

          p.speed[IDIR] = GaussianRandomNumber(0.0, sigma);
          p.speed[JDIR] = GaussianRandomNumber(0.0, sigma);
          p.speed[KDIR] = GaussianRandomNumber(0.0, sigma);

          p.rho   = 1.e-3/np_cell;
          p.color = prank;
          status  = Particles_Insert(&p, data, PARTICLES_CREATE, grid);

          if (status != TRUE){
            print ("!␣Particles_Inject():␣error\n");
            QUIT_PLUTO(1);
          }
        }  /* End for (np...) */
      }  /* end if (InjectionZones() */
    } /* end DOM_LOOP */
    Particles_SetID(data->PHead);
  } /* end if (injection time) */}
```

Beware that **PLUTO** calls this function at every time step and care must be taken in order to avoid overpopulating the domain with unwanted particles.

11.2 Cosmic Ray Particles

Cosmic rays (CR) are enabled by setting `PARTICLES_TYPE` to *`COSMIC_RAYS`* inside the user-defined constant section of your definitions.h.

The Cosmic Rays particle module may be used to describe the the dynamical interaction between a thermal plasma and a non-thermal population of collisionless cosmic rays (CR henceforth). This is the MHD-PIC model first presented by [BCSS15] and the PLUTO code implementation can be found in the method paper [MBVM18]. While the fluid equations are treated using the methods already available for shock capturing MHD, CR particles are evolved with conventional PIC techniques. This formalism aims at capturing the kinetic effects of CR particles without the need to resolve the plasma skin depth, as it is typically required by PIC codes. In the MHD-PIC formalism, instead, only the Larmor (gyration) scale must be adequately resolved,

CR particles obey the equation of motion

$$\begin{aligned}
\frac{d\boldsymbol{x}_p}{dt} &= \boldsymbol{v}_p \\
\frac{d(\gamma\boldsymbol{v})_p}{dt} &= \left(\frac{e}{mc}\right)_p (c\boldsymbol{E} + \boldsymbol{v}_p \times \boldsymbol{B})
\end{aligned} \tag{11.2}$$

where $\gamma = 1/\sqrt{1-\boldsymbol{v}_p^2/\mathbb{C}^2}$ is the particle Lorentz factor while p is an index labeling the particle. Since the actual speed of light does not explicitly appears in the MHD equation, here $\mathbb{C}$ (=`PARTICLES_CR_C`) is used to specify an artificial value for the speed of light and, for consistency reasons, it must be greater than any characteristic signal velocity. The ratio $(e/mc)_p$ (=`PARTICLES_CR_E_MC`) specifies the charge to mass ratio for CR particles.

When feedback is included, CR particles extert a feedback on the thermal plasma introducing modifications in the MHD equations:

$$\begin{aligned}
\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}_g) &= 0 \\
\frac{\partial(\rho\boldsymbol{v}_g)}{\partial t} + \nabla\cdot[\rho\boldsymbol{v}_g\boldsymbol{v}_g - \boldsymbol{B}\boldsymbol{B} + \mathsf{I}p_t] &= -\boldsymbol{F}_{\rm CR} \\
\frac{\partial\boldsymbol{B}}{\partial t} + \nabla\times(c\boldsymbol{E}) &= \boldsymbol{0} \\
\frac{\partial E_t}{\partial t} + \nabla\cdot\left[\left(\frac{1}{2}\rho\boldsymbol{v}_g^2 + \frac{5}{2}p_g\right)\boldsymbol{v}_g + c\boldsymbol{E}\times\boldsymbol{B}\right] &= -\boldsymbol{v}_g\cdot\boldsymbol{F}_{\rm CR}
\end{aligned} \tag{11.3}$$

where $\boldsymbol{v}_g$ is the usual gas velocity. The previous system of conservation laws is known as the MHD-PIC equations.

In the previous equations, $\boldsymbol{F}_{\rm CR}$ is the force density felt by the CR particles and the minus sign is a simple manifestation of Newton 3rd law, stating that the opposite force is felt by a fluid element:

$$\boldsymbol{F}_{\rm CR} = (1-R)\left(q_{\rm CR}\boldsymbol{E}_0 + \frac{1}{c}\boldsymbol{J}_{\rm CR}\times\boldsymbol{B}\right) \tag{11.4}$$

where $R = q_{\rm CR}/(q_i + q_{\rm CR})$ while $c\boldsymbol{E}_0 = -\boldsymbol{v}_g \times \boldsymbol{B}$. The CR force is computed by depositing charges and currents from particles to grid, in the **`Particles_CR_Force()`** function:

$$\left(\frac{q_{\rm CR}}{c}\right)_{\rm i} = \sum_p W(\boldsymbol{x}_{\rm i} - \boldsymbol{x}_p)\frac{e}{mc_p}\varrho_p \qquad \left(\frac{\boldsymbol{J}_{\rm CR}}{c}\right)_{\rm i} = \sum_p W(\boldsymbol{x}_{\rm i} - \boldsymbol{x}_p)\frac{e}{mc_p}\varrho_p\boldsymbol{v}_p \tag{11.5}$$

where ϱ_p (=`p->rho`) and $\boldsymbol{v}_p$ (=`p->speed[]`) are the density and velocity of a single particle while $W()$ are weight functions. The charge density of the fluid follows from $q_i/c = (e/mc)_g\rho$ where $(e/mc)_g$ (=`PARTICLES_CR_E_MC_GAS`) is the charge to mass ratio for the fluid. The total electric field is then computed as

$$c\boldsymbol{E} = -\boldsymbol{v}_g\times\boldsymbol{B} - R(\boldsymbol{v}_{\rm CR} - \boldsymbol{v}_g)\times\boldsymbol{B} = c\boldsymbol{E}_0 - \frac{c\boldsymbol{F}_{\rm CR}}{q_i} \tag{11.6}$$

where the second term is the CR-Hall term. The second expression is used in the code.

The MHD-PIC module works under the restrictions listed in Table 11.2 (only Cartesian geometry). Also, units are most conveniently normalized by taking the ion skin depth c/ω_{pi} as the reference length see Section 4 in [MBVM18].

PHYSICS	*Feedback*	GEOMETRY	TIME_STEPPING	DIM. SPLIT
MHD	*YES/NO*	*CARTESIAN*	*HNCK/ChTr/RK2*	*NO*
RMHD	*NO*	*CARTESIAN*	*HNCK/RK2*	*NO*

Table 11.2: Algorithm compatibility for the MHD-PIC module. Here *Feedback* is a short-hand notation for PARTICLES_CR_FEEDBACK.

11.3 Lagrangian Particles

Lagrangian particles (LP) follow the fluid streamlines such that its position coordinates $\boldsymbol{x}_p$ are evolved in time using the following equation -

$$\frac{d\boldsymbol{x}_p}{dt} = \boldsymbol{v}_p = \boldsymbol{v}_{\rm fluid}(\boldsymbol{x} \to \boldsymbol{x}_p) \tag{11.7}$$

where, the particle velocity $\boldsymbol{v}_p$ is obtained by interpolating the fluid velocity defined at the center of each cell to the particle's position. LP can be used as simple tracer particles or to model sub-grid physical processes although *excluding* backreaction on the fluid. In the latter case, we have implemented a framework to model non-thermal emission from particles embedded in non-relativistic or relativistic magnetized flows (see next section).

11.3.1 Radiative Losses - LP Model

The main objective of the hybrid framework developed involving macro-particles is to model non-thermal emission from astrophysical flows. Each Lagrangian particle in this framework is regarded as *macro-particle* with a finite energy distribution $N(E)$ which is updated in time by solving the relativistic cosmic ray transport equation based on local fluid conditions. The evolution takes into account radiative losses from adiabatic expansion, synchrotron cooling and losses due to Inverse Compton processes via CMB photons. In addition, the spectral distribution of macro-particles experiencing shocks is updated by considering the process of diffusive shock acceleration (DSA) for both quasi-parallel and quasi-perpendicular shocks, see Vaidya et al. (ApJS 2018, submitted).

To enable the radiative losses module, the user has to modify the definitions.h in the user-defined constant section (§ 2.3) as

```
/* [Beg] user-defined constants (do not change this line) */

#define  PARTICLES_LP_SPECTRA                          YES

/* [End] user-defined constants (do not change this line) */
```

If the previous flas is enabled, other structure members (which we list in Table 11.1) are included in the definition.

11.3.2 Spectral Distribution Initialization

The function **Particles_LP_InitSpectra()** should be used to prescribe the initial spectral profile for eeach macro-particle. The following piece of code, for instance, can be used to initialize all macro-particle distribution to a power-law spectra with $s = 6.0$:

```
/* ********************************************************************* */
void Particles_LP_InitSpectra(Particle* pl)
/*!
 *  Initialize spectra for each particle (only for LAGRANGIAN).
 *  Specify here the initial distribution of N(E) with E for each particle
 *
 *  \param [in]      pl      Pointer to the Particle structure.
 *
 ********************************************************************* */
{
  int i;
  double Emin, Emax, N_0, s;
  double lnEmin, lnEmax, dlnE, scrh;
  Emin = 1.0e-2; /* In code units : For physical value multiply by PARTICLES_LP_SPEC_ENERGY */
  Emax = 1.0e4; /* In code units : For physical value multiply by PARTICLES_LP_SPEC_ENERGY */
  lnEmin = log10(Emin);
  lnEmax = log10(Emax);
  dlnE = (lnEmax - lnEmin)/((double)PARTICLES_LP_NEBINS-1);

  pl->nmicro = 0.001; /* The number density of micro-particles in units of UNIT_DENSITY/CONST_amu */
  for (i=0; i< PARTICLES_LP_NEBINS; i++){
    scrh        = lnEmin + i*dlnE;
    pl->eng[i] = pow(10.0, scrh);

    /* /\*Single Power Law*\/ */
    s = 6.0;
    N_0 = (pl->nmicro)*(1.0-s)/(pow(Emax,1.0-s)-pow(Emin,1.0-s));
    pl->chi[i] = N_0 * pow(pl->eng[i],-s);

    /* Setting the default initial values of other essential members */
    pl->cmp_ratio = 1.0;
    pl->shkflag = 0;
    pl->shk_vL[RHO] = -1.0;
    pl->shk_vR[RHO] = -1.0;
    pl->ca = 0.0;
    pl->cr = 0.0;
}
```

Few parameters are defined to provide finer control on the evolution of macro-particles and the associated non-thermal emission from them. These user defined constants are listed below in Appendix B.3.

As the spectral is evolved at every advection time for each macro-particle, we can use this information to compute the Inverse Compton and synchrotron emissivity along with polarisation (Stokes Parameters) from a single macro-particle at every instant. It is possible to output these emissivities using the userdef_output.c. For example, to output the synchrotron emissivity along with Stoke Q and Stoke U polarisation parameters for an observed frequency of 5 GHz and a given line of sight of 20° with respect to the z-axis along the $x - z$ plane (by default the y axis in the fluid frame and observer's frame are parallel) , we have -

```
#ifdef PARTICLES
void Emiss_Deposit(Particle *p, double *qd);
#endif

/* ***************************************************************** */
void ComputeUserVar (const Data *d, Grid *grid)
/*
 *
 *  PURPOSE
 *
 *    Define user-defined output variables
 *
 *
 *
 ***************************************************************** */
{
  #ifdef PARTICLES
     #if PARTICLES_LP_SPECTRUM == YES
      double ***Dq[3];
      Dq[0] = GetUserVar("J5GHz");
      Dq[1] = GetUserVar("Q5GHz");
      Dq[2] = GetUserVar("U5GHz");
      Particles_Deposit(d->PHead, Emiss_Deposit, Dq, 3, grid);
    #endif
 #endif
}

#if PARTICLES_LP_SPECTRA == YES
/*******************************************************************/
void Emiss_Deposit(Particle *p, double *pr)
/*
 *  The helper function for Particles_Deposit to deposit
 *  emissivity density.
 *
 * * ***************************************************************/
{
#if PHYSICS == MHD || PHYSICS == RMHD
  int i;
  double thetaobs = 20.0; /* Angle of incidence in degree i.e., line of sight angle. */
  double JnVals, StQVals, StUVals;
  double frq = 5.0; /* in GHz */
  Particles_LP_Sync_Emissivity(p, frq, thetaobs, &JnVals, &StQVals, &StUVals);
  pr[0] = JnVals;
  pr[1] = StQVals;
  pr[2] = StUVals;
#endif
}
#endif
```

11.4 Particles Output and Visualization

Similarly to the fluid section (§12.1), particles output datafiles are named as particles.nnnn.ext where nnnn is a four-digit zero-padded integer counting the output number and ext is the corresponding file extension. At present, particles datafiles can be written using single, double-precision format (.flt or .dbl) and VTK format. Particles output is controlled by the `[Particles]` block in pluto.ini, see §4.8.

Float and double precision datafiles begin with an ASCII header file followed by raw binary data. The header file consists of lines beginning with a # character providing important information which can be used to read the file (for visualization purposes or for restart). For instance, the number of particles is specified by `nparticles` while the number and names of the particle structure fields that is written to disk is specified by `nfields` and `field_names`, respectively. Raw binary data follows immediately after the header section and contains a sequence of *`nparticles`* blocks each containing the selected structure fields. In the example below, we show the header file of a .flt file where only particle positions and velocities are being written:

```
# PLUTO 4.3-beta5 binary particle data file
# dimensions      2
# nflux           9
# dt_particles    1.000000e-05
# endianity       little
# nparticles      18432
# idCounter       0
# particletype    3
# precision       single
# time            0.000000e+00
# stepNumber      0
# nfields         6
# field_names     x1  x2  x3  vx1  vx2  vx3
<x1, x2, x3, vx1, vx2, vx3}_1
<x1, x2, x3, vx1, vx2, vx3}_2
<x1, x2, x3, vx1, vx2, vx3}_3
...
<x1, x2, x3, vx1, vx2, vx3}_nparticles
```

Members of the particle structure may be individually selected / deselected for a particular format by calling the function **`SetOutputVar()`** from your userdef_output.c in a similar fashion to the fluid part, see §12.2.1. This function should be called inside the **`ChangeOutputVar()`** function. The following piece of code, for instance, disables writing of the *`color`* and *`id`* structure fields when writing single precision (particle) datafiles and *`tinj`* when writing VTK datafiles:

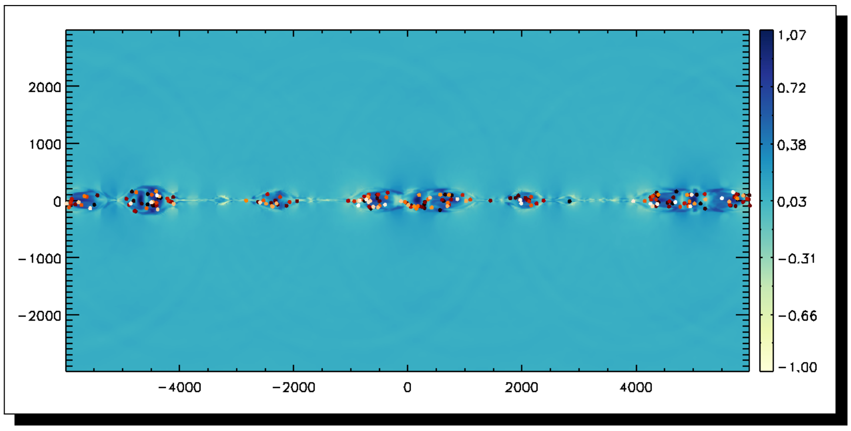


Figure 11.2: Composite image showing fluid density and particles coloured by energy

