Scalable Parallel Astrophysical Codes for Exascale

A Practical Introduction to gPLUTO

Based on the IT4I Workshop tutorial by

A. Mignone, M. Rossazza, S. Truzzi, V. Berta,
A. Suriano, M. Bugli, G. Mattia

SPACE Center for HPC Astrophysical Applications
Physics Department, University of Torino

IT4I Workshop, 22–23 September 2025

and SPACE Winter School, 15–19 December 2025

Editors: G. Perna [ENGINSOFT], E. Sciacca, G. Taffoni [INAF].

These notes are intended for students and early-stage researchers beginning their journey with astrophysical hydrodynamics simulations.

Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and Belgium, Czech Republic, France, Germany, Greece, Italy, Norway, and Spain under grant agreement No 101093441

DISCLAIMER

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European High Performance Computing Joint Undertaking (JU) and Belgium, Czech Republic, France, Germany, Greece, Italy, Norway, and Spain. Neither the European Union nor the granting authority can be held responsible for them.

Contents

1 Introduction to gPLUTO **7**
- 1.1 Introduction . . . 7
- 1.2 Prerequisites . . . 7
  - 1.2.1 Required Software . . . 8
  - 1.2.2 Optional but Recommended . . . 8
- 1.3 Downloading gPLUTO . . . 8
  - 1.3.1 Directory Structure . . . 9
- 1.4 Getting started: The Shock-Tube Problem . . . 9
  - 1.4.1 Setting the PLUTO_DIR Environment Variable . . . 9
  - 1.4.2 The Four Configuration Steps . . . 10
- 1.5 Using the Python Setup Script . . . 10
  - 1.5.1 Main Menu . . . 10
  - 1.5.2 The "Setup Problem" Menu . . . 10
  - 1.5.3 The "HD" Physics Sub-menu . . . 11
  - 1.5.4 User-Defined Parameters . . . 11
  - 1.5.5 Choosing a Makefile . . . 12
- 1.6 Manual Configuration Files . . . 12
  - 1.6.1 Advanced Options in `definitions.hpp` . . . 12
  - 1.6.2 The `init.cpp` File: Initial Conditions . . . 13
  - 1.6.3 The `pluto.ini` File: Runtime Parameters . . . 13
- 1.7 Compiling and Running gPLUTO . . . 14
  - 1.7.1 Compiling . . . 14
  - 1.7.2 Running . . . 15
  - 1.7.3 Reading the Output Log . . . 15
- 1.8 Visualizing the Results . . . 15
  - 1.8.1 Visualization with Gnuplot . . . 16
  - 1.8.2 Visualization with PyPLUTO . . . 17
- 1.9 Exercises . . . 19
- 1.10 Summary . . . 19

2 The MHD Blast Wave Problem **21**
- 2.1 Introduction . . . 21
- 2.2 Physics Background: Ideal MHD . . . 21
  - 2.2.1 The Ideal MHD Equations . . . 21
  - 2.2.2 The Plasma-Beta Parameter . . . 22
  - 2.2.3 The Divergence-Free Constraint and Constrained Transport . . . 22
- 2.3 Problem Description . . . 22
  - 2.3.1 Physical Setup . . . 22
  - 2.3.2 Expected Dynamics . . . 23
- 2.4 Setting Up the Problem . . . 23
  - 2.4.1 Navigating to the Test Problem . . . 23

2.4.2 The Setup Menu . . . 23
2.4.3 The MHD Physics Sub-menu . . . 24
2.4.4 User-Defined Parameters . . . 24
2.5 Specifying Initial Conditions in init.cpp . . . 24
2.5.1 Overview . . . 24
2.5.2 The Init() Function . . . 24
2.5.3 Explanation of Key Lines . . . 25
2.6 Runtime Parameters: pluto.ini . . . 26
2.6.1 What Is New Compared to Chapter 1? . . . 27
2.7 Compiling and Running . . . 27
2.7.1 Serial Run . . . 27
2.7.2 Parallel Run with MPI . . . 27
2.8 Visualizing the Results . . . 28
2.8.1 Gnuplot: 2D Colour Maps and Contours . . . 28
2.8.2 PyPLUTO: 2D Colour Maps and Contours . . . 29
2.9 Parameter Study: The Role of Plasma Beta . . . 30
2.9.1 The Experiment . . . 30
2.9.2 Results and Physical Interpretation . . . 30
2.9.3 Summary of Results . . . 31
2.10 Exercises . . . 32
2.11 Summary . . . 32

3 Kelvin–Helmholtz Instability 33
3.1 Introduction . . . 33
3.2 Physics Background . . . 34
3.2.1 Governing Equations and Initial Configuration . . . 34
3.2.2 Linear Theory: the Vortex-Sheet Case ($a \to 0$) . . . 34
3.2.3 Linear Theory: the Finite Shear Layer ($a \neq 0$) . . . 35
3.2.4 Linear Theory: Effect of a Magnetic Field . . . 36
3.2.5 Nonlinear Evolution . . . 37
3.3 Setting Up the Problem . . . 37
3.3.1 Navigating to the Test Problem . . . 37
3.3.2 The Setup Menu . . . 37
3.3.3 User-Defined Parameters . . . 38
3.4 Specifying Initial Conditions: the InitDomain() Function . . . 38
3.4.1 Two Functions for Initial Conditions . . . 39
3.4.2 The InitDomain() Function . . . 39
3.4.3 Key Points . . . 40
3.5 Runtime Parameters: pluto.ini . . . 40
3.5.1 Notable Features of This File . . . 41
3.6 The Analysis() Function: On-the-Fly Diagnostics . . . 42
3.6.1 What the Analysis() Function Does . . . 42
3.6.2 Output Format: khi.dat . . . 42
3.6.3 Measuring the Growth Rate . . . 42
3.7 Compiling and Running . . . 43
3.8 Visualization . . . 43
3.8.1 Gnuplot: Animation with khi.gp . . . 43
3.8.2 PyPLUTO: Interactive Animation . . . 43
3.9 Analysis and Exercises . . . 45
3.9.1 Identifying the Phases of the KHI . . . 46
3.9.2 Measuring and Comparing the Growth Rate . . . 46
3.9.3 Suppression by a Parallel Magnetic Field . . . 46

3.9.4 Effect of Field Geometry: Perpendicular Field . . . 46
3.9.5 Growth Rate vs. Resolution and Riemann Solver . . . 46
3.10 Summary . . . 47

4 MHD Jet Propagation 49
4.1 Introduction . . . 49
4.2 Physics Background: Supersonic Jet Propagation . . . 49
4.2.1 Anatomy of a Supersonic Jet . . . 49
4.2.2 Key Dimensionless Parameters . . . 50
4.3 Physical Setup and Coordinate System . . . 51
4.3.1 Cylindrical Coordinates . . . 51
4.3.2 Domain and Boundary Conditions . . . 51
4.3.3 Initial and Ambient Conditions . . . 52
4.4 Setting Up the Problem . . . 52
4.4.1 Navigating to the Test Problem and Running Setup . . . 52
4.4.2 The Setup Menu . . . 52
4.4.3 User-Defined Parameters . . . 52
4.5 Specifying Initial and Boundary Conditions . . . 53
4.5.1 The Init() Function: Ambient Values . . . 53
4.5.2 The Boundary Condition Architecture . . . 53
4.5.3 The GetJetValues() Function . . . 54
4.5.4 The UserDefBoundary() Function . . . 55
4.6 Runtime Parameters: pluto.ini . . . 56
4.6.1 Notes on the Grid and Parameters . . . 57
4.7 Compiling and Running . . . 57
4.8 Comparing Jet Morphologies . . . 57
4.8.1 Case 1: Ballistic Heavy Jet ($\eta = 10$, $\sigma_\phi = 0.01$) . . . 57
4.8.2 Case 2: Under-dense, Weakly Magnetized ($\eta = 0.05$, $\sigma_\phi = 0.01$) . . . 58
4.8.3 Case 3: Under-dense, Strongly Magnetized ($\eta = 0.05$, $\sigma_\phi = 10$) . . . 58
4.9 The 3D Jet and the Kink Instability . . . 58
4.9.1 Motivation . . . 59
4.9.2 Setting Up the 3D Run . . . 59
4.9.3 Recommended Makefile and HPC Options . . . 59
4.10 Visualization . . . 59
4.10.1 2D Runs: PyPLUTO Density Maps . . . 59
4.10.2 3D Runs: VisIt and ParaView . . . 60
4.11 Exercises . . . 61
4.12 Summary . . . 61

5 HPC Environment Setup: NVIDIA HPC-SDK, GPU Acceleration, and Cluster Job Submission 63
5.1 Overview . . . 63
5.2 Installing the NVIDIA HPC-SDK on a Local Machine . . . 63
5.2.1 Downloading and Installing . . . 63
5.2.2 Configuring the Environment . . . 64
5.2.3 Verifying the Installation . . . 64
5.3 The IT4I HPC Systems . . . 65
5.3.1 Hardware Summary . . . 65
5.4 Connecting to Karolina and Barbora . . . 66
5.4.1 Initial SSH Login . . . 66
5.4.2 Enabling X11 Forwarding for GUI Applications . . . 66
5.5 Setting Up the Software Environment on Karolina and Barbora . . . 66

5.5.1 Step 1: Download gPLUTO . . . 67
5.5.2 Step 2: Set the PLUTO_DIR Environment Variable . . . 67
5.5.3 Step 3: Create a Python Virtual Environment and Load Modules . . . 67
5.5.4 Step 4: Export NCCL Library Paths . . . 67
5.6 Slurm Job Submission on Karolina and Barbora . . . 68
5.6.1 Understanding the Partitions . . . 68
5.6.2 The sbatch Script . . . 68
5.6.3 Key Slurm Directives Explained . . . 69
5.6.4 Useful Slurm Commands . . . 70
5.7 Interactive GPU Nodes at UniTo HPC4AI . . . 70
5.7.1 Node Specifications . . . 70
5.7.2 How to Connect . . . 70
5.7.3 After Connecting . . . 70
5.8 Running gPLUTO on Jureca at JSC . . . 71
5.8.1 Step-by-Step Workflow . . . 71
5.9 Choosing the Right Makefile for GPU Runs . . . 72
5.10 Quick-Reference Checklist . . . 72
5.11 Summary . . . 73

Chapter 1

Introduction to gPLUTO

1.1 Introduction

gPLUTO is the GPU-enabled version of the PLUTO code, a modular, open-source framework for solving the equations of computational astrophysics and plasma physics. Whether you are studying shock waves, astrophysical jets, or magnetized plasma flows, gPLUTO provides a flexible environment where you define the physics, the geometry, and the initial conditions — and the code handles the numerics for you.

This chapter walks you through everything you need to run your first simulation: downloading and installing the code, understanding its directory structure, configuring a problem, running it, and visualizing the results. We use the classic **Sod Shock Tube** problem as our working example — a benchmark that is simple enough to understand analytically yet rich enough to test the numerical solver.

Learning Objectives

By the end of this chapter you will be able to:

- Download, install, and navigate the gPLUTO directory structure
- Configure a simulation using the interactive Python setup script
- Understand the four key configuration files: `definitions.hpp`, `makefile`, `pluto.ini`, and `init.cpp`
- Compile and run gPLUTO and interpret the output log
- Visualize simulation data using Gnuplot and PyPLUTO

1.2 Prerequisites

gPLUTO runs on any **Linux-based system**. Before you begin, make sure the following software is available on your machine. These are standard tools on any scientific Linux workstation and are available through your package manager (`apt`, `yum`, `pacman`, etc.).

1.2.1 Required Software

Package	Purpose
gPLUTO	The simulation code itself (download via GitLab)
C++ compiler (`g++` or `nvc++`)	Compiles the source code. `g++` is the standard GNU compiler; `nvc++` is NVIDIA's compiler for GPU acceleration
`make`	Automates the compilation process using a Makefile
Python 3	Runs the interactive setup script (`setup.py`)
Gnuplot	Quick and easy visualization of 1D ASCII output files

1.2.2 Optional but Recommended

Package	Purpose
VisIt3	Powerful open-source tool for 3D/2D visualization of VTK and HDF5 files (https://wci.llnl.gov/simulation/computer-codes/visit/)
PyPLUTO	Python library for loading and plotting gPLUTO data. Included in `Tools/PyPLUTO/`

Tip for new users

If you are just starting out, do not worry about VisIt or PyPLUTO right away. Gnuplot is sufficient for 1D problems and is almost certainly already installed on your system. You can add the other tools later as your simulations grow in complexity.

1.3 Downloading gPLUTO

gPLUTO is hosted on a public GitLab repository. To download it, open a terminal and run:

```
git clone https://gitlab.com/PLUTO-code/gPLUTO
```

This creates a directory called `gPLUTO/` in your current location. Navigate inside and inspect its contents:

```
cd gPLUTO
ls -l
```

You should see the following top-level structure:

```
Config/         Doc/            Lib/
Src/            Tools/          Test_Problems/
setup.py
```

1.3.1 Directory Structure

Directory / File	Contents and Purpose
`Config/`	Machine-specific configuration files (compiler flags, library paths, etc.). Used by the Python setup script to generate your Makefile
`Doc/`	Documentation, including the user guide PDF. Start here if you have questions not covered in these notes
`Lib/`	Optional external libraries used by some physics modules
`Src/`	The main C++ source code — all `.cpp` and `.hpp` files. The one exception is `init.cpp`, which lives in your working directory and is yours to modify
`Tools/`	Useful scripts and tools: Python utilities, PyPLUTO, Gnuplot scripts, IDL routines, etc.
`Test_Problems/`	A library of worked example problems organized by physics module (HD, MHD, RHD...). These are your best learning resource and starting templates
`setup.py`	The interactive Python script for configuring your problem and generating the Makefile

1.4 Getting started: The Shock-Tube Problem

The **Sod shock tube** is the "Hello World" of computational fluid dynamics. It models a 1D tube divided at its midpoint ($x = 0.5$) into two regions with different pressure and density. When the dividing membrane is removed at $t = 0$, the initial discontinuity breaks into three characteristic waves:

- a **rarefaction wave** travelling to the left,
- a **contact discontinuity** propagating at intermediate speed,
- a **shock wave** moving to the right.

This test is ideal for verifying that the Riemann solver works correctly because the exact analytical solution is known. The standard Sod initial conditions are:

Region	Density ρ	Velocity v_x	Pressure P
Left ($x < 0.5$)	1.0	0.0	1.0
Right ($x \geq 0.5$)	0.125	0.0	0.1

1.4.1 Setting the PLUTO_DIR Environment Variable

gPLUTO needs to know where its source files live. Set the `PLUTO_DIR` environment variable to point to the root of your installation. You can add this line to your `.bashrc` or `.bash_profile` so it is set automatically each time you open a terminal:

```
export PLUTO_DIR=/<full_path>/gPLUTO
# Example: export PLUTO_DIR=/home/alice/codes/gPLUTO
```

Then move to the shock tube test problem directory:

```
cd $PLUTO_DIR/Test_Problems/HD/Shock_Tubes
```

1.4.2 The Four Configuration Steps

Every gPLUTO simulation requires four configuration files. Two are generated interactively via the Python script; the other two are set manually:

#	File	Method	Purpose
1	`definitions.hpp`	Python script	Compile-time switches: physics module, geometry, numerical methods
2	`makefile`	Python script	Compilation instructions adapted to your machine and compiler
3	`pluto.ini`	Edit manually	Runtime parameters: grid, time, boundary conditions, output options
4	`init.cpp`	Edit manually	C++ code defining initial and boundary conditions

Key Concept: Compile-time vs. Run-time

Compile-time options (in `definitions.hpp`) are baked into the executable when you run `make`. Changing them requires recompiling. **Run-time options** (in `pluto.ini`) are read by the code every time it starts, so you can change things like grid resolution or stop time without recompiling.