```
void ChangeOutputVar ()
{
#ifdef PARTICLES
  SetOutputVar ("color", PARTICLES_FLT_OUTPUT, NO);
  SetOutputVar ("id",    PARTICLES_FLT_OUTPUT, NO);
  SetOutputVar ("tinj",  PARTICLES_VTK_OUTPUT, NO);
#endif
}
```

A complete list of structure field names may be found in particles_set_output.c.

Note: Structure fields *should not* be modified in the case of double-precision datafiles if they're used for restarting purposes. Single-precision and VTK datafile, on the other hand, contain only basic members and can be freely cutomized.

11.4.1 Visualization

Particle datafiles can be visualized in IDL, VisIt or pyPLUTO.

11.4.1.1 Visualization with IDL

The procedure PARTICLES_LOAD can be used to read binary of VTK particle datafiles. The following example can be used to display a composite image of both fluid density and particles coloured by energy:

```
IDL> ; -- Display fluid data --
IDL> DISPLAY,x1=x1,x2=x2,alog10(rho),/VBAR, xrange=xrange, yrange=yrange
IDL>
IDL> ; -- Filter particles by energy --
IDL> Ek        = particles.vx1^2 + particles.vx2^2 + particles.vx3^2
IDL> indx      = WHERE(Ek GT MAX(Ek)*0.9)
IDL> particles = particles[indx]
IDL> Ek        = Ek[indx]

IDL> ; -- Filter particles spatially --
IDL> indx = WHERE(particles.x1 GT xrange[0] AND particles.x1 LT xrange[1])
IDL> particles = particles[indx]
IDL> Ek        = Ek[indx]

IDL> ; -- Sort particles --
IDL> particles = particles[SORT(Ek)]
IDL> Ek        = Ek[SORT(Ek)]

IDL> ; -- Overlay  the most 200 energetic particles  --
IDL> LOADCT,/SIL,3
IDL> nparts = 200
IDL> PARTICLES_OPLOT, particles[0:nparts],symsize=1, color=Ek[0:nparts]
```

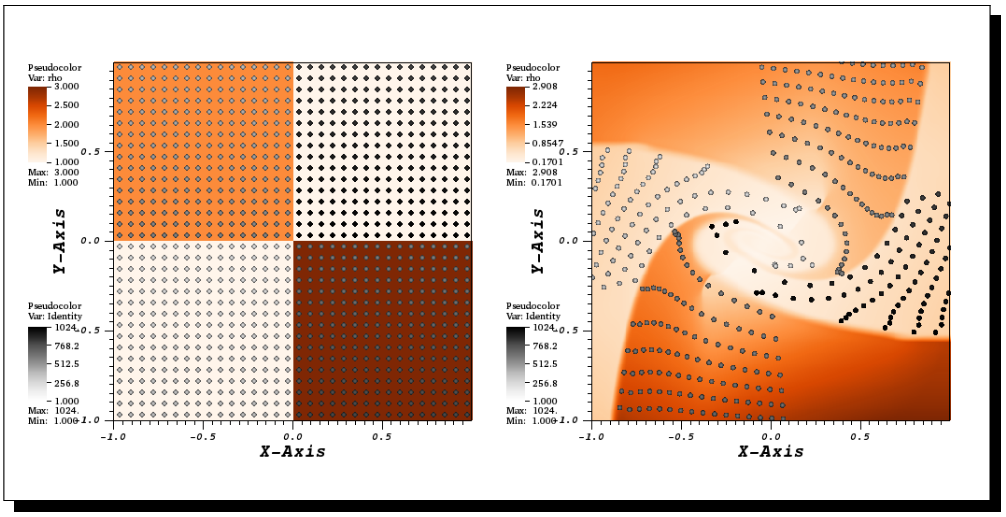


Figure 11.3: A composite image of both 2D fluid density (in color) and particles colored by identity from a binary *vtk* file using the Visit visualization tool. *Left* Initial state and final state (textitright) of the run. Note the Lagrangian particles do not cross the contact discontinuity.

11.4.1.2 Visualization with VisIt

VisIt can be used to visualize particles datafile. If you're overlaying 2D images fluid with particles, the project operator should be applied. The figure 11.3 shows the evolution of fluid density for the 2D Reimann problem in background and particles over-layed that are colored with its identity which is a monotonically increasing set of integers.

11.4.1.3 Visualization with pyPLUTO

The module *ploadparticles* can be used to read binary and VTK particle datafiles. The following example can be used to display a composite image of both 2D fluid density and particles colored by velocity magnitude from a binary *vtk* file:

```
from pylab import *
import pyPLUTO as pp          # importing the pyPLUTO class.

f1 = plt.figure(figsize=[8,8])
ax = f1.add_subplot(111)
ns = 11

P = pp.ploadparticles(ns, datatype='vtk')    # Loading particle data.
PVmag = np.sqrt(P.vx1**2  + P.vx2**2 + P.vx3**2)    # estimating the velocity magnitude
im1 = ax.scatter(P.x1, P.x2, s=10, c=PVmag, cmap=plt.get_cmap('hot'))     # scatter plot
cax1 = f1.add_axes([0.91,0.12,0.03,0.75])
plt.colorbar(im1,cax=cax1)                   # vertical colorbar for particle data.

D = pp.pload(ns, datatype='vtk')             # Load fluid data.
im2 = ax.imshow(D.rho.T, origin='image',extent=[D.x1.min(), D.x1.max(), D.x2.min(), D.x2.max()])  # plotting fluid data.
cax2 = f1.add_axes([0.125,0.92,0.75,0.03])
plt.colorbar(im2,cax=cax2,orientation='horizontal')   #  vertical colorbar for fluid data.


ax.set_xlabel(r'X-axis',fontsize=18)
ax.set_ylabel(r'Y-axis',fontsize=18)
ax.minorticks_on()

plt.axis([0.0,1.0,0.0,1.0])
plt.show()
```

The image obtained after running the above code the problem of Sedov Taylor expansion is shown in figure 11.4

For simulation runs involving spectral evolution, one can structure the code based on the following example to plot the normalized spectral evolution for a particular macro-particle. The spectral information is by default only written in binary files i.e., *dbl* or *flt* files:

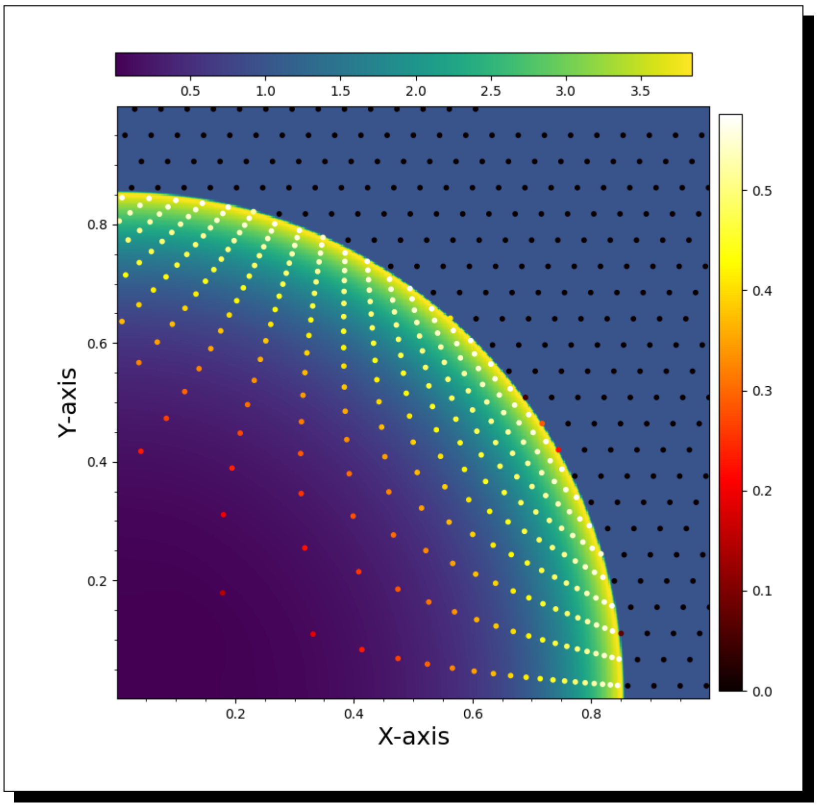


Figure 11.4: A composite image of both 2D fluid density (in color) and particles colored by velocity magnitude from a binary *vtk* file from Sedov-Taylor test problem

```
from pylab import *
from matplotlib import cm
import pyPLUTO as pp

nlast = 35                   # total particlefiles in dbl/flt format
nlines = range(0,nlast+1,5)  # plotting spectrum every 5 file
cols     =  [cm.jet(255*x/(nlast+1)) for x in nlines]
mypid = 12              # choose particles based on its id.
for i in range(len(nlines)):
    P = pp.ploadparticles(nlines[i])
    pineed = np.where(P.id == mypid)[0]
    eps  = P.eng[pineed[0],:]
    neps = P.chi[pineed[0],:]
    plt.loglog(eps, eps*eps*neps/(eps[0]*eps[0]*neps[0]), color=cols[i], lw=2, label=r'$\tau$ = %.2f'%(float(P.time)))

plt.legend()
plt.show()
```

12. 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 popular visualization packages.

12.1 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[1] 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 12.1), nnnn is a four-digit zero-padded integer counting the output number and ext is the corresponding file extension listed above. By default, data files are written in the local working directory unless a different location has been specified using `output_dir` in your pluto.ini, see §4.6. 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	$N_1 \times N_2 \times (N_3 + 1)$
tr1	first tracer	$N_1 \times N_2 \times N_3$

Table 12.1: Base prefix for multiple data set. The size is in units of 4 (for the `flt` format) or 8 (for the `dbl` format) bytes.

[1]Bitmap image format that employs lossless data compression

For each format, it is possible to dump all or just some of the variables. Additional user-defined variables may be written as well, §12.2.0.1. The default setting is described separately for each output in the next subsections and may be changed if necessary, see §12.2.1.

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.
Important: some visualization packages need the information stored in *.out files. We thus strongly recommend to backup these files together with the actual datafiles.

Note: Restart is possible only using the .dbl or .dbl.h5 data formats.

12.1.1 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 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 12.1.

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.

12.1.2 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 §3.2. The file extension is .h5 (and *not* .hdf5 as used by **PLUTO**-Chombo data files, §13.4) and output files are written in Pixie format, a single-block, rectilinear mesh file using HDF5, may be related to the Polar Ionospheric X-Ray Imaging Experiment.

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.

Pixie files can be opened and visualized directly by different softwares, like *VisIt* and *Paraview*. Since we have found compatibility issues with some versions of these visualization softwares, each file comes along with a supplementary .xmf text file in XDMF format that describes the content of the corresponding HDF5 file and can be opened correctly by *VisIt* and *Paraview*, see §12.3.5.

Restart can be performed from double precision HDF5 data files by invoking **PLUTO** with the `-h5restart n` command line option (§1.4.2), 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.

12.1.3 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 while a structured grid format is employed for *POLAR* or *SPHERICAL* geometries. Data is written with the *CELL_DATA* attribute and grid nodes (or vertices) are used to store the mesh.

The following symbolic constants (§2.3) can be used to control some options of the output .vtk files:

- `VTK_TIME_INFO`: when set to *YES*, time information will be added to the header section of the .vtk file. Beware that standard VTK files do not have a specific construct for adding time information and, by doing so, this information will be available only to the VisIt visualisation software (see §12.3.5) which implements a convention where CYCLE and TIME values can be specified as FieldData in the file. If you're using Paraview or other visualisation software different from VisIt, enabling this option will most likely result in a software crash.
- `VTK_VECTOR_DUMP`: by default, all flow quantities (e.g. density, or the x_1 component of velocity) are written with "scalar" attribute as they are. However, by setting `VTK_VECTOR_DUMP` to *YES*, vector fields (such as velocity and magnetic field) can be saved with the "vector" attribute and their components are automatically transformed to Cartesian.

See also Table B.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.

12.1.4 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.

12.1.5 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 colormap. 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 **`ChangeOutputVar()`** in Src/userdef_output.c, see §12.2.1.
Default: By default, only density is written.

12.1.6 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 4.2 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
# ******************************************************
```

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.

12.2 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.

12.2.0.1 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->dx[IDIR]; /* shortcut to dx */
  dy = grid->dx[JDIR]; /* 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];
  }
}
```

PLUTO automatically allocates static memory area for the new variables `T` and `vortz` when calling the **`GetUserVar()`** function. The **DOM_LOOP** 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.

Beware that if the number of `uservar` is reset to zero but the previous function is still executed, a segmentation fault error will occur since user-defined variables (such as `T` and `vortz` in the example above) have not been allocated into memory.

In order to change the default attributes, follow the example in the next subsection.

12.2.1 Changing Attributes

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

To include/exclude a variable from a certain output type, use **`SetOutputVar()`**`(var, type, YES/NO)`. Here "`var`" is a string containing the name of a variable listed in Table 12.1 or an additional one defined in your pluto.ini. The "`type`" argument can take any value among: *DBL_OUTPUT*, *FLT_OUTPUT*, *VTK_OUTPUT*, *DBL_H5_OUTPUT*, *FLT_H5_OUTPUT*, *TAB_OUTPUT*, *PPM_OUTPUT*, *PNG_OUTPUT*. This is a sketch of how this function may be used:

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

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

  image = GetImage ("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 of .dbl.h5 datasets should never be changed since restarting from a given file requires ALL variables being evolved in time.

Table 12.2: Output data formats and supported graphic visualization packages.

File Format	gnuplot	IDL	Mathematica	Paraview	PyPluto	Visit
.dbl	√	√	√	√	√	√
.flt	√	√	√	√	√	√
.vtk	-	√	-	√	√	√
.dbl.h5	-	√	-	√	-	√
.flt.h5	-	√	-	√	-	√
.hdf5	-	√	-	√	√	√
.tab	√	-	-	-	-	-

12.3 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], VisIt[3], ParaView[4], pyPLUTO (§12.3.3), Mathematica[5] and Gnuplot[6]. Table 12.2 show some of the visualization softwares supporting different output formats.

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.

12.3.1 Visualization with Gnuplot

Gnuplot can be used to visualize relatively small or moderately large 1- or 2D datasets written with the tabulated (.tab) or binary data formats (.dbl or .flt)[7]. Gnuplot can be started at the command line by simply typing

```
> gnuplot
```

In the following we give a short summary of the available options while a more detailed documentation can be found in Doc/gnuplot.html.

Ascii Data Files. If you enabled the .tab output format in pluto.ini, you can plot 1D data from, e.g., the first output file by typing

```
gnuplot> plot   "data.0001.tab" u 1:3  title "Density"
gnuplot> replot "data.0001.tab" u 1:5  title "Pressure"  # overplot
```

Here the first column corresponds to the x coordinate, the second column to the y coordinate and flow data values start from the third column. Fig. 12.1 shows the density and pressure profiles for the Sod shock tube problem (conf. #03 in Test_Problems/HD/Sod) using the previous commands.

Two-dimensional ascii datafiles can also be visualized using the `splot` command. Fig. 12.2 shows a simple contour drawing of the final solution of the Mach reflection test problem (remember to enable .tab output) using