1.5 Using the Python Setup Script

The easiest way to create `definitions.hpp` and the `makefile` is via the interactive Python setup script. The test problem directory already contains pre-made example configurations. Copy one to use as your starting point, then launch the script:

```
cp definitions_01.hpp definitions.hpp
cp pluto_01.ini pluto.ini
python3 $PLUTO_DIR/setup.py
```

1.5.1 Main Menu

The script opens a text-based menu. The most important options are:

- **Setup problem** — configure compile-time physics options (creates/updates `definitions.hpp`)
- **Change makefile** — select the appropriate compiler configuration for your machine
- **Auto-update** — re-run the setup with the existing configuration (useful after editing `definitions.hpp` manually)
- **Save Setup** — write configuration to disk
- **Quit** — exit the script

1.5.2 The "Setup Problem" Menu

Selecting *Setup problem* takes you to a screen listing all compile-time switches:

Option	Description
PHYSICS	The equations to solve. HD = hydrodynamics (Euler equations); MHD = magnetohydrodynamics; RHD = relativistic HD, etc.
DIMENSIONS	Spatial dimensionality: 1, 2, or 3
GEOMETRY	Coordinate system: CARTESIAN, CYLINDRICAL, SPHERICAL, etc.
MAPPED_GRID	Enable non-uniform (stretched) grids (YES/NO)
BODY_FORCE	Include gravitational or other body forces
COOLING	Enable radiative cooling in the energy equation
RECONSTRUCTION	Spatial interpolation: FLAT (1st order), LINEAR (2nd order), PARABOLIC (3rd order)
TIME_STEPPING	Time integration: EULER (1st order), RK2 (2nd order), RK3 (3rd order)
NTRACER	Number of passive scalar tracers (0 if not needed)
PARTICLES	Enable Lagrangian particle module (YES/NO)
USER_DEF_PARAMETERS	Number of user-defined runtime parameters accessible from pluto.ini

1.5.3 The "HD" Physics Sub-menu

When PHYSICS = HD is selected, a sub-menu appears with options specific to hydrodynamics:

Option	Description
DUST_FLUID	Enable a pressureless dust fluid component (YES/NO)
EOS	Equation of state. IDEAL uses gamma-law $P = (\gamma - 1)\rho e$; ISOTHERMAL removes the energy equation
ENTROPY_SWITCH	Solve the entropy equation instead of total energy (more robust in smooth flows, less so near shocks)
THERMAL_CONDUCTION	Add heat conduction term to the energy equation
VISCOSITY	Include viscous stress tensor, i.e. solve Navier–Stokes equations
ROTATING_FRAME	Simulate in a rotating reference frame (adds Coriolis and centrifugal terms)

For the Sod shock tube, we use EOS = IDEAL and set all dissipative options to NO.

1.5.4 User-Defined Parameters

The setup script also shows the names of the user-defined runtime parameters. For the shock tube, there are 10 parameters (indexed 0–9) that let you set the left and right states directly in pluto.ini without recompiling:

Index	Name	Meaning
0	RHO_LEFT	Density in the left region
1	VX1_LEFT	x-velocity in the left region
2	VX2_LEFT	y-velocity in the left region
3	VX3_LEFT	z-velocity in the left region
4	PRS_LEFT	Gas pressure in the left region
5	RHO_RIGHT	Density in the right region
6	VX1_RIGHT	x-velocity in the right region
7	VX2_RIGHT	y-velocity in the right region
8	VX3_RIGHT	z-velocity in the right region
9	PRS_RIGHT	Gas pressure in the right region

1.5.5 Choosing a Makefile

From the main menu, select *Change makefile*. You will see a list of pre-defined configuration files from the `Config/` directory:

Configuration file	When to use it
`linux.g++.defs`	Standard serial compilation with the GNU `g++` compiler. Start here if in doubt
`linux.mpicxx.defs`	Parallel compilation using MPI with the GNU compiler (multi-core / multi-node runs)
`linux.nvc++.defs`	Serial GPU compilation using NVIDIA's `nvc++` compiler
`linux.nvc++.acc.defs`	GPU compilation with OpenACC acceleration (NVIDIA GPUs)
`linux.mpicxx.acc.defs`	MPI + OpenACC for multi-GPU runs
`linux.nvc++.debug.defs`	Debug build with `nvc++` (slower but more error information)

1.6 Manual Configuration Files

1.6.1 Advanced Options in `definitions.hpp`

After running the setup script, you can open `definitions.hpp` in any text editor and add further options in the user-defined constants section:

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

#define GNUPLOT_HEADER  YES          // Write pluto.gp for easy plotting
#define LIMITER         VANLEER_LIM  // Piecewise linear slope limiter

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

Constant	Effect
GNUPLOT_HEADER YES	Writes a file pluto.gp at runtime containing grid information and variable names; makes Gnuplot plotting much easier
LIMITER VANLEER_LIM	Sets the slope limiter used in piecewise-linear reconstruction. Other options include MINMOD_LIM, MC_LIM. Van Leer is a good general-purpose choice

Many more options are described in the userguide.pdf *in the* Doc/ *directory.*

1.6.2 The init.cpp File: Initial Conditions

The file init.cpp is the heart of your problem setup. It lives in your working directory (not in Src/) and is the only C++ file you write yourself. The main function you must implement is Init(), called once for every grid cell at the start of the simulation.

Here is the Init() function for the Sod shock tube:

```
void Init(double *v, double x1, double x2, double x3,
          RunConfig &run_config)
{
  double *inputParam = run_config.inputParam;
  run_config.gamma = 1.4;       // Adiabatic index

  if (x1 < 0.5) {               // Left state
    v[RHO] = inputParam[RHO_LEFT];
    v[VX1] = inputParam[VX1_LEFT];
    v[VX2] = inputParam[VX2_LEFT];
    v[VX3] = inputParam[VX3_LEFT];
    #if EOS == IDEAL
    v[PRS] = inputParam[PRS_LEFT];
    #endif
  } else {                       // Right state
    v[RHO] = inputParam[RHO_RIGHT];
    v[VX1] = inputParam[VX1_RIGHT];
    v[VX2] = inputParam[VX2_RIGHT];
    v[VX3] = inputParam[VX3_RIGHT];
    #if EOS == IDEAL
    v[PRS] = inputParam[PRS_RIGHT];
    #endif
  }
}
```

The function receives the physical coordinates (x1, x2, x3) of the cell and a pointer v to the array of primitive variables. You set each variable based on position. The inputParam array gives access to the user-defined parameters from pluto.ini, so you can change initial conditions without recompiling.

1.6.3 The pluto.ini File: Runtime Parameters

The file pluto.ini is read by gPLUTO every time you launch the code. It is organized into named blocks:

Block	Key options
[Grid]	Domain extent and number of cells in each direction. Example: X1-grid 0.0 400 1.0 means 400 uniform cells from $x = 0$ to $x = 1$
[Time]	CFL number (stability condition, typically 0.4–0.8), maximum simulation time (tstop), first time step (first_dt)
[Solver]	The Riemann solver to use (roe, hll, hllc, tvdlf, etc.)
[Boundary]	Boundary conditions at each domain edge (outflow, periodic, reflective, etc.)
[Static Grid Output]	What output files to write and at what cadence (dbl = double binary, flt = single, tab = ASCII, vtk = VTK for VisIt)
[Parameters]	Values of your user-defined runtime parameters (e.g. RHO_LEFT = 1.0)

A minimal pluto.ini for the Sod test:

```
[Grid]
X1-grid    0.0   400   1.0     # 400 cells, x from 0 to 1
X2-grid    0.0     1   1.0
X3-grid    0.0     1   1.0

[Time]
CFL           0.8
tstop         0.2
first_dt      1.e-4

[Solver]
Solver        roe

[Boundary]
X1-beg        outflow
X1-end        outflow

[Parameters]
RHO_LEFT      1.0
PRS_LEFT      1.0
RHO_RIGHT     0.125
PRS_RIGHT     0.1
VX1_LEFT      0.0
VX1_RIGHT     0.0
```

What is the CFL number?

The Courant–Friedrichs–Lewy (CFL) number controls the time step. At each step, gPLUTO chooses $\Delta t = \text{CFL} \times \Delta x / v_{\max}$. A value between 0.4 and 0.8 is typical. Too large and the simulation may become unstable; too small and it runs unnecessarily slowly.

1.7 Compiling and Running gPLUTO

1.7.1 Compiling

Once your four configuration files are in place, compile the code using make:

```
make -j    # -j enables parallel compilation (faster on multi-core machines)
```

This produces an executable called `pluto` in your working directory. You will need to recompile only when you change `definitions.hpp` or `init.cpp`. Changes to `pluto.ini` take effect immediately without recompiling.