```
gnuplot> set contour
gnuplot> set cntrparam level incremental 0.1,0.2,20 # Uniform levels from 0.1 to 20
gnuplot> set view map
gnuplot> unset surface
gnuplot> unset key
gnuplot> splot "data.0001.tab" u 1:2:3 w lines
```

[2]http://www.exelisvis.com/
[3]https://wci.llnl.gov/codes/visit/home.html
[4]http://www.paraview.org/
[5]http://www.wolfram.com
[6]http://www.gnuplot.info
[7]Version 4.2 or higher is recommended.

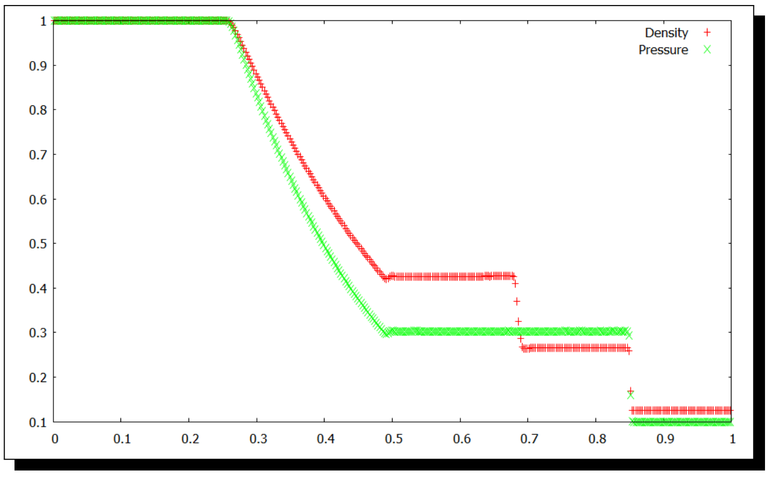


Figure 12.1: Density and pressure plots for the Sod shock tube using Gnuplot.

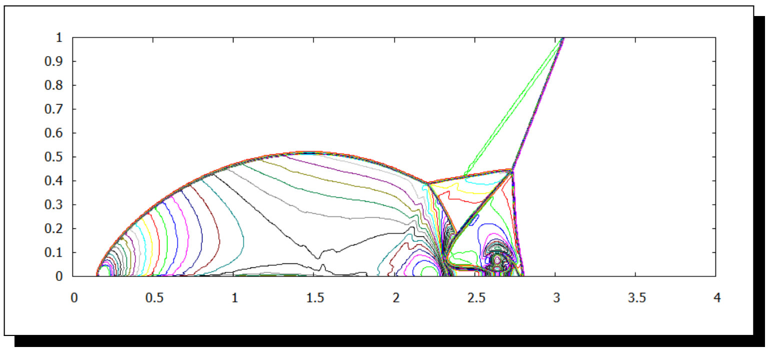


Figure 12.2: Density and pressure plots for the Sod shock tube using Gnuplot.

Binary Data Files. Starting with Gnuplot 4.2, raw binary files are also supported. In this case, grid information (being stored in separate files) must be supplied explicitly through appropriate keywords making the syntax a little awkward.

To ease up this task, one can take advantage of the scripts provided with the code distribution in Tools/Gnuplot. For this, we recommend 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 can also define the loadpath directly from Gnuplot:

```
gnuplot> set loadpath '<pluto_full_path>/Tools/Gnuplot'
```

A typical gnuplot session can be started as follows:

```
gnuplot> load "grid.gp"          # read and store grid information
gnuplot> dsize = 8; load "macros.gp"
gnuplot> load "pm3D_setting.gp"   # set the display canvas for pm3d plot style
```

The first line invokes the grid.gp script which is used to read grid information, the second script sets the data file size (8 or 4 for double or single precision) while the last one initializes a default environment for viewing binary data files using the pm3d style of Gnuplot.

For additional documentation and examples please refer to Doc/gnuplot.html.

12.3.2 Visualization with IDL

IDL (Interactive Data Language) is a vectorized programming language commonly used in the astronomical community for interactive processing of large amounts of data. The **PLUTO** code distribution comes with a number of useful routines written in the IDL programming language to read and visualized data written with **PLUTO** . Several functions and procedures for data visualization and analysis can be found in the Tools/IDL directory which we strongly advise adding to the IDL search path. IDL function documentation can be found in Doc/idl_tools.html.

The PLOAD Procedure The PLOAD procedure can be considered the main driver for reading data stored in one of the following formats: .dbl, .flt, .vtk, .dbl.h5, .flt.h5 or .hdf5. PLOAD requires the information stored in the corresponding data log file (e.g. grid.out, dbl.out, flt.out, etc...) to initialize common block variables and grid information shared by other functions and procedures in the Tools/IDL/ subdirectory. Because of this reason, it should always be called at the very beginning of an IDL session. For example:

```
IDL> PLOAD,3;  Read variables {rho, vx1, ...} from 3rd output .dbl file

IDL> PLOAD,5,/FLOAT, VAR="prs"; Read pressure from 5th output .flt datafile

IDL> ; Read output 9 from the directory "Large_Data/", do not store it
IDL> ; into memory but use IDL file association (preferred for large datasets):
IDL> PLOAD,dir="Large_Data/",9,/ASSOC
```

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, see Doc/idl_tools.html).

When PLOAD is called for the first time, 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`).
- `PLUTO_USERDEF`: (new in **PLUTO** 4.3) an optional common block which can be used to collect user-defined variables. User-defined variables written to disk can be loaded by introducing the the MATCH_USERDEF_VARNAME procedure and manually adding the desired variable names. A typical example is the file userdef.pro (in your local working directory):

  ```
  COMMON PLUTO_USERDEF, flag, vort;   Define the common block

  PRO MATCH_USERDEF_VARNAME, vpt, name, silent=silent
    COMMON PLUTO_GRID
    COMMON PLUTO_VAR
    COMMON PLUTO_RUN
    COMMON PLUTO_USERDEF

    CASE name OF
      "flag":  BEGIN flag = vpt & PRINT,"> Reading ",name & END
      "vort":  BEGIN vort = vpt & PRINT,"> Reading ",name & END
    ELSE:
    ENDCASE

    vpt = 0; Free memory
  END
  ```

  The previous example will add the variables `flag` and `vort` and allow PLOAD to read and store the content in the corresponding variable names. During an IDL session, both PLOAD and userdef.pro must be compiled first (e.g. `IDL> .r pload` followed by `IDL> .r userdef`).

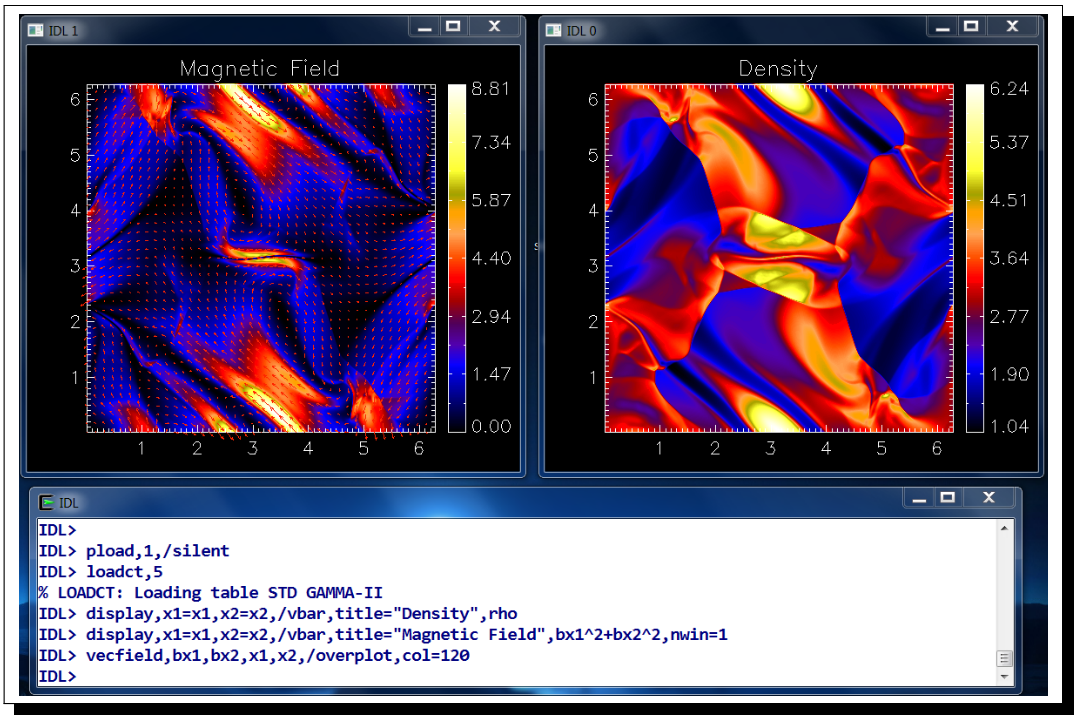
PLOAD can be used inside a normal IDL script, after it has been invoked at least once (or compiled with `.r pload`).

The DISPLAY Procedure DISPLAY is a general-purpose visualization routine and the source code can be found in Tools/IDL/display.pro The DISPLAY procedure can be used to display the intensity map of a 2D variable to a graphic window, e.g.,

```
IDL> PLOAD,3  ; load the third data set in double precision
IDL> DISPLAY,x1=x1,x2=x2,alog10(rho), title='Density',/vbar
IDL> DISPLAY,x1=x1,x2=x2,vx1,title='X-Velocity',nwin=1
```

The second line displays the density logarithm and the third line displays the x_1 component of velocity in a new window.

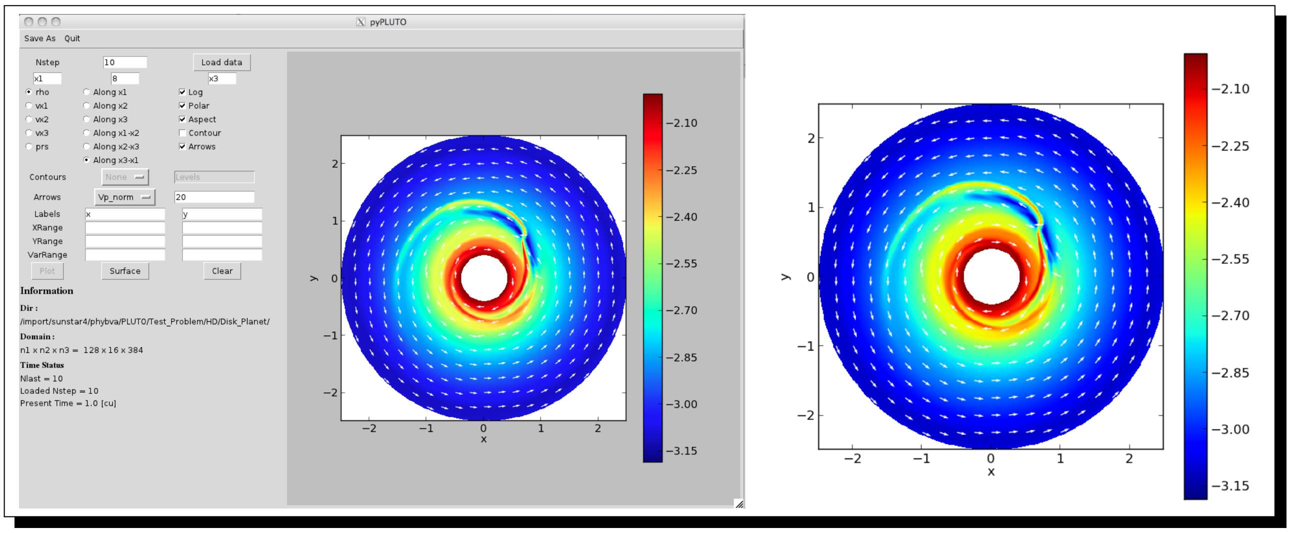
Another example, shown in Fig. 12.3, shows how to visualize magnetic pressure and density in two different windows and overplot the velocity field. For more details, consult Doc/idl_tools.html.



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

12.3.3 Visualization with pyPLUTO



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 12.4: An example of visualization with the pyPLUTO tool.

Binary data files (.dbl, .flt) and VTK files (.vtk) can be visualized using the pyPLUTO code. 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 a quick check, Graphic User Interface (GUI) is provided (requires Python Tkinter). Details of the Installation and Getting Started can be found in the Doc/pyPLUTO.html.

On successful 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 regarding the variable names and their values which are stored as arrays. It also has the respective grid and time 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 visualized using matplotlib, python's plotting library. The pyPLUTO's also provides two classes - Image and Tools. They have some frequently used functions for analysis and data plotting. Details about these classes along with their usage can be found in HTML document referred above.

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 Doc/pyPLUTO.html) and then apply the following commands in the data directory -

```
> GUI_pyPLUTO.py                        # default is for .dbl files
> GUI_pyPLUTO.py --float              # for .flt files
> GUI_pyPLUTO.py --vtk                # for .vtk files
```

Along with the code, an example folder with some sample .py files are provided for certain test problems. The source codes from these files along with their outputs are listed in the HTML documentation.

It is required to first run the respective test problem and generate the data files, after which the user can run the sample .py files as follows from the Tools/pyPLUTO/examples folder:

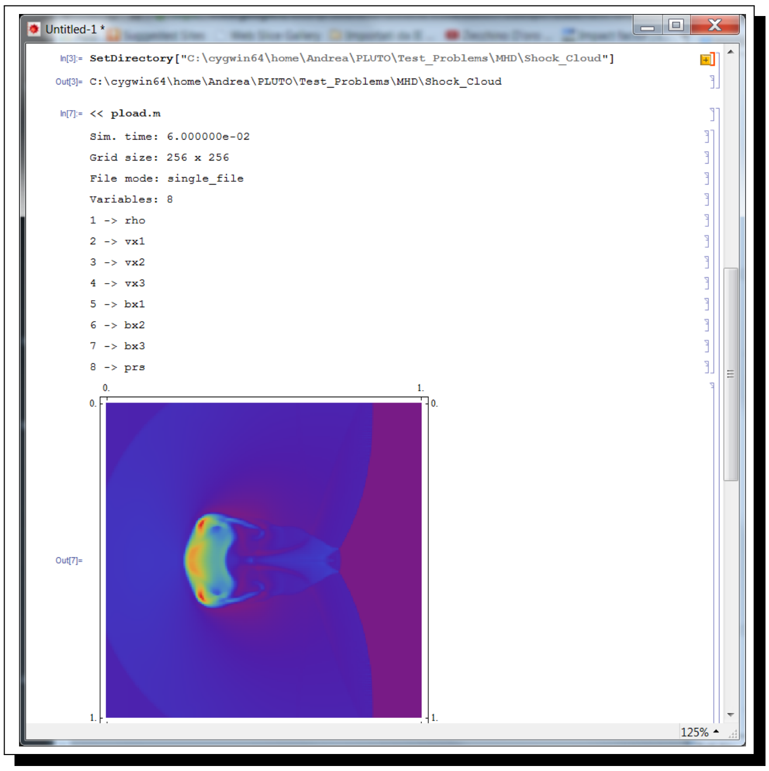
```
> python samplefile.py
```

where the samplefile.py are listed in 12.3,

Table 12.3: List of sample .py files provided in the Tools/pyPLUTO/examples folder

samplefile.py	Test Problem
sod.py	HD/Sod
Rayleigh_taylor.py	HD/Rayleigh_Taylor
stellar_wind.py	HD/Stellar_Wind
jet.py	MHD/Jet
orszag_tang.py	MHD/Orszag_Tang
Sph_disk.py	MHD/FARGO/Spherical_Disk
flow_past_cyc.py	HD/Viscosity/Flow_Past_Cylinder

12.3.4 Visualization with Mathematica



```
In[3]:= SetDirectory["C:\cygwin64\home\Andrea\PLUTO\Test_Problems\MHD\Shock_Cloud"]
Out[3]= C:\cygwin64\home\Andrea\PLUTO\Test_Problems\MHD\Shock_Cloud

In[7]:= << pload.m
        Sim. time: 6.000000e-02
        Grid size: 256 x 256
        File mode: single_file
        Variables: 8
        1 -> rho
        2 -> vx1
        3 -> vx2
        4 -> vx3
        5 -> bx1
        6 -> bx2
        7 -> bx3
        8 -> prs
Out[7]=
```

Figure 12.5: An example of visualization of a binary datafile with mathematica.

PLUTO data files can be displayed with Mathematica[8] using a notebook interface to create an interactive document. A simple reader interface is provided by Tools/Mathematica/pload.m and it can be launched to load and display binary datafiles written in single or double precision (.flt and .dbl). Data is stored into *lists* and can be handled using a variety of built-in functions in Mathematica. Grid and time information are also read from the .out log files and stored into the variables `nx` and `ny` (number of points), `t` (current time level), `nvar` (number of variables), `dt` (current time step).

A typical interactive session once you open an empty book is