1.7.2 Running

Launch the simulation:

```
./pluto                  # Normal run
./pluto > out.log        # Redirect all output to a log file
```

Several useful command-line options are available:

Option	Effect
`-xres n1`	Override the X1 grid resolution in `pluto.ini` to `n1` cells. Grid in other directions is rescaled to keep square cells
`-maxsteps n`	Stop after `n` time steps (useful for quick testing)
`-no-write`	Run without writing any output files (useful for benchmarking)

1.7.3 Reading the Output Log

As gPLUTO runs, it prints a line for every time step:

```
> nstep = 0; t = 0.000000e+00; dt = 1.000000e-04; 0.0 %
  Max(MACH) = 0.140992
  Max(vc) = 1.000000e+00 1.303189e-01 0.000000e+00 0.000000e+00 1.000000e+00
  Min(vc) = 1.250000e-01 0.000000e+00 0.000000e+00 0.000000e+00 1.000000e-01
...
> nstep = 233; t = 1.996363e-01; dt = 3.636627e-04; 99.8 %
  Max(MACH) = 1.100734
...
> Total allocated memory   0.10 Mb
> Elapsed time (sec): 0.031060
> Done.
```

Field	Meaning
`nstep`	Current time step number
`t`	Current simulation time
`dt`	Current time step size (determined by CFL condition)
`99.8%`	Percentage of `tstop` elapsed
`Max(MACH)`	Maximum Mach number in the domain — useful for monitoring shocks
`Max(vc) / Min(vc)`	Maximum and minimum of each primitive variable: ρ, v_{x1}, v_{x2}, v_{x3}, P
`Done.`	Simulation completed successfully

1.8 Visualizing the Results

gPLUTO can write output in several formats. The right visualization tool depends on the format and the dimensionality of your problem:

Tool	Free?	File formats	Analysis?	Best for
Gnuplot	Yes	ASCII (`.tab`), Binary	No	Quick 1D plots; always available
PyPLUTO	Yes	All gPLUTO formats	Yes	Analysis and Python-based plots
VisIt	Yes	VTK, HDF5	No	Interactive 2D/3D visualization
Paraview	Yes	VTK, HDF5	No	Large-scale 3D visualization
IDL	No	All	Yes	Legacy tool; not recommended

1.8.1 Visualization with Gnuplot

Gnuplot is the fastest way to inspect results for 1D problems. The ASCII `.tab` output files have a simple column structure: column 1 is the x-coordinate, and columns 3, 4, 5, 6, 7 are ρ, v_{x1}, v_{x2}, v_{x3}, and P respectively.

Plot the initial condition:

```
gnuplot> plot "data.0000.tab" using 1:3
```

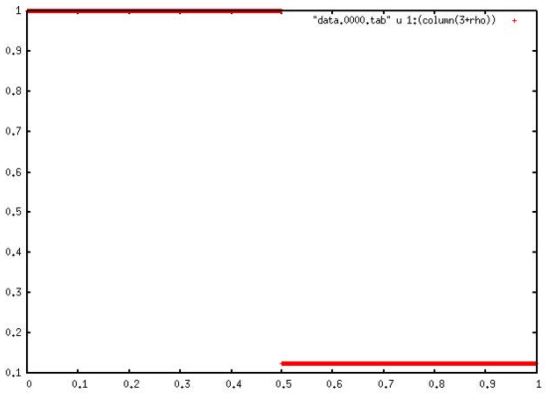


Figure 1.1: Plot of the initial conditions

Plot density at the 10th output:

```
gnuplot> plot "data.0010.tab" using 1:3
```

If you set `GNUPLOT_HEADER YES` in `definitions.hpp`, gPLUTO writes a helper file `pluto.gp` that defines named variables for each column:

```
gnuplot> dtype='tab'                          # Select file format
gnuplot> load "pluto.gp"                      # Load variable definitions
gnuplot> plot "data.0005.tab" u 1:(column(vx1)) # Plot velocity by name
```

Variables in the shock tube are named: `rho` (density), `vx1` (x-velocity), `prs` (gas pressure).

To produce an animation, first set the environment variable:

```
export GNUPLOT_LIB=$PLUTO_DIR/Tools/Gnuplot
```

then load the animation script:

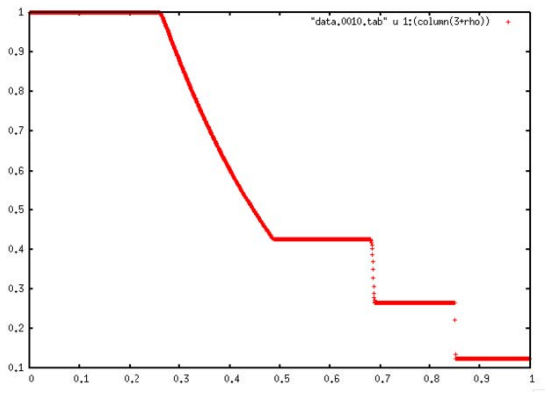


Figure 1.2: Density plot at step 10

```
gnuplot> load "shock_tubes.gp"
```

1.8.2 Visualization with PyPLUTO

PyPLUTO is a Python package that provides a high-level interface to gPLUTO output files. It requires Python 3.10 or higher.

Installation

It is good practice to work inside a virtual environment:

```
# Option 1: Python venv
python3 -m venv workshop
source workshop/bin/activate

# Option 2: conda
conda create -n workshop
conda activate workshop

# Install PyPLUTO
cd $PLUTO_DIR/Tools/PyPLUTO
pip install ./
```

This automatically installs all required dependencies: `numpy`, `matplotlib`, `scipy`, `pandas`, and others.

Basic Usage

```
import pyPLUTO as pp

D = pp.Load(0)                      # Load output #0 (initial condition)
I = pp.Image().plot(D.x1, D.rho)    # Plot density vs x
pp.show(block=False)

D = pp.Load()                       # Load the last available output
```

Variables are attributes of the `Load` object: `D.rho`, `D.vx1`, `D.prs`, etc. Grid coordinates are `D.x1`, `D.x2`, `D.x3`.

To plot multiple variables on the same axes:

```
import pyPLUTO as pp

D = pp.Load()
```

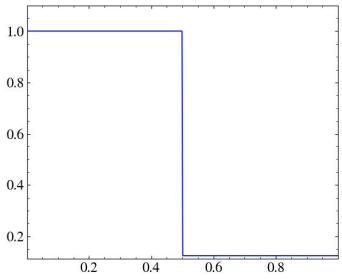


Figure 1.3: Configuration at step 0

```
I = pp.Image()
I.plot(D.x1, D.rho, label='rho', legpos=0)
I.plot(D.x1, D.prs, xtitle='x')
pp.show()
```

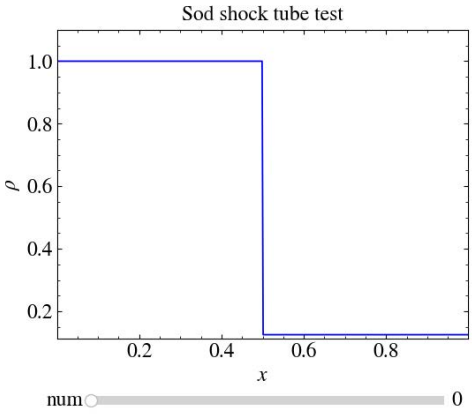


Figure 1.4: PyPLUTO anumation example

For in-line documentation:

```
help(pp.Image().plot)     # Detailed help for the plot function
print(pp.Image())         # Summary of the Image class
```

To produce an animation:

```
python ST_animation.py
```

The PyPLUTO GUI

PyPLUTO also includes a graphical user interface for quick inspection without writing Python code:

```
pypluto-gui
```

The GUI allows you to select any output file, choose the variable to plot, adjust axis ranges and colour maps, and overlay multiple time steps — all interactively.

1.9 Exercises

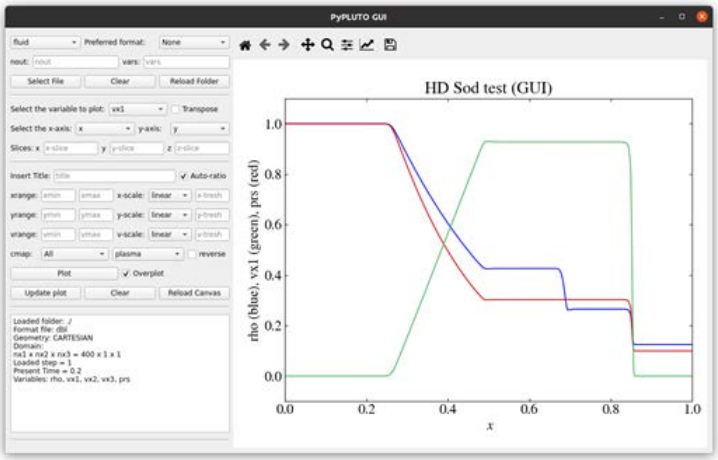


Figure 1.5: PyPLUTO GUI

Once you have successfully run the default Sod test, try the following experiments to build intuition about the code and the physics:

1. **Change the initial conditions.** Modify `pluto.ini` to use a stronger pressure ratio (e.g. `PRS_LEFT = 10.0`, `PRS_RIGHT = 0.1`) and rerun. Does the shock travel faster? Does the contact discontinuity sharpen or spread?
2. **Change the grid resolution.** Try running with `-xres 100` and `-xres 800` and compare the density plots. How does resolution affect the sharpness of the shock and the contact wave?
3. **Compare Riemann solvers.** Change `Solver` in `pluto.ini` from `roe` to `hll` and rerun. Plot the density profile for both. You will notice that HLL smears the contact discontinuity — this is a known property of the HLL solver.
4. **Simulate an isolated contact wave.** Set up a test where only the density differs between left and right (same pressure and velocity on both sides). A pure Roe or HLLC solver should track the contact sharply, while HLL will diffuse it significantly.
5. **Explore output formats.** Enable VTK output in `pluto.ini` by adding a `vtk` line under `[Static Grid Output]`, then try opening the result in VisIt.