```
AppendTo[$Path, ToFileName[{"/home/mignone/PLUTO/Tools/Mathematica"}]]
SetDirectory["/home/mignone/PLUTO/Test_Problems/MHD/Shock_Cloud"]
<< pload.m
```

Please remember to type $\boxed{Shift}$+$\boxed{Enter}$ after each line to make the Wolfram Language process your input. The first line simply adds the Tools/Mathematica directory to the path, the second line changes directory to the working location and the third invokes the reader. Once executed, pload.m reader prompts for the output number, single or double precision and then the variable to display. The output of this session is shown in Fig. 12.5 for the MHD Shock-Cloud interaction test (conf. #01 in Test_Problems/MHD/Shock_Cloud).

The function `ArrayPlot[]` is used to display 2D datasets and is directly included in the interface reader. For instance, to change colormap and visualize pressure in log scale, use

```
ArrayPlot[Log[data[[8]]], DataReversed -> True, ColorFunction -> "RoseColors",
          DataRange -> {{xmin, xmax}, {ymin, ymax}}]
```

For different colormaps, consult http://reference.wolfram.com/language/guide/ColorSchemes.html.

A 1D cut of pressure, for instance, through the x direction at constant y be plotted using

```
ListPlot[data[[8,ny/2]]]
```

The current directory can be displayed by typing `Directory[]` while the home directory can shown by typing `$HomeDirectory`

[8]http://www.wolfram.com

12.3.5 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. 12.6 for both software packages.

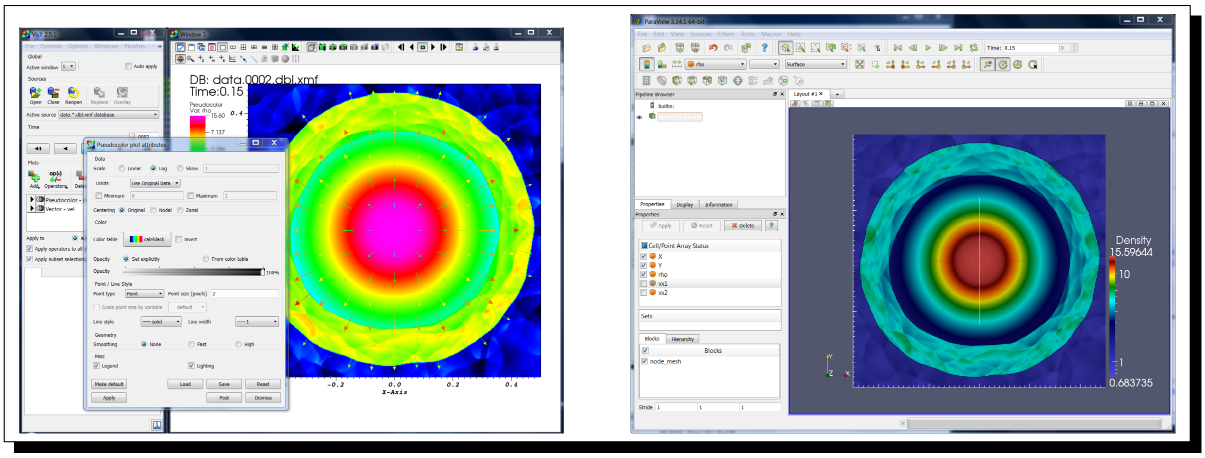


Figure 12.6: An example of visualization of an .xmf (.h5) data file using *VisIt* (left) or *ParaView* (right).

Visualization of HDF5 files. Both *VisIt* 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 the Pixie reader crashes (e.g. using *ParaView* 3.14 - 3.98) or incorrectly reads the .h5 files (e.g. using *VisIt* versions after 2.6), while all versions of both *VisIt* and *Paraview* properly open the .xmf files. All the variables are read as scalar quantities.

Visualization of VTK files. **PLUTO** 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.

13. Adaptive Mesh Refinement (AMR)

PLUTO provides adaptive mesh refinement (AMR) functionality in 1, 2 and 3 dimesions through the Chombo library[1]. Chombo provides a distributed infrastructure for parallel calculations over block-structured, adaptively refined grids. **PLUTO**-Chombo is compatible with any of the available physics modules (i.e. *HD*, *MHD*, *RHD*, *RMHD*) and grid refinement is supported in all coordinate systems. Moreover, grid zones are no longer constrained to be equilateral but sides can have different lengths. Magnetic fields are evolved using cell-centered schemes i.e., either Powell's *EIGHT_WAVES* or *DIV_CLEANING*. Constrained transport is not yet available. I/O is provided by the Hierarchical Data Format (HDF5) library[2], 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 [MZT$^+$12].
For compatibility reasons, not all the algorithms available with the static grid version of **PLUTO** have been extended to the AMR version. The AMR implementation of **PLUTO** is not compatible, at present, with:

- constrained-transport MHD;
- finite difference schemes;
- the ShearingBox module (§10.1)
- 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. For instance, the main function main.c has been replaced by amrPluto.cpp.

13.1 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 Chombo library[1] (version 3.2 is recommended);
- the HDF5 library[2] (version $<$ 1.8.14 is recommended);
- the chombo-3.2-patch.tar.gz 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.

[1]https://commons.lbl.gov/display/chombo/
[2]http://www.hdfgroup.org/HDF5/

Note: The chombo-3.2-patch.tar.gz patch archive provided with **PLUTO** 4.2 stopped being compatible with Chombo 3.2 starting from revision 23085 of the library, corresponding to patch 5 of the 3.2 release. A comprehensive list of the patches and revisions of Chombo 3.2 can be found at https://anag-repo.lbl.gov/chombo-3.2/patches.html, where it is possible to find the instructions to download older revisions. The patch archive distributed with **PLUTO** 4.3 should be now compatible with all the revisions of Chombo 3.2 and it has been tested using the currently distributed patch 5 up to revision 23352. To use **PLUTO** 4.3 with AMR we therefore advise you to download the latest Chombo 3.2 revision. If in the future you will find any problems compiling or running the Chombo 3.2 library after having applied the **PLUTO** 4.3 patch, we advise you to revert to an older and tested revision of Chombo 3.2, for example:

```
> svn --username username co -r23352 https://anag-repo.lbl.gov/svn/Chombo/release/3.2 Chombo-3.2
```

13.1.1 Installing HDF5

The HDF5 library can be downloaded from http://www.hdfgroup.org/HDF5/ and it can be used for the static grid version (§12.1.2) while it is mandatory for the AMR version and it must be installed before compiling Chombo.

Note: Both **PLUTO** (static) and **PLUTO**-Chombo (AMR) have been succesffuly tested for serial and parallel computation using with HDF5 v. $<$ 1.8.14 while parallel I/O problems were found on Ubuntu systems using newer versions.

Different builds are necessary for serial or parallel execution and, 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/lib/HDF5-serial
> make
> make check    # optional
> make install
```

This will install the libraries under `/usr/local/lib/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: The I/O of Chombo 3.2 has been updated to use the HDF5 1.8 API. Howerer, if HDF5 1.6.x is installed on your system, the support for the 1.6 API is still provided by adding the `-DH5_USE_16_API` flag to the `HDFINCFLAGS` variable inside your Make.defs.local, see §13.1.2. Nevertheless, it is not guaranteed that the HDF5 1.6 API will be supported in future Chombo releases.

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/lib/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/lib/HDF5-serial/lib` in the example above). Please make sure to add, for example,

```
> setenv LD_LIBRARY_PATH /usr/local/lib/HDF5-serial/lib:$LD_LIBRARY_PATH
```

to your .tcshrc if you're using the tcsh shell or

```
> export LD_LIBRARY_PATH="/usr/local/lib/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.

13.1.2 Installing and Configuring Chombo

Chombo 3.2 can be downloaded by direct access to the SVN server repository after free registration. The Chombo source code distribution should be unpacked under PLUTO/Lib/ and some of the library source files must be replaced using the chombo-3.2-patch.tar.gz patch-archive provided with the **PLUTO** distribution. A typical session is

```
> # get the 3.2 release of Chombo
> svn --username username co https://anag-repo.lbl.gov/svn/Chombo/release/3.2 Chombo-3.2
> tar xzvf chombo-3.2-patch.tar.gz -C Chombo-3.2/  # 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, we suggest to compile all possible configurations (that is 1, 2 and 3D serial or 1, 2 and 3D parallel). Libraries are automatically named by Chombo after the chosen configuration.

The default configuration can be set by editing manually Chombo-3.2/lib/mk/Make.defs.local where, depending on your local system and configuration, you need to set make variables. To this end:

```
> cd Chombo-3.2/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            = g++
FC             = gfortran
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-3.2/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.

13.2 Configuring and running 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 have been built under PLUTO/Lib/Chombo-3.2):

```
~/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
```

(do not use spaces in `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 Chapter 3. 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 but definitions.h and pluto.ini may contain some AMR-specific directives.

13.2.1 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.n.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.

Pre-configured examples can be found in the Test_Problems/ folder, e.g.,

- Configuration #04 of Test_Problems/HD/Mach_Reflection;
- Configuration #04 of Test_Problems/HD/Stellar_Wind;
- Configuration #03 of Test_Problems/RHD/Blast;
- Configuration #08 of Test_Problems/MHD/Rayleigh_Taylor;
- Configuration #07,#08,#11 of Test_Problems/MHD/Rotor;
- Configuration #08 of Test_Problems/MHD/Shock_Cloud.
- Configuration #03 of Test_Problems/RMHD/KH.
- Configuration #01, #02 of Test_Problems/RMHD/Shock_Cloud.

13.3 Controlling Refinement

Refinement is controlled by a series of runtime parameters defined in the *[Chombo Refinement]* block (§4.2) of your pluto.ini. Zones are tagged for refinement whenever a prescribed function $\chi(\boldsymbol{U})$ of the conserved variables and of its derivatives exceeds the threshold value χ_r 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 $\chi(\boldsymbol{U})$.

The default criterion is based on the second derivative error norm [Loh87] and it is implemented in the function **`computeRefGradient()`** in the source file Src/Chombo/TagCells.cpp. The test function adopted for this purpose is

$$\chi(\boldsymbol{U}) = \sqrt{\frac{\sum_d |\Delta_{d,+\frac{1}{2}}U - \Delta_{d,-\frac{1}{2}}U|^2}{\sum_d \left(|\Delta_{d,+\frac{1}{2}}U| + |\Delta_{d,-\frac{1}{2}}U| + \epsilon U_{d,\mathrm{ref}}\right)^2}} \tag{13.1}$$

where $U \in \boldsymbol{U}$ is a conserved variable, $\Delta_{d,\pm\frac{1}{2}}U$ are the undivided forward and backward differences in the direction d, e.g., $\Delta_{x,\pm\frac{1}{2}}U = \pm(U_{i\pm 1} - U_i)$ (see also section 4.1 in [MZT+12]). The last term appearing in the denominator, $U_{d,\mathrm{ref}}$, prevents regions of small ripples from being refined and it is defined by

$$U_{x,\mathrm{ref}} = |U_{i+1}| + 2|U_i| + |U_{i-1}| \tag{13.2}$$

with $\epsilon = 0.01$ (similar expressions hold when $d = x_2$ or $d = x_3$).

By default U is the total energy density if the EOS is not isothermal, or the mass density otherwise, see `CHOMBO_REF_VAR` in Appendix B.3. A different variable $q = q(U)$ (e.g. $q = m_x^2/2\rho$) can be used to replace U in Eq. (13.1) by copying the source file Src/Chombo/TagCells.cpp in your local working area, setting `CHOMBO_REF_VAR` to -1 and defining the appropriate expression through the function **`computeRefVar()`**. Beware, however, that $\chi(U)$ may become ill-defined if $U_{x,\mathrm{ref}}$ changes sign. This occurs, for example, when U is a vector component (e.g. momentum or magnetic field) and a better solution would be to replace $U_{d,\mathrm{ref}}$ with a constant reference value.

13.4 Output and Visualization

PLUTO-Chombo output employs the HDF5 format and the frequency of output is controlled at runtime by specifying the relevant parameters described in §4.7. 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 and, at present, Chombo data files have been successfully opened and visualized with IDL[3], VisIt[4], ParaView[5] and pyPLUTO. Examples are provided in the following. A comprehensive list of application software using HDF5 may be found 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 in the IDL programming language.

13.4.1 Visualization with IDL

PLUTO-Chombo comes with a set of visualization routines for the IDL programming language. For more information consult idl_tools.html.

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

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 (§12.3.2) 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
```

Note that IDL re-interpolates the required dataset to a uniform mesh with resolution given by the specified refinement level.

As an example, we show how to visualize the density map for the relativistic Kelvin-Helmholtz test problem as in Fig. 13.1 corresponding to the output of configuration # 03 of Test_Problems/RMHD/KH.

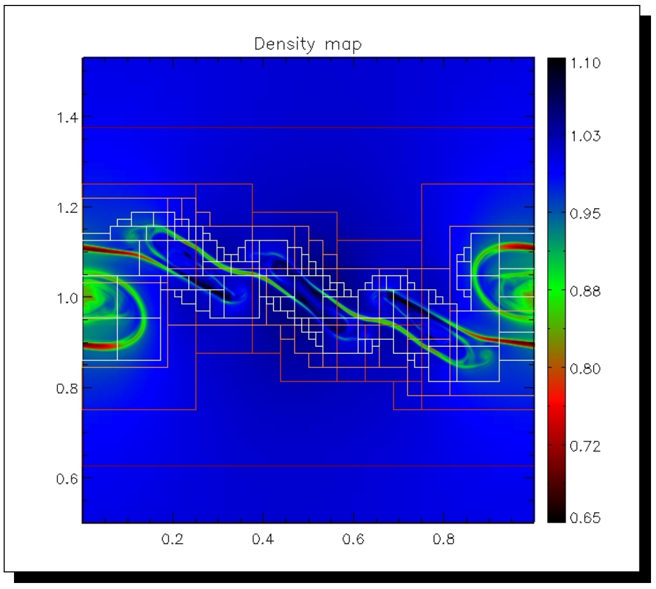


Figure 13.1: Density maps of the relativistic Kelvin-Helmholtz test problem, at $t = 5$. Refinement levels are displayed, using the oplotbox routine.

The figure has been produced with the following IDL commands:

```
IDL> PLOAD, /HDF5, dir="DATA_03",5,lev=4, x2range=[0.5,1.5]
IDL> LOADCT,6
IDL> DISPLAY,x1=x1,x2=x2,/vbar,rho,imax=1.1,imin=0.65,title="Density map"
IDL> OPLOTBOX,ctab=3
```

The last command (OPLOTBOX) overplots the levels of refinement, utilizing the color table 3. If you are plotting a 2D map in curvilinear coordinates (polar or spherical) using the DISPLAY procedure setting the `/POLAR` keyword, you can use the same `/POLAR` keyword for the OPLOTBOX procedure to correctly overplot the levels of refinement.

Reading Large Datasets. 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 subdomain through the keywords `x1range`, `x2range` and `x3range`. For instance:

```
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
```

13.4.2 Visualization with VisIt

VisIt can read Chombo HDF5 datafiles. Individual .hdf5 files or databases can be opened and visualized from the GUI in exactly the same way as .vtk of .h5 files and level plots can be over-imposed from `Add` → `Subset` → `levels`.

If you are using curvilinear coordinates or cartesian coordinates with an origin offset (i.e. the domain's lower corner $\neq [0,0,0]$) and/or different grid spacings along different directions, the correct coordinate transformation can be done by applying the `Displacement` operator. Example:

- Select a valid data.*.hdf5 database by clicking on `Open`
- `Add` → `Pseudocolor` → `rho`
- `Operators` → `Transform` → `Displace`
- Click on the Displace operator to set the attributes: `Displacement variable` → `Default` → `Vectors` → `Displacement`.

As an example we show, in Fig. 13.2, a close-up of the final solution obtained with configuration #08 of the Test_Problems/HD/Disk_Planet/ problem.

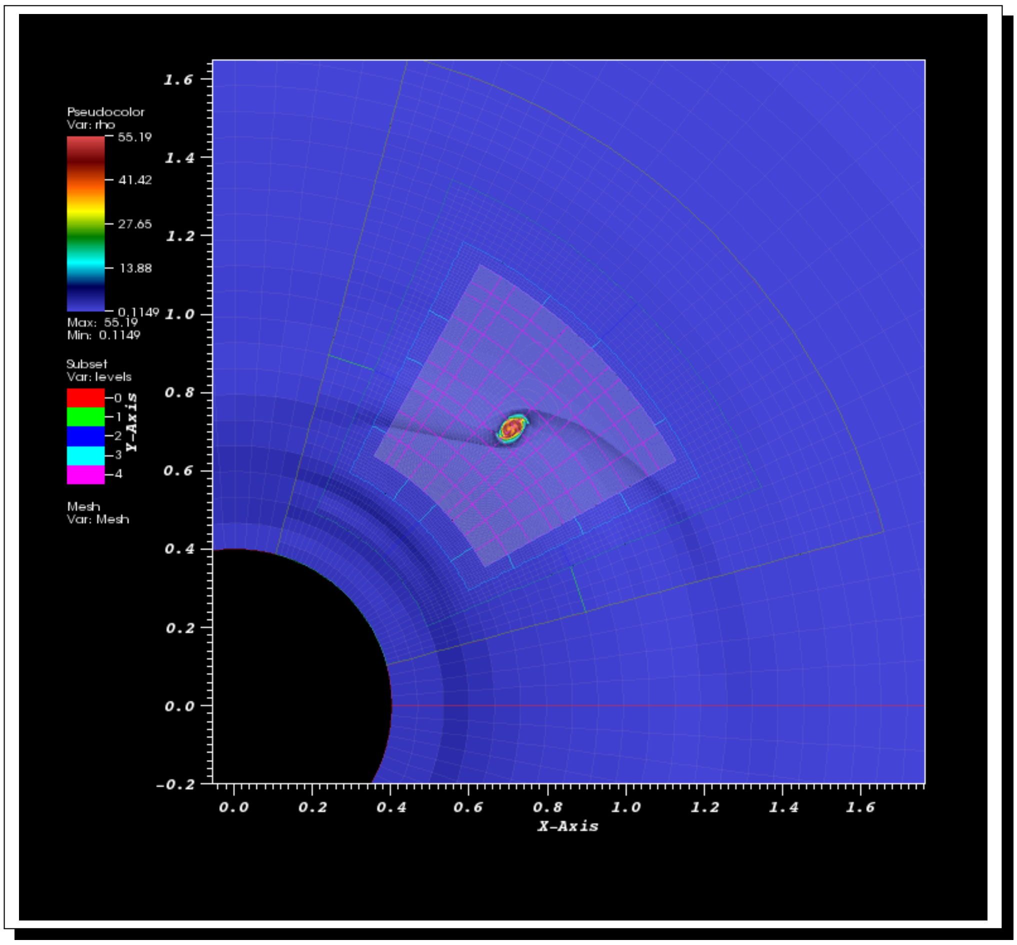


Figure 13.2: Density distribution with overplotted AMR levels for the disk-planet interaction.

13.4.3 Visualization with pyPLUTO

The simulation data obtained from **PLUTO**-Chombo is written as a HDF5 file, which can now be visualised and analysed using pyPLUTO (§12.3.3). The reader for HDF5 files with AMR data in pyPLUTO has been developed by Dr. Antoine Strugarek (Departement of Physics, University of Montreal) and has same capablities as that of IDL's HDF5LOAD. In order to use this reader it is required to install, *h5py* package, the Pythonic interface to HDF5 data.

The syntax you need is similar to that used for static grids. For example, in order to read data.0001.hdf5 at the equivalent resolution provided by the 4th refinement level,

```
> ipython --pylab
In [1]: import pyPLUTO as pp
In [2]: D = pp.pload(1,datatype='hdf5',level=4)
```

Now, the *pyPLUTO pload object*, D has all information regarding the data. To visualize (say) the density ρ, one can use the *pyPLUTO.Image* class as follows.

```
In [3]: I = pp.Image()
In [4]: I.pldisplay(D, D.rho, x1 = D.x1, x2 = D.x2, cbar=(True,
        'vertical'))
# To plot 2D R-Phi data obtained from a POLAR AMR Grid.
In [5]: I.pldisplay(D, D.rho, x1 = D.x1, x2 = D.x2, cbar=(True,
        'vertical'), polar=[True, True])
# To plot 2D R-Theta Slice from 3D POLAR AMR Data.
In [6]: I.pldisplay(D, D.rho[:,:,D.n3/2], x1 = D.x1, x2 = D.x2, cbar=(True,
        'vertical'), polar=[True, False])
```

Further, AMR levels in form of boxes can be overplotted using the *oplotbox* routine. Here, we plot the boxes for all 4 refine levels in addition to the base coarse grid.

```
In [7]: I.oplotbox(D.AMRLevel, lrange=[0,4],cval=['r','g','k','c','m'],geom=D.geometry)
```

The figure 13.3 shows the total magnetic pressure obtained for the MHD Rotor problem in 2D at the equivalent resolution provided by the 4th refinement level, also, overplotted are the AMR levels in different colored lines for all of these 4 levels.

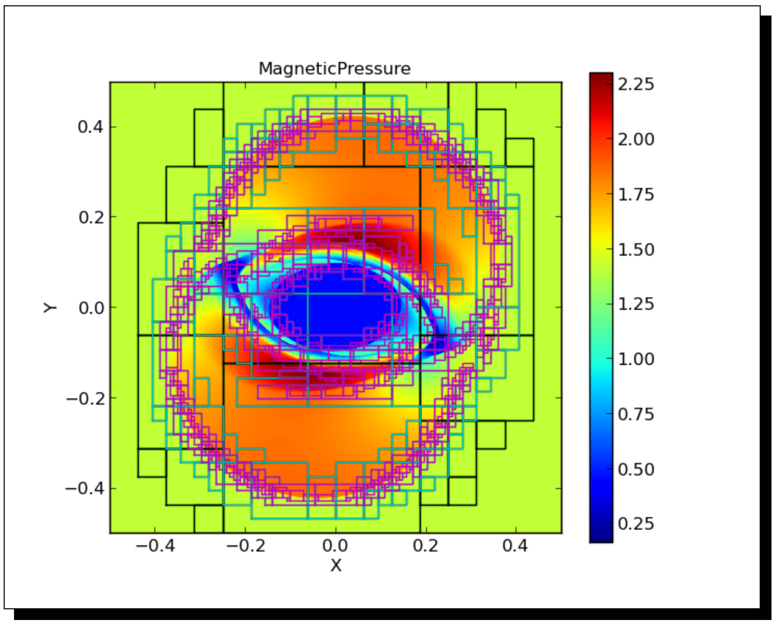


Figure 13.3: AMR Data visualisation using pyPLUTO.

<u>**Note**</u>: : The HDF5 reader is not yet integrated into the pyPLUTO's graphical user interface

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 (6.4) are discretized using the following divergence form

$$\begin{array}{lcl}
\dfrac{\partial\rho}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}) & = & 0 \\
\dfrac{\partial m_x}{\partial t} + \nabla\cdot(m_x\boldsymbol{v} - B_x\boldsymbol{B}) + \dfrac{\partial p_t}{\partial x} & = & \rho\left(g_x - \dfrac{\partial\Phi}{\partial x}\right) \\
\dfrac{\partial m_y}{\partial t} + \nabla\cdot(m_y\boldsymbol{v} - B_y\boldsymbol{B}) + \dfrac{\partial p_t}{\partial y} & = & \rho\left(g_y - \dfrac{\partial\Phi}{\partial y}\right) \\
\dfrac{\partial m_z}{\partial t} + \nabla\cdot(m_z\boldsymbol{v} - B_z\boldsymbol{B}) + \dfrac{\partial p_t}{\partial z} & = & \rho\left(g_z - \dfrac{\partial\Phi}{\partial z}\right) \\
\dfrac{\partial}{\partial t}(E_t + \rho\Phi) + \nabla\cdot\Big[(E_t + p_t + \rho\Phi)\boldsymbol{v} - \boldsymbol{B}\,(\boldsymbol{v}\cdot\boldsymbol{B})\Big] & = & \rho\boldsymbol{v}\cdot\boldsymbol{g} \\
\dfrac{\partial B_x}{\partial t} + \dfrac{\partial E_z}{\partial y} - \dfrac{\partial E_y}{\partial z} & = & 0 \\
\dfrac{\partial B_y}{\partial t} + \dfrac{\partial E_x}{\partial z} - \dfrac{\partial E_z}{\partial x} & = & 0 \\
\dfrac{\partial B_z}{\partial t} + \dfrac{\partial E_y}{\partial x} - \dfrac{\partial E_x}{\partial y} & = & 0
\end{array} \tag{A.1}$$

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, (E_x, E_y, E_z) are the components of the electromotive force $\boldsymbol{E} = -\boldsymbol{v}\times\boldsymbol{B}$, $\boldsymbol{g}$ 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 (6.4) are discretized using the following divergence form

$$
\begin{array}{lcl}
\displaystyle \frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}) & = & 0 \\[2ex]
\displaystyle \frac{\partial m_R}{\partial t} + \nabla\cdot(m_R\boldsymbol{v} - B_R\boldsymbol{B}) + \frac{\partial p_t}{\partial R} & = & \displaystyle \rho\left(g_R - \frac{\partial\Phi}{\partial R}\right) + \frac{\rho v_\phi^2 - B_\phi^2}{R} \\[2ex]
\displaystyle \frac{\partial m_\phi}{\partial t} + \nabla^R\cdot(m_\phi\boldsymbol{v} - B_\phi\boldsymbol{B}) + \frac{1}{R}\frac{\partial p_t}{\partial\phi} & = & \displaystyle \rho\left(g_\phi - \frac{1}{R}\frac{\partial\Phi}{\partial\phi}\right) \\[2ex]
\displaystyle \frac{\partial m_z}{\partial t} + \nabla\cdot(m_z\boldsymbol{v} - B_z\boldsymbol{B}) + \frac{\partial p_t}{\partial z} & = & \displaystyle \rho\left(g_z - \frac{\partial\Phi}{\partial z}\right) \\[2ex]
\displaystyle \frac{\partial}{\partial t}(E_t + \rho\Phi) + \nabla\cdot\Big[(E_t + p_t + \rho\Phi)\boldsymbol{v} - \boldsymbol{B}\,(\boldsymbol{v}\cdot\boldsymbol{B})\Big] & = & \rho\boldsymbol{v}\cdot\boldsymbol{g} \\[2ex]
\displaystyle \frac{\partial B_R}{\partial t} + \frac{1}{R}\frac{\partial E_z}{\partial\phi} - \frac{\partial E_\phi}{\partial z} & = & 0 \\[2ex]
\displaystyle \frac{\partial B_\phi}{\partial t} + \frac{\partial E_R}{\partial z} - \frac{\partial E_z}{\partial R} & = & 0 \\[2ex]
\displaystyle \frac{\partial B_z}{\partial t} + \frac{1}{R}\frac{\partial(RE_\phi)}{\partial R} - \frac{1}{R}\frac{\partial E_R}{\partial\phi} & = & 0\,,
\end{array}
\tag{A.2}
$$

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

$$
\begin{array}{lcl}
\nabla\cdot\boldsymbol{F} & = & \displaystyle \frac{1}{R}\frac{\partial(RF_R)}{\partial R} + \frac{1}{R}\frac{\partial F_\phi}{\partial\phi} + \frac{\partial F_z}{\partial z}\,, \\[2ex]
\nabla^R\cdot\boldsymbol{F} & = & \displaystyle \frac{1}{R^2}\frac{\partial(R^2F_R)}{\partial R} + \frac{1}{R}\frac{\partial F_\phi}{\partial\phi} + \frac{\partial F_z}{\partial z}
\end{array}
\tag{A.3}
$$

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, (E_R, E_ϕ, E_z) are the components of the electromotive force $\boldsymbol{E} = -\boldsymbol{v}\times\boldsymbol{B}$, $\boldsymbol{g}$ is the body force vector and Φ is the gravitational potential.

The operator $\nabla^R\cdot()$ is discretized in a conservative way using the following representation:

$$
\nabla^R\cdot\boldsymbol{F} = \left(\frac{A^R_{i+\frac{1}{2}}R_{i+\frac{1}{2}}F^R_{i+\frac{1}{2}} - A^R_{i-\frac{1}{2}}R_{i-\frac{1}{2}}F^R_{i-\frac{1}{2}}}{R_i\Delta\mathcal{V}}\right) + \left(\frac{A^\phi_{j+\frac{1}{2}}F^\phi_{j+\frac{1}{2}} - A^\phi_{j-\frac{1}{2}}F^\phi_{j-\frac{1}{2}}}{\Delta\mathcal{V}}\right) + \left(\frac{A^z_{k+\frac{1}{2}}F^z_{k+\frac{1}{2}} - A^z_{k-\frac{1}{2}}F^z_{k-\frac{1}{2}}}{\Delta\mathcal{V}}\right) \tag{A.4}
$$

where the half-integer notation have been kept for the sake of exposition (i.e. $\Delta\mathcal{V} \equiv \Delta\mathcal{V}_{ijk} = R_i\Delta R_i\Delta\phi_j\Delta z_k$). Note that, starting with **PLUTO** 4.3, area and volume do not depend on one coordinate at a time but are fully 3D arrays (example: $A^R_{i+\frac{1}{2}} = R_{i+\frac{1}{2}}\Delta\phi_j\Delta z_k$).

A.1.3 Spherical Coordinates

In spherical coordinates (r, θ, ϕ) the ideal MHD equations (6.4) are discretized using the following divergence form

$$
\begin{array}{lcl}
\dfrac{\partial \rho}{\partial t} + \nabla\cdot(\rho\boldsymbol{v}) & = & 0 \\[2ex]
\dfrac{\partial m_r}{\partial t} + \nabla\cdot(m_r\boldsymbol{v} - B_r\boldsymbol{B}) + \dfrac{\partial p_t}{\partial r} & = & \rho\left(g_r - \dfrac{\partial\Phi}{\partial r}\right) + \dfrac{\rho v_\theta^2 - B_\theta^2}{r} + \dfrac{\rho v_\phi^2 - B_\phi^2}{r} \\[2ex]
\dfrac{\partial m_\theta}{\partial t} + \nabla\cdot(m_\theta\boldsymbol{v} - B_\theta\boldsymbol{B}) + \dfrac{1}{r}\dfrac{\partial p_t}{\partial\theta} & = & \rho\left(g_\theta - \dfrac{1}{r}\dfrac{\partial\Phi}{\partial\theta}\right) - \dfrac{\rho v_\theta v_r - B_\theta B_r}{r} + \cot\theta\dfrac{\rho v_\phi^2 - B_\phi^2}{r} \\[2ex]
\dfrac{\partial m_\phi}{\partial t} + \nabla^r\cdot(m_\phi\boldsymbol{v} - B_\phi\boldsymbol{B}) + \dfrac{1}{r\sin\theta}\dfrac{\partial p_t}{\partial\phi} & = & \rho\left(g_\phi - \dfrac{1}{r\sin\theta}\dfrac{\partial\Phi}{\partial\phi}\right) \\[2ex]
\dfrac{\partial}{\partial t}(E_t + \rho\Phi) + \nabla\cdot\Big[(E_t + p_t + \rho\Phi)\boldsymbol{v} - \boldsymbol{B}\,(\boldsymbol{v}\cdot\boldsymbol{B})\Big] & = & \rho\boldsymbol{v}\cdot\boldsymbol{g} \\[2ex]
\dfrac{\partial B_r}{\partial t} + \dfrac{1}{r\sin\theta}\dfrac{\partial(\sin\theta E_\phi)}{\partial\theta} - \dfrac{1}{r\sin\theta}\dfrac{\partial E_\theta}{\partial\phi} & = & 0 \\[2ex]
\dfrac{\partial B_\theta}{\partial t} + \dfrac{1}{r\sin\theta}\dfrac{\partial E_r}{\partial\phi} - \dfrac{1}{r}\dfrac{\partial(rE_\phi)}{\partial r} & = & 0 \\[2ex]
\dfrac{\partial B_\phi}{\partial t} + \dfrac{1}{r}\dfrac{\partial(rE_\theta)}{\partial r} - \dfrac{1}{r}\dfrac{\partial E_r}{\partial\theta} & = & 0
\end{array}
\tag{A.5}
$$

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

$$
\begin{array}{lcl}
\nabla\cdot\boldsymbol{F} & = & \dfrac{1}{r^2}\dfrac{\partial(r^2F_r)}{\partial r} + \dfrac{1}{r\sin\theta}\dfrac{\partial(\sin\theta F_\theta)}{\partial\theta} + \dfrac{1}{r\sin\theta}\dfrac{\partial F_\phi}{\partial\phi} \\[2ex]
\nabla^r\cdot\boldsymbol{F} & = & \dfrac{1}{r^3}\dfrac{\partial(r^3F_r)}{\partial r} + \dfrac{1}{r\sin^2\theta}\dfrac{\partial(\sin^2\theta F_\theta)}{\partial\theta} + \dfrac{1}{r\sin\theta}\dfrac{\partial F_\phi}{\partial\phi}
\end{array}
\tag{A.6}
$$

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, (E_r, E_θ, E_ϕ) are the components of the electromotive force $\boldsymbol{E} = -\boldsymbol{v}\times\boldsymbol{B}$, $\boldsymbol{g}$ is the body force vector and Φ is the gravitational potential.