1.10 Summary

This chapter covered the full workflow for setting up, running, and visualizing a gPLUTO simulation. The key points to remember are:

- **Four files define a simulation:** `definitions.hpp` (compile-time physics), `makefile` (compiler settings), `pluto.ini` (runtime parameters), and `init.cpp` (initial conditions).
- **The Python setup script** (`setup.py`) generates the first two files interactively. You write the last two manually.
- **Compile once, run many times:** you only need to recompile when changing `definitions.hpp` or `init.cpp`. Changes to `pluto.ini` take effect immediately.
- **The output log** tells you the current step, time, time step size, completion percentage, and maximum variable values. Watch the Mach number for signs of numerical trouble.
- **Gnuplot** is quick and always available for 1D inspection; **PyPLUTO** offers full Python integration and a GUI for more thorough analysis.

What comes next?

In the next chapter, we will move to 2D problems and explore more complex physics, including the Kelvin–Helmholtz instability and magnetohydrodynamics. You will also learn how to define custom boundary conditions and use the analysis output.

Chapter 2

The MHD Blast Wave Problem

2.1 Introduction

In Chapter 1 we ran a purely hydrodynamic simulation in one spatial dimension. This chapter takes two simultaneous steps forward: we move to **two spatial dimensions** and we switch from hydrodynamics (HD) to **magnetohydrodynamics (MHD)**.

The test problem is the **MHD blast wave**. A circular, over-pressurized region sits at the centre of a 2D Cartesian domain threaded by a uniform horizontal magnetic field. When the simulation starts, the pressure imbalance drives a rapidly expanding blast wave whose shape is strongly influenced by the ambient magnetic field. This problem is a classic benchmark precisely because it reveals — in a vivid and visual way — how magnetism constrains plasma motion.

Learning Objectives

By the end of this chapter you will be able to:

- Set up and run a 2D MHD simulation in gPLUTO
- Understand the role of the constrained transport method in controlling $\nabla \cdot \mathbf{B}$
- Implement 2D initial conditions in `init.cpp`, including magnetic field and vector potential initialization
- Configure `pluto.ini` for a 2D problem, including binary and HDF5 output formats
- Run gPLUTO in parallel using MPI
- Produce 2D colour maps and contour plots with Gnuplot and PyPLUTO
- Interpret the effect of the plasma-β parameter on blast wave morphology

2.2 Physics Background: Ideal MHD

2.2.1 The Ideal MHD Equations

In ideal (non-dissipative) MHD, the standard fluid equations are extended to include the magnetic field $\mathbf{B}$ as a dynamical variable. In conservative form, the system reads:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \qquad \text{(mass)} \tag{2.1}$$

$$\frac{\partial (\rho \mathbf{v})}{\partial t} + \nabla \cdot (\rho \mathbf{v}\mathbf{v} - \mathbf{B}\mathbf{B} + \mathbf{I}\, P_{\rm tot}) = 0 \qquad \text{(momentum)} \tag{2.2}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot [(E + P_{\rm tot})\mathbf{v} - \mathbf{B}(\mathbf{v} \cdot \mathbf{B})] = 0 \qquad \text{(energy)} \tag{2.3}$$

$$\frac{\partial \mathbf{B}}{\partial t} - \nabla \times (\mathbf{v} \times \mathbf{B}) = 0 \qquad \text{(induction)} \tag{2.4}$$

where the **total pressure** $P_{\rm tot} = P + |\mathbf{B}|^2/2$ combines thermal gas pressure P and magnetic pressure $|\mathbf{B}|^2/2$ (in units where $\mu_0 = 1$). The total energy density is

$$E = \frac{P}{\gamma - 1} + \frac{1}{2}\rho|\mathbf{v}|^2 + \frac{|\mathbf{B}|^2}{2}.$$

The induction equation implies the **divergence-free constraint** $\nabla \cdot \mathbf{B} = 0$, which must be actively enforced in any numerical scheme.

2.2.2 The Plasma-Beta Parameter

The **plasma beta** β measures the ratio of thermal to magnetic pressure:

$$\beta = \frac{P}{|\mathbf{B}|^2/2}.$$

- $\beta \gg 1$: thermally dominated plasma; the magnetic field is dynamically weak and the flow is nearly isotropic.
- $\beta \sim 1$: thermal and magnetic forces are comparable.
- $\beta \ll 1$: magnetically dominated plasma; the field strongly constrains and anisotropizes the flow.

Varying β is the primary control parameter in this problem.

2.2.3 The Divergence-Free Constraint and Constrained Transport

A key challenge in numerical MHD is maintaining $\nabla \cdot \mathbf{B} = 0$ throughout the simulation. Spurious magnetic monopoles generated by truncation errors can exert non-physical forces and eventually destabilise the solution.

gPLUTO implements **Constrained Transport (CT)**, the gold standard method for structured-grid MHD codes. CT stores the magnetic field components on cell *faces* and evolves them using the electric field on cell *edges*. By construction, the discrete divergence is preserved to machine precision at every time step.

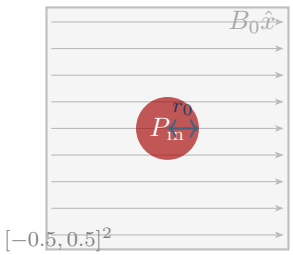
Why $\nabla \cdot \mathbf{B} = 0$ matters

Magnetic monopoles do not exist in nature. If a numerical scheme generates them, they act as spurious source terms that accelerate the fluid non-physically, leading to incorrect dynamics and eventual blow-up. CT eliminates this issue entirely by encoding the no-monopole condition into the discrete update algorithm itself.

2.3 Problem Description

2.3.1 Physical Setup

The computational domain is the 2D Cartesian square $[-0.5, 0.5]^2$. The ambient medium is a uniform, low-pressure fluid threaded by a horizontal magnetic field $\mathbf{B} = B_0\,\hat{x}$. A circular region of radius $r_0 = 0.1$ centred at the origin is initialized to a high pressure $P_{\rm in}$ while the density and magnetic field remain unchanged everywhere.



2.3.2 Expected Dynamics

When the simulation starts, the pressure imbalance drives an expanding blast wave. Because the field is horizontal, it resists compression perpendicular to $\hat{x}$ (vertical direction) while offering little resistance along $\hat{x}$. The blast wave therefore develops a characteristic **anisotropic** structure:

- An **outer fast magnetosonic shock** propagates outward; its shape ranges from nearly circular (high β) to strongly elongated along $\hat{x}$ (low β).
- **Magnetic field lines pile up** behind the outer shock as the ambient field is compressed, building a shell of enhanced magnetic pressure.
- An **inner contact discontinuity** separates the hot blast material from the compressed shell.

Magnetic tension and pressure anisotropy

The magnetic force has two components: an isotropic magnetic pressure $-\nabla(|\mathbf{B}|^2/2)$ and a tension force $(\mathbf{B}\cdot\nabla)\mathbf{B}/\mu_0$ directed along field lines. For a horizontal field, the tension term resists bending of field lines in the vertical direction. As a parcel moves vertically it must work against this tension; moving horizontally it need not. The result is that the blast wave expands preferentially along $\hat{x}$, with the degree of elongation increasing as β decreases.

2.4 Setting Up the Problem

2.4.1 Navigating to the Test Problem

```
cd $PLUTO_DIR/Test_Problems/MHD/Blast_Simple
cp definitions_01.hpp definitions.hpp
cp pluto_01.ini      pluto.ini
python3 $PLUTO_DIR/setup.py
```

2.4.2 The Setup Menu

After running the Python script, the *Setup problem* menu should show the following configuration. The two key changes compared to Chapter 1 are `PHYSICS = MHD` and `DIMENSIONS = 2`.

Option	Value	Comment
PHYSICS	MHD	Full MHD equation set
DIMENSIONS	2	Two-dimensional
GEOMETRY	CARTESIAN	Cartesian (x, y) grid
MAPPED_GRID	NO	Uniform spacing
BODY_FORCE	NO	
COOLING	NO	
RECONSTRUCTION	LINEAR	Second-order interpolation
TIME_STEPPING	RK2	Second-order Runge–Kutta
NTRACER	0	No passive scalars
PARTICLES	NO	
USER_DEF_PARAMETERS	2	P_IN and BETA

2.4.3 The MHD Physics Sub-menu

Selecting `PHYSICS = MHD` reveals MHD-specific compile-time options:

Option	Description
`EOS`	Equation of state (`IDEAL` = adiabatic gamma-law)
`ENTROPY_SWITCH`	Solve the entropy equation in smooth regions for improved robustness near rarefactions
`DIVB_CONTROL`	Method for $\nabla \cdot \mathbf{B}$ control. Use `CONSTRAINED_TRANSPORT` for best accuracy
`RESISTIVITY`	Ohmic resistivity (diffusion term in the induction equation)
`HALL_MHD`	Hall term (important for weakly ionized or small-scale plasma)
`THERMAL_CONDUCTION`	Heat conduction term in the energy equation
`VISCOSITY`	Viscous stress tensor
`ROTATING_FRAME`	Coriolis and centrifugal terms for a rotating reference frame

For this problem use `EOS = IDEAL`, `DIVB_CONTROL = CONSTRAINED_TRANSPORT`, and set all dissipative options to `NO`.

2.4.4 User-Defined Parameters

This problem uses only two runtime parameters:

Index	Name	Physical Meaning
0	`P_IN`	Pressure inside the over-pressurized circular region
1	`BETA`	Plasma beta of the ambient medium, $\beta = P_{\rm amb}/(\lvert\mathbf{B}\rvert^2/2)$

2.5 Specifying Initial Conditions in init.cpp

2.5.1 Overview

Initial conditions for the blast wave are set in the `Init()` function of `init.cpp`, exactly as in Chapter 1. Two important differences from the shock tube appear:

1. You must also assign the **magnetic field components** `v[BX1]`, `v[BX2]`, `v[BX3]`.
2. With constrained transport active, you must additionally initialize the **vector potential components** `v[AX1]`, `v[AX2]`, `v[AX3]`, from which gPLUTO derives the face-centred magnetic field at the very first step.

In MHD problems, runtime parameters are commonly accessed via the **global array** `g_inputParam[<index>]`. This is equivalent to `run_config.inputParam` used in Chapter 1; either syntax works.

2.5.2 The Init() Function

```
void Init(double *v, double x1, double x2, double x3,
          RunConfig &run_config)
{
  double *inputParam = run_config.inputParam;
  double r, B0;

  /* Distance from the origin (dimension-independent) */
  r = DIM_EXPAND(x1*x1, + x2*x2, + x3*x3);
  r = sqrt(r);

  /* --- Density and Volocity --- */
  v[RHO] = 1.0;
  v[VX1] = 0.0;
  v[VX2] = 0.0;
  v[VX3] = 0.0;

  /* --- Background magnetic field: uniform, horizontal --- */
  /* B0 is derived from the ambient plasma beta:
       beta = P_amb / (B0^2/2)  =>  B0 = sqrt(2*P_amb/beta)
     The ambient thermal pressure is P_amb = 1/gamma in code units.  */
  B0      = sqrt(2.0 * inputParam[P_IN] / inputParam[BETA]);
  v[BX1]  =  B0;
  v[BX2]  =  0.0;
  v[BX3]  =  0.0;

  /* --- Pressure --- */
  if (r <= 0.1) {
    /* Over-pressurized inner disk */
    v[PRS] = inputParam[P_IN] * inputParam[BETA]; /* Change pressure internally */
  } else {
    /* Ambient (low) pressure */
    v[PRS] = 1.0 / run_config.gamma;
  }

  /* --- Vector potential (required for constrained transport) --- */
  /* For a uniform field B0 x-hat: A = B0 * y * z-hat  =>  AX3 = B0 * x2  */
  v[AX1] =  0.0;
  v[AX2] =  0.0;
  v[AX3] = -v[BX2]*x1 + v[BX1]*x2;   /* = B0 * x2 */
}
```

2.5.3 Explanation of Key Lines

The `DIM_EXPAND` macro. The macro `DIM_EXPAND(a, +b, +c)` expands to `a` in 1D, `a+b` in 2D, and `a+b+c` in 3D. Using it to compute the radius $r = \sqrt{x_1^2 + x_2^2 + x_3^2}$ makes the function portable across dimensionalities without any `#ifdef` blocks.

Deriving B_0 from β. The ambient thermal pressure in code units is $P_{\rm amb} = 1/\gamma$. The field strength follows from:

$$B_0 = \sqrt{\frac{2\,P_{\rm amb}}{\beta}}.$$

Because `P_IN` here actually encodes a combination of the over-pressure and β, the precise formula shown keeps all physical scales consistent between the inner and outer regions.

The vector potential. For a uniform field $B_0\hat{x}$, we need $\nabla \times \mathbf{A} = B_0\hat{x}$. One solution is $A_z = B_0\, y$, i.e. `v[AX3] = B0 * x2`. The general expression `v[AX3] = -v[BX2]*x1 + v[BX1]*x2` reduces to this for our setup but also works for a tilted field. gPLUTO uses $\mathbf{A}$ only at $t = 0$ to

initialize the staggered face-centred field; thereafter CT advances **B** directly.

Boundary conditions. Zero-gradient (`outflow`) boundary conditions are applied on all four sides of the 2D domain. This simply copies the last interior cell into the ghost cells, imposing a zero normal derivative. It is appropriate here because we do not expect the blast wave to reach the domain boundary.

In 2D you must specify all four boundaries

The `[Boundary]` block in `pluto.ini` must list `X1-beg`, `X1-end`, `X2-beg`, and `X2-end`. Forgetting one will cause a runtime error.

2.6 Runtime Parameters: pluto.ini

The complete `pluto.ini` for this problem is:

```
[Grid]
X1-grid    -0.5   192   0.5
X2-grid    -0.5   192   0.5
X3-grid     0.0     1   0.5

[Time]
CFL          0.4
CFL_max_var  1.1
tstop        1.0
first_dt     1.e-6

[Solver]
Solver       hll

[Boundary]
X1-beg       outflow
X1-end       outflow
X2-beg       outflow
X2-end       outflow
X3-beg       outflow
X3-end       outflow

[Static Grid Output]
uservar      0
dbl.h5      -2.5e-3  -1  single_file
flt         -1.0     -1  single_file
dbl         -2.5e-3  -1  single_file
tab          5.0e-3  -1
ppm         -1.0     -1
png         -1.0     -1
log           1
analysis    -1.0     -1

[Chombo HDF5 output]
Checkpoint_interval  -1.0   0
Plot_interval         1.0   0

[Parameters]
P_IN         1.e2
BETA         0.1
```

2.6.1 What Is New Compared to Chapter 1?

A 2D grid. Both `X1-grid` and `X2-grid` are now specified with 192 cells each over $[-0.5, 0.5]$. The total cell count is $192 \times 192 = 36\,864$ — roughly two orders of magnitude more work than the 1D shock tube, so this run will take a few minutes.

Lower CFL number. The CFL is set to 0.4 (versus 0.8 in Chapter 1). MHD problems are often more stiff because fast magnetosonic waves can travel much faster than the sound speed, requiring a smaller time step.

Multiple output formats.

- `dbl.h5` — double-precision **HDF5** files; the best format for loading with PyPLUTO or VisIt and for long-term archival.
- `dbl` — legacy double-precision binary; useful as a fallback.
- `tab` — ASCII column files; usable with Gnuplot but large for 2D.
- The negative numbers (e.g. `-2.5e-3`) mean "write every $|\Delta t| = 2.5 \times 10^{-3}$ simulation time units"; a positive integer means "every n time steps"; `-1` disables the format.

The Chombo HDF5 block. This optional block enables output compatible with the Chombo AMR framework. For standard runs it can be ignored; setting both intervals to `-1.0` disables it.

Problem parameters. `P_IN = 100` produces a strong explosion; `BETA = 0.1` sets the ambient field in the strongly magnetized regime.

Output file size

At 192×192 with many output times the `tab` files can grow large. For 2D runs it is generally better to rely on binary (`dbl`) or HDF5 (`dbl.h5`) output and enable `tab` only for quick diagnostic checks.

2.7 Compiling and Running

Compilation is unchanged from Chapter 1:

```
make -j
```

2.7.1 Serial Run

```
./pluto                    # Uses linux.g++.defs
```

2.7.2 Parallel Run with MPI

At 192×192 the serial run may take several minutes on a laptop. If you selected `linux.mpicxx.defs` as your makefile, you can parallelise the run with MPI:

```
mpirun -np <n> ./pluto     # Replace <n> with the number of MPI processes
```

gPLUTO automatically decomposes the domain into n strips along the x_1 direction. Each MPI process handles one strip and exchanges ghost-cell data with its neighbours after each update. The speedup is roughly proportional to n for well-balanced decompositions up to the number of physical cores on your machine.