At the discrete level, the operator $\nabla^r\cdot()$ keeps its conservative nature by using the following representation:

$$
\nabla^r\cdot\boldsymbol{F} = \left(\frac{A^r_{i+\frac{1}{2}}r_{i+\frac{1}{2}}F^r_{i+\frac{1}{2}} - A^r_{i-\frac{1}{2}}r_{i+\frac{1}{2}}F^r_{i-\frac{1}{2}}}{\Delta\mathcal{V}_i}\right) + \left(\frac{A^\theta_{j+\frac{1}{2}}F^\theta_{j+\frac{1}{2}} - A^\theta_{j-\frac{1}{2}}F^\theta_{j-\frac{1}{2}}}{\Delta\mathcal{V}_i}\right) + S^r + S^\theta \tag{A.7}
$$

where $\Delta\mathcal{V} = (r^3_{i+\frac{1}{2}} - r^3_{i-\frac{1}{2}})\Delta(1 - \cos\theta_j)\Delta\phi_k$. Written in this form, the inclusion of the source term does not violate conservation and it is carried out in the file rhs_source.c.

A.2 (Special) Relativistic MHD Equations

A.2.1 Cartesian Coordinates

In Cartesian coordinates (x, y, z), the relativistic MHD equations (6.12) take the form

$$
\begin{array}{lcl}
\dfrac{\partial D}{\partial t} + \nabla\cdot(D\boldsymbol{v}) & = & 0 \\
\dfrac{\partial m_x}{\partial t} + \nabla\cdot\left[(w+b^2)v_x\boldsymbol{v} - b_x\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial x} & = & \rho g_x \\
\dfrac{\partial m_y}{\partial t} + \nabla\cdot\left[(w+b^2)v_y\boldsymbol{v} - b_y\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial y} & = & \rho g_y \\
\dfrac{\partial m_z}{\partial t} + \nabla\cdot\left[(w+b^2)v_z\boldsymbol{v} - b_z\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial z} & = & \rho g_z \\
\dfrac{\partial E_t}{\partial t} + \nabla\cdot(\boldsymbol{m} - D\boldsymbol{v}) & = & D\boldsymbol{v}\cdot\boldsymbol{g} \\
\dfrac{\partial B_x}{\partial t} + \dfrac{\partial E_z}{\partial y} - \dfrac{\partial E_y}{\partial z} & = & 0 \\
\dfrac{\partial B_y}{\partial t} + \dfrac{\partial E_x}{\partial z} - \dfrac{\partial E_z}{\partial x} & = & 0 \\
\dfrac{\partial B_z}{\partial t} + \dfrac{\partial E_y}{\partial x} - \dfrac{\partial E_x}{\partial y} & = & 0
\end{array}
\tag{A.8}
$$

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, (E_x, E_y, E_z) are the components of the electromotive force $\boldsymbol{E} = -\boldsymbol{v}\times\boldsymbol{B}$ and $\boldsymbol{g}$ is the body force vector.

A.2.2 Polar Coordinates

In polar cylindrical coordinates (R, ϕ, z), the RMHD Equations (6.12) are discretized using the following form

$$
\begin{array}{lcl}
\dfrac{\partial D}{\partial t} + \nabla\cdot(D\boldsymbol{v}) & = & 0 \\
\dfrac{\partial m_R}{\partial t} + \nabla\cdot\left[(w+b^2)v_R\boldsymbol{v} - b_R\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial R} & = & \rho g_R + \dfrac{m_\phi v_\phi}{R} - \left(\dfrac{B_\phi}{\gamma^2} + (\boldsymbol{v}\cdot\boldsymbol{B})v_\phi\right)\dfrac{B_\phi}{R} \\
\dfrac{\partial m_\phi}{\partial t} + \nabla^R\cdot\left[(w+b^2)v_\phi\boldsymbol{v} - b_\phi\boldsymbol{b}\right] + \dfrac{1}{R}\dfrac{\partial p_t}{\partial \phi} & = & \rho g_\phi \\
\dfrac{\partial m_z}{\partial t} + \nabla\cdot\left[(w+b^2)v_z\boldsymbol{v} - b_z\boldsymbol{b}\right] + \dfrac{\partial p_t}{\partial z} & = & \rho g_z \\
\dfrac{\partial E_t}{\partial t} + \nabla\cdot(\boldsymbol{m} - D\boldsymbol{v}) & = & D\boldsymbol{v}\cdot\boldsymbol{g} \\
\dfrac{\partial B_R}{\partial t} + \dfrac{1}{R}\dfrac{\partial E_z}{\partial \phi} - \dfrac{\partial E_\phi}{\partial z} & = & 0 \\
\dfrac{\partial B_\phi}{\partial t} + \dfrac{\partial E_R}{\partial z} - \dfrac{\partial E_z}{\partial R} & = & 0 \\
\dfrac{\partial B_z}{\partial t} + \dfrac{1}{R}\dfrac{\partial (RE_\phi)}{\partial R} - \dfrac{1}{R}\dfrac{\partial E_R}{\partial \phi} & = & 0\,,
\end{array}
\tag{A.9}
$$

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

$$\begin{aligned}
\nabla\cdot\boldsymbol{F} &= \frac{1}{R}\frac{\partial(RF_R)}{\partial R}+\frac{1}{R}\frac{\partial F_\phi}{\partial\phi}+\frac{\partial F_z}{\partial z}\,, \\
\nabla^R\cdot\boldsymbol{F} &= \frac{1}{R^2}\frac{\partial(R^2F_R)}{\partial R}+\frac{1}{R}\frac{\partial F_\phi}{\partial\phi}+\frac{\partial F_z}{\partial z}
\end{aligned}\tag{A.10}$$

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, (E_R,E_ϕ,E_z) are the components of the electromotive force $\boldsymbol{E}=-\boldsymbol{v}\times\boldsymbol{B}$, $\boldsymbol{g}$ 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 (6.12) are discretized using the following divergence form

$$\begin{aligned}
\frac{\partial D}{\partial t}+\nabla\cdot(D\boldsymbol{v}) &= 0 \\
\frac{\partial m_r}{\partial t}+\nabla\cdot\left[(w+b^2)v_r\boldsymbol{v}-b_r\boldsymbol{b}\right]+\frac{\partial p_t}{\partial r} &= \rho g_r+\frac{m_\theta v_\theta+m_\phi v_\phi}{r}+ \\
&\quad -\left(\frac{B_\theta}{\gamma^2}+(\boldsymbol{v}\cdot\boldsymbol{B})v_\theta\right)\frac{B_\theta}{r}-\left(\frac{B_\phi}{\gamma^2}+(\boldsymbol{v}\cdot\boldsymbol{B})v_\phi\right)\frac{B_\phi}{r} \\
\frac{\partial m_\theta}{\partial t}+\nabla\cdot\left[(w+b^2)v_\theta\boldsymbol{v}-b_\theta\boldsymbol{b}\right]+\frac{1}{r}\frac{\partial p_t}{\partial\theta} &= \rho g_\theta-\frac{m_\theta v_r-\cot\theta m_\phi v_\phi}{r} \\
&\quad +\left(\frac{B_\theta}{\gamma^2}+(\boldsymbol{v}\cdot\boldsymbol{B})v_\theta\right)\frac{B_r}{r}-\cot\theta\left(\frac{B_\phi}{\gamma^2}+(\boldsymbol{v}\cdot\boldsymbol{B})v_\phi\right)\frac{B_\phi}{r} \\
\frac{\partial m_\phi}{\partial t}+\nabla^r\cdot\left[(w+b^2)v_\phi\boldsymbol{v}-b_\phi\boldsymbol{b}\right]+\frac{1}{r\sin\theta}\frac{\partial p_t}{\partial\phi} &= \rho g_\phi \\
\frac{\partial E_t}{\partial t}+\nabla\cdot(\boldsymbol{m}-D\boldsymbol{v}) &= D\boldsymbol{v}\cdot\boldsymbol{g} \\
\frac{\partial B_r}{\partial t}+\frac{1}{r\sin\theta}\frac{\partial(\sin\theta E_\phi)}{\partial\theta}-\frac{1}{r\sin\theta}\frac{\partial E_\theta}{\partial\phi} &= 0 \\
\frac{\partial B_\theta}{\partial t}+\frac{1}{r\sin\theta}\frac{\partial E_r}{\partial\phi}-\frac{1}{r}\frac{\partial(rE_\phi)}{\partial r} &= 0 \\
\frac{\partial B_\phi}{\partial t}+\frac{1}{r}\frac{\partial(rE_\theta)}{\partial r}-\frac{1}{r}\frac{\partial E_r}{\partial\theta} &= 0
\end{aligned}\tag{A.11}$$

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

$$\begin{aligned}
\nabla\cdot\boldsymbol{F} &= \frac{1}{r^2}\frac{\partial(r^2F_r)}{\partial r}+\frac{1}{r\sin\theta}\frac{\partial(\sin\theta F_\theta)}{\partial\theta}+\frac{1}{r\sin\theta}\frac{\partial F_\phi}{\partial\phi} \\
\nabla^r\cdot\boldsymbol{F} &= \frac{1}{r^3}\frac{\partial(r^3F_r)}{\partial r}+\frac{1}{r\sin^2\theta}\frac{\partial(\sin^2\theta F_\theta)}{\partial\theta}+\frac{1}{r\sin\theta}\frac{\partial F_\phi}{\partial\phi}
\end{aligned}\tag{A.12}$$

B. Predefined Constants and Macros

B.1 Predefined Physical Constants

PLUTO has several predefined physical and astronomical constants in c.g.s. units which may be used anywhere in the code (see pluto.h):

```
#define CONST_AH      1.008              /**< Atomic weight of Hydrogen  */
#define CONST_AHe     4.004              /**< Atomic weight of Helium  */
#define CONST_AZ      30.0               /**< Mean atomic weight of heavy elements */
#define CONST_amu     1.66053886e-24     /**<  Atomic mass unit.          */
#define CONST_au      1.49597892e13      /**<  Astronomical unit.         */
#define CONST_c       2.99792458e10      /**<  Speed of Light.            */
#define CONST_eV      1.602176463158e-12 /**<  Electron Volt in erg.      */
#define CONST_G       6.6726e-8          /**<  Gravitational Constant.    */
#define CONST_h       6.62606876e-27     /**<  Planck Constant.           */
#define CONST_kB      1.3806505e-16      /**<  Boltzmann constant.        */
#define CONST_ly      0.9461e18          /**<  Light year.                */
#define CONST_mp      1.67262171e-24     /**<  Proton mass.               */
#define CONST_mn      1.67492728e-24     /**<  Neutron mass.              */
#define CONST_me      9.1093826e-28      /**<  Electron mass.             */
#define CONST_mH      1.6733e-24         /**<  Hydrogen atom mass.        */
#define CONST_Msun    2.e33              /**<  Solar Mass.                */
#define CONST_Mearth  5.9736e27          /**<  Earth Mass.                */
#define CONST_NA      6.0221367e23       /**<  Avogadro Contant.          */
#define CONST_pc      3.0856775807e18    /**<  Parsec.                    */
#define CONST_PI      3.14159265358979   /**<  \f$ \pi \f$.               */
#define CONST_Rearth  6.378136e8         /**<  Earth Radius.              */
#define CONST_Rsun    6.96e10            /**<  Solar Radius.              */
#define CONST_sigma   5.67051e-5         /**<  Stephan Boltmann constant. */
#define CONST_sigmaT  6.6524e-25         /**<  Thomson Cross section.    */
```

B.2 Predefined Function-Like Macros

PLUTO comes with a number of pre-defined function-like macros to implement simple arithmetic operations such as maximum (**MAX**), minimum (**MIN**), or looping over a specific portion of the computational domain (e.g. **DOM_LOOP** or **TOT_LOOP**). Please refer to the Doc/Doxygen/html/macros_8h.html page in the API reference guide (Doc/Doxygen/html/index.html).

B.3 Advanced Options

PLUTO allows a number of switches to be fine-tuned directly from your definitions.h in the user-defined constant section, see §2.3. These advanced options are described in Table B.1.

Table B.1: PLUTO advanced options.

Name	**Value**	**Description**
ARTIFICIAL_VISC	(*real*)	The amount of artificial (Lapidus-type) viscosity ν added to the two-shock Riemann solver flux (only): $$\boldsymbol{F} \quad\rightarrow\quad \boldsymbol{F} + \nu\max(v_{n,L} - v_{n,R}, 0)(\boldsymbol{U}_L - \boldsymbol{U}_R) \qquad (B.1)$$ where v_n is the velocity normal to the interface. This term introduces an extra amount of numerical dissipation [CW84] useful to reduce small-amplitude oscillations occurring when a characteristic speed associatedwith a strong shock, measured relative to the grid, vanishes. Typical values are around 0.1. By default it is not used. **Note**: this constant has no effect with other Riemann solvers.
ASSIGN_VECTOR_POTENTIAL	(*YES*/*NO*)	When set to *YES*, magnetic field components are initialized from the vector potential. In the constrained transport algorithm (CT, §6.2.3.3), this guarantees that the magnetic field has zero divergence. When set to *NO*, assignment proceeds in the usual way, see §6.2.2 for more details.
CHAR_LIMITING	(*YES*/*NO*)	Set it 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, reconstruction in characteristic ariables is known to produce better quality solutions by suppressing unwanted numerical oscillation in proximity of strong discontinuities and leading to a better entropy enforcement.
CHOMBO_CONS_AM	(*YES/NO*)	In curvilinear coordinates, set this switch to *YES* to enforce angular momentum conservation during the prolongation and restriction operation with **PLUTO**-Chombo . Default value is *YES* when CHOMBO_EN_SWITCH is set to *YES*.
CHOMBO_LOGR	(*YES*/*NO*)	Enable this switch if you want to produce an equally-spaced logarithmic grid in the radial direction in *POLAR* or *SPHERICAL* coordinates when using **PLUTO**-Chombo . A logarithmic grid has the advantage of preserving the cell aspect ratio both close to and far away from the origin. The default value is *NO*.
CHOMBO_REF_VAR	(<*vname*>)	Sets the name of the conservative variable used by Chombo when tagging zones for refinement. Allowed values are *RHO* (for density), *ENG* (for total energy), *MX1* (for normal component of momentum), etc... The default value is total energy density or density when there's no energy equation. The special value -1 can be given to supply a user-defined variable instead (e.g. pressure or kinetic energy) using the **computeRefVar()** function. See §13.3 for more detail. Notice that, since CHOMBO_REF_VAR is one of the conservative variables used to perform prolongation and restriction operations, if CHOMBO_CONS_AM is set to *YES* (see §B.3), iMPHI stands for the conserved angular momentum. Important: owing to the different type of conserved variable names, the CHOMBO_REF_VAR should never be used inside pre-processor conditional statements.