Choosing the number of processes

A good starting point is one process per physical CPU core. For a 192×192 grid, 4–8 processes reduces the runtime to seconds. Oversubscribing (more processes than cores)

typically slows things down due to communication overhead and context switching.

2.8 Visualizing the Results

2.8.1 Gnuplot: 2D Colour Maps and Contours

For 2D problems, Gnuplot's `splot` command replaces the 1D `plot`. The `.tab` ASCII files now have three leading columns: x_1, x_2, and a blank separator column, followed by the data columns.

Producing a Colour Map

```
gnuplot> load "pluto.gp"              # Load variable name definitions
gnuplot> set pm3d map                 # Project onto the 2D plane
gnuplot> splot "data.0010.tab" u 1:2:(column(rho))
```

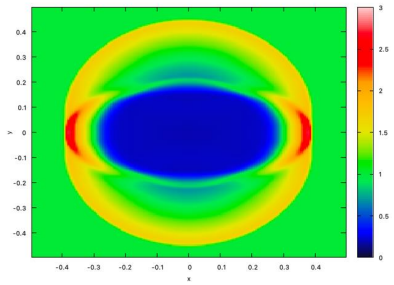


Figure 2.1: MHD Blast Wave: density color map

The `pm3d map` mode fills the 2D plane with colour according to the third argument — here the density. The `column(rho)` syntax uses the variable name defined in `pluto.gp`.

Drawing Contour Lines

```
gnuplot> load "pluto.gp"
gnuplot> set contour base        # Place contours on the base (z=0) plane
gnuplot> set view map            # View from directly above
gnuplot> unset surface           # Hide the 3D surface; show contours only
gnuplot> set style data lines
gnuplot> splot "data.0010.tab" u 1:2:(column(prs))
```

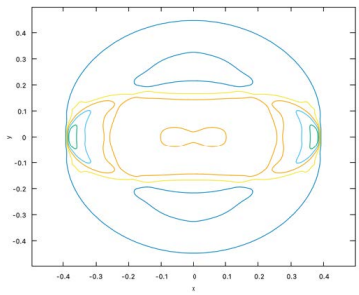


Figure 2.2: MHD Problem output countour lines

Animation

```
export GNUPLOT_LIB=$PLUTO_DIR/Tools/Gnuplot
```

```
gnuplot> load "mhd_blast.gp"
```

The pre-made script `mhd_blast.gp` loops over all available output files and renders an animated sequence of the density or pressure field.

2.8.2 PyPLUTO: 2D Colour Maps and Contours

PyPLUTO's `Image` class provides the `display()` method for pseudocolour maps and the `contour()` method for contour lines. These are the PyPLUTO equivalents of Gnuplot's `splot` modes.

Producing a Colour Map

```
import pyPLUTO as pp

D = pp.Load()       # Load the last available output
pp.Image().display(D.prs, x1=D.x1, x2=D.x2, cpos='right')
pp.show()
```

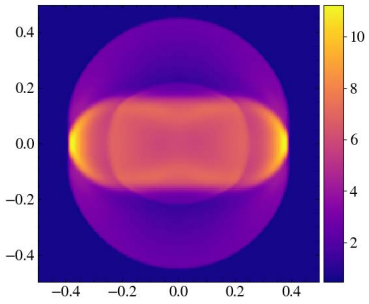


Figure 2.3: MHD Blast Wave: pressure color map

`D.prs` is a 2D NumPy array of shape `(nx1, nx2)`. The `cpos='right'` argument places the colour bar to the right of the image. Replace `D.prs` with `D.rho`, `D.bx1`, etc. to plot other variables.

Drawing Contour Lines

```
import pyPLUTO as pp

D = pp.Load()
pp.Image().contour(D.prs, x1=D.x1, x2=D.x2, cmap='plasma', levels=10)
pp.show()
```

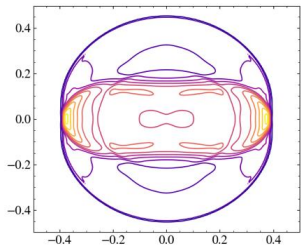


Figure 2.4: MHD Problem output countour lines PyPLUTO

The `levels=10` argument draws 10 automatically-spaced contour lines. You can pass an explicit list instead, e.g. `levels=[1, 5, 10, 20, 50]`.

Animation

```
python BW_video.py
```

The supplied script `BW_video.py` iterates over all output files, renders each frame with PyPLUTO, and saves the result as a video file.

Combining display() and contour()

Call both methods on the same `Image` object before `pp.show()` to overlay contour lines on a pseudocolour map — a standard way to highlight shock positions while preserving the overall morphology.

2.9 Parameter Study: The Role of Plasma Beta

2.9.1 The Experiment

This problem is ideal for a quick parameter study. All you need to do is change `BETA` in `pluto.ini` and rerun — no recompilation required. Run the simulation three times:

Run	BETA	Physical regime
A	100	Weakly magnetized: $P \gg P_B$
B	1	Equipartition: $P \approx P_B$
C	0.01	Strongly magnetized: $P \ll P_B$

Keep `P_IN = 100`, the 192×192 grid, and the same `tstop`.

2.9.2 Results and Physical Interpretation

Run A: $\beta = 100$ (Weakly Magnetized)

The magnetic field is dynamically negligible. The blast wave expands nearly isotropically, producing a near-circular pressure ring. This run is very close to what a purely hydrodynamic simulation would produce.

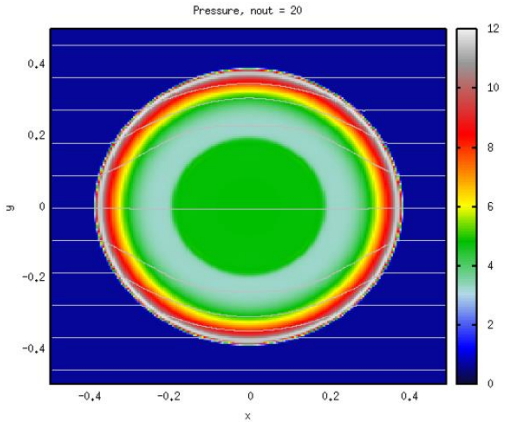


Figure 2.5: MHD parametric: Run A

Run B: $\beta = 1$ (Equipartition)

With comparable thermal and magnetic pressures, the field starts to shape the expansion. The outer shock is noticeably wider along $\hat{x}$ than along $\hat{y}$, forming a slightly elliptical ring.

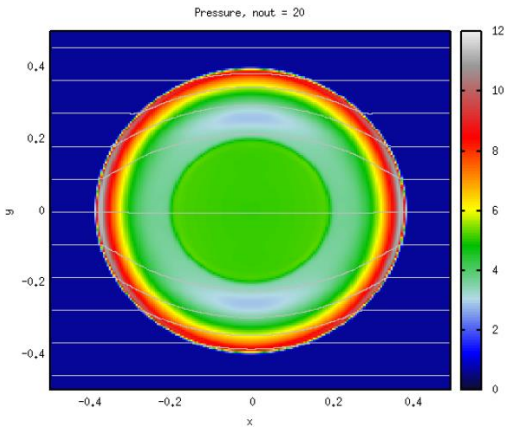


Figure 2.6: MHD parametric: Run B

Run C: $\beta = 0.01$ (Strongly Magnetized)

The blast wave is strongly confined perpendicular to **B** and expands almost exclusively along the field lines. The pressure map shows a highly elongated, cigar-shaped structure. The magnetic pressure in the compressed shell exceeds the thermal pressure by a large factor.

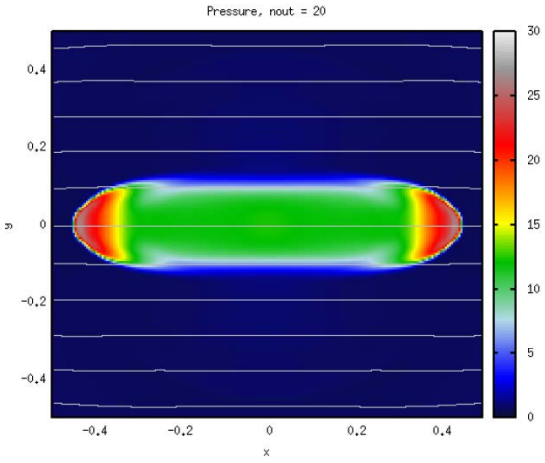


Figure 2.7: MHD parametric: Run C

The physical explanation is clear from the MHD force balance. A fluid element trying to move in the $\hat{y}$ direction (perpendicular to $\hat{x}$) must compress the field lines, which costs magnetic energy and exerts a restoring tension force. Motion along $\hat{x}$ encounters no such resistance. The lower β, the stronger this anisotropy.