Continued on next page

Table B.1 – *Continued from previous page*

Name	Value	Description
`CHTR_REF_STATE`	(*1/2/3*)	Set the reference state used during the characteristic tracing step (see Src/States/char_tracing.c), as explained in section 3.3 of [MZT+12]. The allowed values are 1, 2 or 3: 1: use the cell-centered value: $w^{\rm ref}_{i,\pm} = w_{i,0}$. This choice is slightly more diffusive but has found to work well for flows containing strong discontinuities; 2: No reference state is introduced and the interpolated states at the base time level are used: $w^{\rm ref}_{i,\pm} = w_{i,\pm}$. This is found to be a good choice in presence of smooth flow and equlibrium configurations. 3: reference states are constructed as in the original PPM algorithm [CW84, Col90] to minimize the size of the term susceptible to characteristic limiting (see Eq. [29] and [30] of [MZT+12]). The default value is 3 except for *PARABOLIC/WENO* reconstruction in characteristic variable for which 2 is used.
`CT_EMF_AVERAGE`	(*string*)	Control how the electromotive force (EMF) is integrated from the face center to the edges. This is discussed in more detailed in §6.2.3.3.
`CT_EN_CORRECTION`	(*YES/NO*)	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 [BS99].
`EPS_PSHOCK_ENTROPY`	(*real*)	Sets the maximum shock strength above which fluid variables inside a given computational zone can be safely updated using the entropy equation (see §2.2.4 and the source file Src/flag_shock.c). It has effect only when `ENTROPY_SWITCH` has been enabled. A lower value will trigger the flattening procedure in more zones. Default is 0.05.
`EPS_PSHOCK_FLATTEN`	(*real*)	Sets the minimum shock strength above which the *MULTID* shock flattening algorithm flags a zone to be inside a shock (see Src/flag_shock.c). It has effect only when `SHOCK_FLATTENING` is set to *MULTID*. A lower value will trigger the flattening procedure in more zones. Default is 5.0.
`FARGO_AVERAGE_VELOCITY`	(*YES/NO*)	Set this to *YES* if the FARGO orbital velocity w should be computed by averaging the azimuthal velocity every fixed number of steps. When set to *NO*, w is computed from the **`FARGO_SetVelocity()`** function. Default is *NO*, or *YES* if FARGO is used with the shearing box module.
`FARGO_NSTEP_AVERAGE`	(*int*)	Sets how often the orbital velocity should be recomputed in the FARGO transport step. Default is 10.
`FARGO_ORDER`	(*int*)	Sets the spatial order of the reconstruction used during the linear transport step of the FARGO algorithm. The allowed values are 3 (third-order, PPM-like reconstruction) or 2 (second-order MUSCL-HANCOCK scheme). Default is 3.
`FARGO_OUTPUT_VTOT`	(*YES/NO*)	Enable/disable writing of the total velocity to main output data (default is *YES*).
`FORCED_TURB_ENERGY`	(*real*)	Defines the input value of stirring energy that is injected. By default its value is set to 2×10^{-3}.
`FORCED_TURB_DECAY`	(*real*)	The decay time for the turbulence is set by this constant. Its default value is set to be 0.5

Continued on next page

Table B.1 – *Continued from previous page*

Name	Value	Description
FORCED_TURB_KMIN	(*real*)	The minimum value of wave-vector at which the energy is injected (on larger scales). The default value is set to be 2π implying injection at the scales equivalent to the box size.
FORCED_TURB_KMAX	(*real*)	The maximum value of wave-vector at which the energy is injected (on smaller scales). The default value is set to be 6π implying injection at the scales equivalent to the one third of the box size.
FORCED_TURB_FREQ	(*int*)	This parameter determines the time interval in terms of number of time steps at which the phases have to be updated by the OU process. By default its value is set to 1 implying that the phases are updated at each time step.
GLM_ALPHA	(*real*)	Sets the value of the constant α used monopole damping in the GLM formalism, see §6.2.3.2. The default value is 0.1.
GLM_EXTENDED	(*YES/NO*)	Enable the (E)xtented GLM form of the MHD equations, see §6.2.3.2. Default value is *NO*.
H_MASS_FRAC	(*double*)	Hydrogen mass fraction, $X = m_u n_H A_H/\rho$. Used to compute FRAC_He and FRAC_Z in the definition of the mean molecular weight, see §5.1.2. Default value is $X = 0.7110$.
He_MASS_FRAC	(*double*)	Helium mass fraction, $Y = m_u n_{He} A_{He}/\rho$. Used to compute FRAC_He and FRAC_Z in the definition of the mean molecular weight, see §5.1.2. Note that the fraction of metals is always computed as $Z = 1 - X - Y$. The default value is $Y = 0.2741$.
ID_NZ_MAX	(*int*)	An integer number specifying the size (in the 3rd dimension) of the buffer used to read data from input data files. Default is 4.
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).
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 §5.3. 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).

Continued on next page

Table B.1 – *Continued from previous page*

Name	**Value**	**Description**
LIMITER	(*string*)	Sets the limiter(s) to be applied when RECONSTRUCTION is set to *LINEAR*. Here *string* can be one of • *FLAT_LIM*: set slope to zero (1st order reconstruction); • *MINMOD_LIM*: use the minmod limiter (most diffusive); • *VANALBADA_LIM*: use the van Albada limiter function; • *OSPRE_LIM*: use the OSPRE limiter; • *VANLEER_LIM*: use the harmonic mean limiter of van Leer; • *UMIST_LIM*: use the umist limiter; • *MC_LIM*: use the monotonized central difference limiter (least diffusive); • *DEFAULT*: keep the default setting (defined in Src/States/set_limiter.c). where *MINMOD_LIM* is the most diffusive and *MC_LIM* is the least diffusive limiter. The TVD diagram for the various limiter functions is shown in Fig B.1. All of the above limiters employ a 3-point stencil.
PARTICLES_CR_C	(*real*)	[*Particle Module* only] Specifies the speed of light $\mathbb{C}$ used in the equation of motion of CR particles. Default is 10^4.
PARTICLES_CR_E_MC	(*real*)	[*Particle Module* only] The charge to mass ratio of CR particles in dimensionless units. This constants enters in the definition of the CR force Eq (11.4) and in the particle equation of motion (11.2). When set to unity (default) is equivalent to assume the ion skin depth c/ω_{pi} as the unit length.
PARTICLES_CR_E_MC_GAS	(*real*)	[*Particle Module* only] The charge to mass ratio of the fluid, used to compute $\boldsymbol{F}_{\rm CR}$, Eq. (11.4). Default is 1.
PARTICLES_CR_FEEDBACK	(*YES/NO*)	[*Particle Module* only] Enable / disable CR particle feedback on the fluid. When enabled (default), the full MHD-PIC equations (11.3) are solved. When disabled, particles are treated as passive test-particles obeying Eq. (11.2) evolving in the fluid' electromagnetic field $\boldsymbol{B}$ and $c\boldsymbol{E} = -\boldsymbol{v} \times \boldsymbol{B}$.
PARTICLES_CR_LARMOR_EPS	(*real*)	[*Particle Module* only] A safety factor giving the fraction of Larmor time captured during a single particle integration step or sub-step. A lower value means more steps must be taken to reproduce a particle orbit. Default is 0.3.
PARTICLES_CR_NCELLS_EPS	(*real*)	[*Particle Module* only] The maximum number of zones that a particle is allowed to cross during a single integration step or sub-step. Default is 1.8.
PARTICLES_CR_NSUB	(*int*)	[*Particle Module* only] When > 0, it specifies the maximum number of sub-steps (per hydro step) used during particle subcycling. The overall time step will be adjusted so that no more than PARTICLES_CR_NSUB sub-steps should be necessary during a single sub-cycle. However, a negative number can also be provided to force the number of sub-steps to stay constant regardless of the fluid step.

Continued on next page

Table B.1 – *Continued from previous page*

Name	**Value**	**Description**
PARTICLES_CR_PREDICTOR	(*0/1/2*)	[*Particle Module* only] Enable the particle predictor/subcycling strategy. When set to 0, no predictor is done and sub-cycling is automatically engaged when PARTICLES_CR_NSUB. When set to 1, a predictor step is performed to compute the full electric field at the half time level of each sub-step. When set to 2 (default), sub-cycling can be used with an even number of sub-steps and the CR force is re-computed every other step.
PARTICLES_CR_WRITE_4VEL	(*YES/NO*)	[*Particle Module* only] Enable / disable writing of particles four-velocity (rather than 3-velocity) to VTK and binary files.
PARTICLES_LP_MICROETA	(*real*)	[*Particle Module* only] A dimensionless parameter giving the ratio between the Larmor frequency and the electron scattering frequency. This parameter governs the un-resolved turbulence and describes the degree to which the particles are magnetized. (see [TK15]).
PARTICLES_LP_ICCMBZ	(*real*)	[*Particle Module* only] Red-shift of the source used to scale the temperature of the CMB photons. This is relevant for estimating the IC emission due to interaction of leptons with CMB photons. The default value is 0.0, implying the default CMB temperature is 2.73 K.
PARTICLES_LP_SHK_THRESHOLD	(*real*)	[*Particle Module* only] A float value greater than 1.0 (by default its value is set to 3.0) that controls when the particle has to be considered inside the shock.
PARTICLES_LP_SHK_GRADP	(*real*)	[*Particle Module* only] A float value which determines the boundary between the upstream and downstream while the particle is inside the shock.
PARTICLES_LP_NONTH_FRACN	(*real*)	[*Particle Module* only] A fraction of non-thermal micro-particles that are injected in the shock as the macro-particle leaves the domain.
PARTICLES_LP_NONTH_FRACE	(*real*)	[*Particle Module* only] A fraction of fluid internal energy at shocks that goes to energize the non-thermal particles that are injected in the shock.
PARTICLES_TYPE	(*name*)	[*Particle Module* only] A macro selecting the particle type. Allowed values are *COSMIC_RAYS*, *LAGRANGIAN*.
PARTICLES_SHAPE	(*1/2/3*)	[*Particle Module* only] Sets the particles shape function when interpolating on the grid. Here 1, 2 and 3 refers to *Nearest Neighbour Point* (NGP), *Cloud-In-Cell* (CIC) and *Triangular Shape Cloud* (TGP). Default is 3 (TGP).
PPM_ORDER	(*3/4/5*)	Sets the order of reconstruction when using *PARABOLIC* reconstruction. Allowed values are 3 (uses three-point stencil, third-order accurate), 4 (fourth order) and 5 (fifth order). For more information see [Mig14]. The default value is 4 (as in the original PPM method).
PRNG	(*values*)	Sets the pseudo random number generator used by the **RandomNumber()** function.Allowed values are *PRNG_DEFAULT* (employs standard C **drand48()** function), *PRNG_ECUYER* (long period L'Ecuyer, adapted from *Numerical Recipe*) and *PRNG_MT* (64-bit version of Mersenne-Twister pseudo random number generator). Default is *PRNG_DEFAULT*.
PV_TEMPERATURE_TABLE	(*YES/NO*)	Used for the *PVTE_LAW* EOS in ionization equilibrium, §7.3.2. When set to *YES* replaces function evaluations of the thermal EOS ($p = nk_BT$) and its inverse with lookup table and bilinear interpolation. This results in a considerably faster execution. Default is *YES*.

Continued on next page

Table B.1 – *Continued from previous page*

Name	**Value**	**Description**
PV_TEMPERATURE_TABLE_NX	(*int*)	Sets the number of x-points used to construct the temperature table for the *PVTE_LAW* EOS. The default value is set in Src/EOS/PVTE/thermal_eos.c.
PV_TEMPERATURE_TABLE_NY	(*int*)	Sets the number of y-points used to construct the temperature table for the *PVTE_LAW* EOS. The default value is set in Src/EOS/PVTE/thermal_eos.c.
RECONSTRUCT_4VEL	(*YES/NO*)	Use the four-velocity $\boldsymbol{u} = \gamma\boldsymbol{v}$ instead of the three velocity when reconstructing left and right states in the *RHD* and *RMHD* modules. Not compatible with conservative form of MUSCL-Hancock scheme. Default is *NO*.
RKL_ORDER	(*int*)	Sets the order of integration of RKL algorithm. Set it to 1 or 2 for a 1^{st} or 2^{nd} order algorithm respectively. Default is 2.
RMHD_FAST_EIGENVALUES	(*YES/NO*)	If set to YES, use approximate (and faster) expressions when computing the fast magnetosonic speed of the RMHD equations, see Sect. 3.3 of [DZBL07]. Solutions of quartic equation is avoided and replaced with upper bounds provided by quadratic equation. Default is *NO*.
RMHD_REDUCED_ENERGY	(*YES/NO*)	Used in the RMHD module (§6.4) to evolve the total energy minus the mass density contribution (see [MM07]). This is more advisable in order to avoid numerical errors in the non-relativistic limit and catastrophic cancellation problems. Default is *YES*.
SB_OMEGA	(*real*)	[Shearing box module only, §10.1]. The value of the orbital frequency parameter Ω_0, see §10.1. The default value is `1.0`.
SB_Q	(*real*)	[Shearing box module only, §10.1]. The value of the differential shear parameter q, see §10.1. The default value is `1.5` proper for a Keplerian profile.
SB_SYMMERIZE_HYDRO	(*YES/NO*)	[Shearing box module only, §10.1]. Symmetrize the hydrodynamical fluxes at the left and right x-boundaries in order to enforce conservation of hydrodynamic variables like density, momentum and energy (no magnetic field). Default is `YES`.
SB_SYMMERIZE_EY	(*YES/NO*)	[Shearing box module only, §10.1]. Symmetrize the y-component of the electric field at the left and right x-boundaries to enforce conservation of magnetic field (only in 3D, see Src/MHD/Shearing_Box/sb_fluxes.c). Default value if `YES`.
SB_SYMMERIZE_EZ	(*YES/NO*)	[Shearing box module only, §10.1]. Symmetrize the z-component of electric field at the left and right x-boundaries to enforce conservation of magnetic field. Default is *YES*.
SHOCK_FLATTENING	*NO* *ONED* *MULTID*	Provides additional dissipation in proximity of strong shocks. When set to *ONED*, spatial slopes are progressively reduced following following a one-dimensional shock recognition pattern, as in [CW84]. This is done separately dimension by dimension. When set to *MULTID*, a multi dimensional strategy is used by which upon shock detection, i) reconstruction (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/flag_shoock.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.
SHOW_TIME_STEPS	(*YES/NO*)	Print, during the integration log, the time steps (hyperbolic, parabolic, etc...) from which the CFL condition is estimated.

Continued on next page

Table B.1 – *Continued from previous page*

Name	**Value**	**Description**
STS_NU	(*real*)	Sets the value of the ν parameter used to control the efficiency of Super-Time-Stepping integration for parabolic (diffusion) terms, see Chapter 8 and §8.5.2. If not set, the default value is 0.01.
TC_SATURATED_FLUX	(*YES/NO*)	Include saturation effects when computing the thermal conduction flux. Default value is *YES*.
T_CUT_RHOE	(*real*)	Sets the cut-off temperature (in K) used in the *PVTE_LAW* equation of state (§7.3). Zones with temperature below T_CUT_RHOE will be reset to this value and the internal energy will be redefined accordingly. Default value is 10 K.
TV_ENERGY_TABLE	(*YES/NO*)	Used for the *PVTE_LAW* EOS in ionization equilibrium, §7.3.2. When set to *YES* replaces function evaluations of the caloric EOS (internal energy) and its inverse ($e(T,\rho)$ and $T(e,\rho)$) with lookup table and bilinear interpolation. This results in a considerably faster execution. Default is *YES*.
TV_ENERGY_TABLE_NX	(*int*)	Sets the number of x-points used to construct the temperature table for the *PVTE_LAW* EOS. Default value is set in Src/EOS/PVTE/internal_energy.c.
TV_ENERGY_TABLE_NY	(*int*)	Sets the number of y-points used to construct the temperature table for the *PVTE_LAW* EOS. Default value is set in Src/EOS/PVTE/internal_energy.c.
UNIT_DENSITY	(*real*)	Sets the unit density in gr/cm^3. Default value is the proton mass per cm^3.
UNIT_LENGTH	(*real*)	Sets the unit length in cm. Default value is 1 astronomical unit.
UNIT_VELOCITY	(*real*)	Sets the unit velocity in cm/sec. Default value is 1 Km/sec.
UPDATE_VECTOR_POTENTIAL	(*YES/NO*)	Enable this option if you wish to evolve the vector potential in time and save it to disk. Note that ASSIGN_VECTOR_POTENTIAL must be enabled.
VTK_TIME_INFO	(*YES/NO*)	Enable writing of time information to .vtk output files. Notice that this information is useful only when reading data files with VisIt and may give problems with other visualisation softwares, §12.1.3. Default value is *NO*.
VTK_VECTOR_DUMP	(*YES/NO*)	Enable writing of vector fields (velocity and magnetic field) during VTK output (§12.1.3). Default value is NO (all variables are written with the scalar attribute).
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.

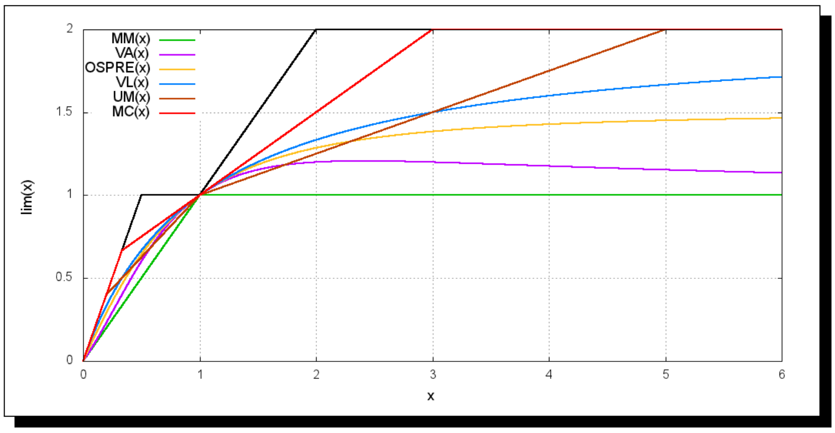


Figure B.1: Second-order TVD limiter functions used by **PLUTO** as functions of the left to right slope ratio $x = \Delta V_{i-\frac{1}{2}}/\Delta V_{i+\frac{1}{2}}$. Larger values of $\lim(x)$ indicate larger compressive behavior. In this sense, the minmod limiter (MM(x)) and the monotonized central limiter (MC(x)) are the least and most compressive, respectively.

Bibliography

[AAG96] Vasilios Alexiades, Geneviève Amiez, and Pierre-Alain Gremaud, *Super-time-stepping acceleration of explicit schemes for parabolic problems*, Communications in Numerical Methods in Engineering **12** (1996), no. 1, 31–42.

[AAZN97] T. Abel, P. Anninos, Y. Zhang, and M. L. Norman, *Modeling primordial gas in numerical cosmology*, New Astronomy**2** (1997), 181–207.

[Bal86] S. A. Balbus, *Magnetized thermal conduction fronts*, ApJ**304** (1986), 787–798.

[BCCD08] R. Borges, M. Carmona, B. Costa, and W. S. Don, *An improved weighted essentially non-oscillatory scheme for hyperbolic conservation laws*, Journal of Computational Physics **227** (2008), 3191–3211.

[BCSS15] X.-N. Bai, D. Caprioli, L. Sironi, and A. Spitkovsky, *Magnetohydrodynamic-particle-in-cell Method for Coupling Cosmic Rays with a Thermal Plasma: Application to Non-relativistic Shocks*, ApJ**809** (2015), 55.

[Bec92] J. M. Beckers, *Analytical linear numerical stability conditions for an anisotropic three-dimensional advection-diffusion equation*, SIAM Journal on Numerical Analysis **29** (1992), no. 3, 701–713.

[BS99] D. S. Balsara and D. S. Spicer, *A Staggered Mesh Algorithm Using High Order Godunov Fluxes to Ensure Solenoidal Magnetic Fields in Magnetohydrodynamic Simulations*, Journal of Computational Physics **149** (1999), 270–292.

[BS03] T. J. M. Boyd and J. J. Sanderson, *The Physics of Plasmas*, January 2003.

[BTH08] D. S. Balsara, D. A. Tilley, and J. C. Howk, *Simulating anisotropic thermal conduction in supernova remnants - I. Numerical methods*, MNRAS**386** (2008), 627–641.

[CM77] L. L. Cowie and C. F. McKee, *The evaporation of spherical clouds in a hot gas. I - Classical and saturated mass loss rates*, ApJ**211** (1977), 135–146.

[Col85] Phillip Colella, *A direct eulerian muscl scheme for gas dynamics*, SIAM Journal on Scientific and Statistical Computing **6** (1985), no. 1, 104–117.

[Col90] P. Colella, *Multidimensional upwind methods for hyperbolic conservation laws*, Journal of Computational Physics **87** (1990), 171–200.

[ČT09] M. Čada and M. Torrilhon, *Compact third-order limiter functions for finite volume methods*, Journal of Computational Physics **228** (2009), 4118–4145.

[CW84] P. Colella and P. R. Woodward, *The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations*, Journal of Computational Physics **54** (1984), 174–201.

[DBL03] L. Del Zanna, N. Bucciantini, and P. Londrillo, *An efficient shock-capturing central-type scheme for multi-dimensional relativistic flows. II. Magnetohydrodynamics*, A&A**400** (2003), 397–413.

[DKK$^+$02] A. Dedner, F. Kemm, D. Kröner, C.-D. Munz, T. Schnitzer, and M. Wesenberg, *Hyperbolic Divergence Cleaning for the MHD Equations*, Journal of Computational Physics **175** (2002), 645–673.

[DZBL07] L. Del Zanna, O. Zanotti, N. Bucciantini, and P. Londrillo, *ECHO: a Eulerian conservative high-order scheme for general relativistic magnetohydrodynamics and magnetodynamics*, A&A**473** (2007), 11–30.

[GP98] D. Galli and F. Palla, *The chemistry of the early Universe*, A&A**335** (1998), 403–420.

[GS05] T. A. Gardiner and J. M. Stone, *An unsplit Godunov method for ideal MHD via constrained transport*, Journal of Computational Physics **205** (2005), 509–539.

[HEOC87] A. Harten, B. Engquist, S. Osher, and S. R. Chakravarthy, *Uniformly high order accurate essentially non-oscillatory schemes, III*, Journal of Computational Physics **71** (1987), 231–303.

[HM79] D. Hollenbach and C. F. McKee, *Molecule formation and infrared emission in fast interstellar shocks. I Physical processes*, ApJS**41** (1979), 555–592.

[Hub05] J. D. Huba, *Numerical methods: ideal and Hall MHD*, Proceedings of ISSSS **7** (2005), 26–31.

[JS96] G.-S. Jiang and C.-W. Shu, *Efficient Implementation of Weighted ENO Schemes*, Journal of Computational Physics **126** (1996), 202–228.

[Kle98] W. Kley, *On the treatment of the Coriolis force in computational astrophysics*, A&A**338** (1998), L37–L41.

[Ld04] P. Londrillo and L. del Zanna, *On the divergence-free condition in Godunov-type schemes for ideal magnetohydrodynamics: the upwind constrained transport method*, Journal of Computational Physics **195** (2004), 17–48.

[Lio96] M.-S. Liou, *A Sequel to AUSM: AUSM $^+$*, Journal of Computational Physics **129** (1996), 364–382.

[LL87] L. D. Landau and E. M. Lifshitz, *Fluid mechanics*, 2 ed., Pergamon Press, Oxford, 1987.

[LOC94] X.-D. Liu, S. Osher, and T. Chan, *Weighted Essentially Non-oscillatory Schemes*, Journal of Computational Physics **115** (1994), 200–212.

[Loh87] R. Lohner, *An adaptive finite element scheme for transient problems in CFD*, Computer Methods in Applied Mechanics and Engineering **61** (1987), 323–338.

[MB05] A. Mignone and G. Bodo, *An HLLC Riemann solver for relativistic flows - I. Hydrodynamics*, MNRAS**364** (2005), 126–136.

[MBA12] C. D. Meyer, D. S. Balsara, and T. D. Aslam, *A second-order accurate Super TimeStepping formulation for anisotropic thermal conduction*, MNRAS**422** (2012), 2102–2115.

[MBM$^+$07] A. Mignone, G. Bodo, S. Massaglia, T. Matsakos, O. Tesileanu, C. Zanni, and A. Ferrari, *PLUTO: A Numerical Code for Computational Astrophysics*, ApJS**170** (2007), 228–242.

[MBVM18] A. Mignone, G. Bodo, B. Vaidya, and G. Mattia, *A Particle Module for the PLUTO Code: I - an implementation of the MHD-PIC equations*, ArXiv e-prints (2018).

[MFS$^+$12] A. Mignone, M. Flock, M. Stute, S. M. Kolb, and G. Muscianisi, *A conservative orbital advection scheme for simulations of magnetized shear flows with the PLUTO code*, A&A**545** (2012), A152.

[Mig07] A. Mignone, *A simple and accurate Riemann solver for isothermal MHD*, Journal of Computational Physics **225** (2007), 1427–1441.

[Mig14] ______, *High-order conservative reconstruction schemes for finite volume methods in cylindrical and spherical coordinates*, Journal of Computational Physics **270** (2014), 784–814.

[MK05] T. Miyoshi and K. Kusano, *A multi-state HLL approximate Riemann solver for ideal magnetohydrodynamics*, Journal of Computational Physics **208** (2005), 315–344.

[MM96] J. M. . Martí and E. Müller, *Extension of the Piecewise Parabolic Method to One-Dimensional Relativistic Hydrodynamics*, Journal of Computational Physics **123** (1996), 1–14.

[MM07] A. Mignone and J. C. McKinney, *Equation of state in relativistic magnetohydrodynamics: variable versus constant adiabatic index*, MNRAS**378** (2007), 1118–1130.

[MPB05] A. Mignone, T. Plewa, and G. Bodo, *The Piecewise Parabolic Method for Multidimensional Relativistic Fluid Dynamics*, ApJS**160** (2005), 199–219.

[MT10] A. Mignone and P. Tzeferacos, *A second-order unsplit Godunov scheme for cell-centered MHD: The CTU-GLM scheme*, Journal of Computational Physics **229** (2010), 2117–2138.

[MTB10] A. Mignone, P. Tzeferacos, and G. Bodo, *High-order conservative finite difference GLM-MHD schemes for cell-centered MHD*, Journal of Computational Physics **229** (2010), 5896–5920.

[MUB09] A. Mignone, M. Ugliano, and G. Bodo, *A five-wave Harten-Lax-van Leer Riemann solver for relativistic magnetohydrodynamics*, MNRAS**393** (2009), 1141–1156.

[MZT$^+$12] A. Mignone, C. Zanni, P. Tzeferacos, B. van Straalen, P. Colella, and G. Bodo, *The PLUTO Code for Adaptive Mesh Computations in Astrophysical Fluid Dynamics*, ApJS**198** (2012), 7.

[OBR$^+$08] S. Orlando, F. Bocchino, F. Reale, G. Peres, and P. Pagano, *The Importance of Magnetic-Field-Oriented Thermal Conduction in the Interaction of SNR Shocks with Interstellar Clouds*, ApJ**678** (2008), 274–286.

[Pow94] K. G. Powell, *Approximate Riemann solver for magnetohydrodynamics (that works in more than one dimension)*, Tech. report, March 1994.

[PRL$^+$99] K. G. Powell, P. L. Roe, T. J. Linde, T. I. Gombosi, and D. L. De Zeeuw, *A Solution-Adaptive Upwind Scheme for Ideal Magnetohydrodynamics*, Journal of Computational Physics **154** (1999), 284–309.

[RBMF97] P. Rossi, G. Bodo, S. Massaglia, and A. Ferrari, *Evolution of Kelvin-Helmholtz instabilities in radiative jets. II. Shock structure and entrainment properties.*, A&A**321** (1997), 672–684.

[Roe81] P. L. Roe, *Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes*, Journal of Computational Physics **43** (1981), 357–372.

[Sal94] J. Saltzman, *An Unsplit 3D Upwind Method for Hyperbolic Conservation Laws*, Journal of Computational Physics **115** (1994), 153–168.

[SG10] J. M. Stone and T. A. Gardiner, *Implementation of the Shearing Box Approximation in Athena*, ApJS**189** (2010), 142–155.

[SH97] A. Suresh and H. T. Huynh, *Accurate Monotonicity-Preserving Schemes with Runge Kutta Time Stepping*, Journal of Computational Physics **136** (1997), 83–99.

[SO89] C.-W. Shu and S. Osher, *Efficient Implementation of Essentially Non-oscillatory Shock-Capturing Schemes, II*, Journal of Computational Physics **83** (1989), 32–78.

[Spi62] L. Spitzer, *Physics of Fully Ionized Gases*, 1962.

[Str68] G. Strang, *On the Construction and Comparison of Difference Schemes*, SIAM Journal on Numerical Analysis **5** (1968), 506–517.

[Syn57] John Lighton Synge, *The relativistic gas*, North-Holland Publishing Company ; Interscience Publishers, Amsterdam; New York, 1957.

[TK15] M. Takamoto and J. G. Kirk, *Rapid Cosmic-ray Acceleration at Perpendicular Shocks in Supernova Remnants*, ApJ**809** (2015), 29.

[TMG08] G. Tóth, Y. Ma, and T. I. Gombosi, *Hall magnetohydrodynamics on block-adaptive grids*, Journal of Computational Physics **227** (2008), 6967–6984.

[TMM08] O. Teşileanu, A. Mignone, and S. Massaglia, *Simulating radiative astrophysical flows with the PLUTO code: a non-equilibrium, multi-species cooling function*, A&A**488** (2008), 429–440.

[Tor97] Eleuterio F. Toro, *Riemann solvers and numerical methods for fluid dynamics*, 2 ed., Springer-Verlag Berlin Heidelberg, 1997.

[van79] B. van Leer, *Towards the ultimate conservative difference scheme. V - A second-order sequel to Godunov's method*, Journal of Computational Physics **32** (1979), 101–136.

[VMBM15] B. Vaidya, A. Mignone, G. Bodo, and S. Massaglia, *Astrophysical fluid simulations of thermally ideal gases with non-constant adiabatic index: numerical implementation*, A&A**580** (2015), A110.

[VPM$^+$17] B. Vaidya, D. Prasad, A. Mignone, P. Sharma, and L. Rickler, *Scalable explicit implementation of anisotropic diffusion with Runge-Kutta-Legendre super-time stepping*, MNRAS**472** (2017), 3147–3160.

[WAMM07] J. Woodall, M. Agúndez, A. J. Markwick-Kemper, and T. J. Millar, *The UMIST database for astrochemistry 2006*, A&A**466** (2007), 1197–1204.

[YC09] N. K. Yamaleev and M. H. Carpenter, *A systematic methodology for constructing high-order energy stable WENO schemes*, Journal of Computational Physics **228** (2009), 4248–4272.