2.9.3 Summary of Results

β	Regime	Blast shape at $t = t_{\text{stop}}$	Dominant mechanism
100	Weak B	Nearly circular	Thermal pressure
1	Equipart.	Slightly elliptical	Thermal + magnetic
0.01	Strong B	Strongly elongated along $\hat{x}$	Magnetic pressure + tension

2.10 Exercises

1. **Reproduce the parameter study.** Run the three cases $\beta = 100$, 1, 0.01 and produce side-by-side pressure maps at the same simulation time. Confirm that the blast wave elongates as β decreases.
2. **Increase the resolution.** Double the grid to 384×384 (edit `X1-grid` and `X2-grid` in `pluto.ini`). How does the shock structure change? Are there finer features visible?
3. **Rotate the field.** Modify `init.cpp` to set the magnetic field in the y-direction (`v[BX2] = B0`, `v[BX1] = 0`) and update the vector potential accordingly (`v[AX3] = -B0*x1`). What happens to the blast shape?
4. **Try a different Riemann solver.** Change `Solver` in `pluto.ini` from `hll` to `hlld` (the MHD-optimized four-wave solver). Compare the sharpness of the contact discontinuity and the outer shock with the HLL result.
5. **Test MPI scaling.** If MPI is available, run with 1, 2, and 4 processes and record the elapsed time reported at the end of each run. Is the speedup close to linear?
6. **Measure the shock anisotropy.** Using two output files, estimate the radial velocity of the outer shock along $\hat{x}$ and along $\hat{y}$ for $\beta = 1$. Compare the ratio to the ratio of the fast magnetosonic speeds $v_{f,x}/v_{f,y}$ expected analytically.

2.11 Summary

This chapter introduced the MHD module and 2D simulations in gPLUTO through the blast wave problem. The key takeaways are:

- **MHD extends HD** by adding the induction equation and the magnetic field as a dynamical variable. The total pressure becomes $P_{\rm tot} = P + |\mathbf{B}|^2/2$.
- **Constrained transport** (CT) maintains $\nabla \cdot \mathbf{B} = 0$ to machine precision. In `init.cpp`, you must initialize the vector potential `v[AX1]`, `v[AX2]`, `v[AX3]` alongside the field components `v[BX1]`, `v[BX2]`, `v[BX3]`.
- **The plasma beta** $\beta = P/(|\mathbf{B}|^2/2)$ is the main control parameter. Decreasing β makes the blast wave increasingly elongated along the field direction.
- **2D Gnuplot visualization** uses `splot` with `set pm3d map` for colour maps and `set contour` for contour lines.
- **PyPLUTO 2D visualization** uses `Image().display()` for colour maps and `Image().contour()` for contour lines.
- **MPI parallelism** requires selecting the MPI makefile and launching with `mpirun -np <n>`; gPLUTO handles domain decomposition automatically.

What comes next?

Chapter 3 covers Session #3: the **Kelvin–Helmholtz Instability (KHI)**. This classical shear-flow instability arises at the interface between two fluid layers moving at different velocities and provides an excellent testbed for studying perturbation growth, the transition to turbulence, and the stabilizing or destabilizing role of magnetic fields.

Chapter 3

Kelvin–Helmholtz Instability

3.1 Introduction

The **Kelvin–Helmholtz Instability (KHI)** is one of the most ubiquitous and visually striking phenomena in fluid dynamics. It develops at the interface between two fluids moving at different velocities — a shear layer — and can be triggered by arbitrarily small perturbations in the incompressible limit. As it grows, it rolls the interface into a characteristic sequence of curling vortices before eventually saturating and producing turbulent mixing.

The KHI is not merely a textbook curiosity. It plays an active role in a remarkable range of physical environments: it shapes the billow clouds visible on Jupiter's bands; it drives mixing at the boundary between the solar wind and the Earth's magnetosphere; it governs the entrainment of gas into astrophysical jets; and it controls turbulence at fluid interfaces in laboratory and geophysical flows.

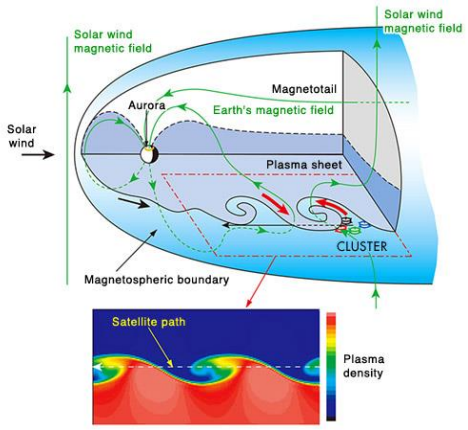


Figure 3.1: Example of Kelvin-Helmoz instability

Learning Objectives

By the end of this chapter you will be able to:

- Derive the linear dispersion relation for the KHI in both HD and MHD, and understand its dependence on the Mach number and the magnetic field geometry
- Use the `InitDomain()` function to assign initial conditions by looping over the computational domain
- Employ a passive scalar tracer (`NTRACER = 1`) to follow fluid mixing
- Use the `Analysis()` function for on-the-fly diagnostics and measure the numerical growth rate

- Produce interactive animations with PyPLUTO and compare numerical results against linear theory
- Understand how a parallel magnetic field suppresses the instability while a perpendicular field leaves it qualitatively unchanged

3.2 Physics Background

3.2.1 Governing Equations and Initial Configuration

The instability is analyzed using the **compressible Euler equations** (or their MHD generalization):

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho\, \mathbf{u}) = 0 \tag{3.1}$$

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u}\right) + \nabla p = 0 \tag{3.2}$$

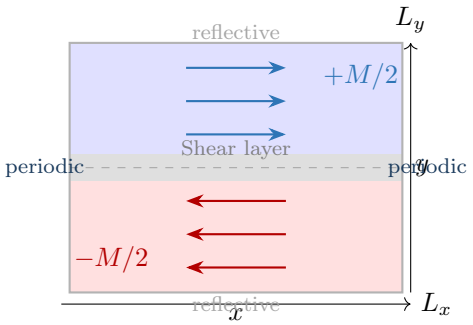
$$\frac{\partial p}{\partial t} + \mathbf{u} \cdot \nabla p + \gamma\, p \nabla \cdot \mathbf{u} = 0 \tag{3.3}$$

The background state is uniform in density ($\rho = 1$) and pressure ($c_s = 1$, so $p_0 = \rho_0 c_s^2/\gamma = 1/\gamma$). The velocity field consists of a horizontal flow that changes sign across $y = 0$, with the transition smoothed over a layer of half-width a:

$$\boxed{u_x(y) = \frac{M}{2} \tanh\left(\frac{y}{a}\right) \hat{e}_x} \tag{3.4}$$

Here $M = \Delta v / c_s$ is the **Mach number** of the shear, defined as the velocity jump across the layer divided by the sound speed. The parameter a controls how sharp the transition is: the limit $a \to 0$ corresponds to a vortex sheet (discontinuous velocity), while finite a corresponds to a smooth shear layer.

The domain is a 2D Cartesian box of dimensions $L_x \times L_y$, as shown below. Periodic boundary conditions are applied in the x-direction (so the flow is repeated without boundary effects in the horizontal), and **reflective** boundary conditions are applied at the top and bottom ($y = \pm L_y/2$).



3.2.2 Linear Theory: the Vortex-Sheet Case ($a \to 0$)

In the sharp-interface limit $a \to 0$, the velocity profile becomes a **vortex sheet**: a piecewise constant flow with a jump at $y = 0$. Linearizing about this background, we look for perturbations of the form $q'(\mathbf{x}, t) = \tilde{q}(y)\, e^{i(kx - \omega t)}$, where a complex $\omega(k)$ signals an instability.

The linearization process yields the analytical **dispersion relation**:

$$\frac{\omega}{k\,c_s} = \frac{M}{2} \pm \frac{i}{2}\left[\sqrt{M^2+1-\left(\frac{M^2}{4}+1\right)^{1/2}}\right] \tag{3.5}$$

The imaginary part of ω gives the **growth rate** of the instability. Two key features emerge from this relation:

- The instability exists for *all* Mach numbers when $a = 0$ (pure vortex sheet in a compressible fluid). The growth rate is positive for $M < \sqrt{8} \approx 2.83$ and the flow becomes stable above this supersonic cut-off.
- At low Mach number ($M \ll 1$), the growth rate is $\mathrm{Im}(\omega) \approx kMc_s/2$, increasing linearly with both wavenumber and Mach number.

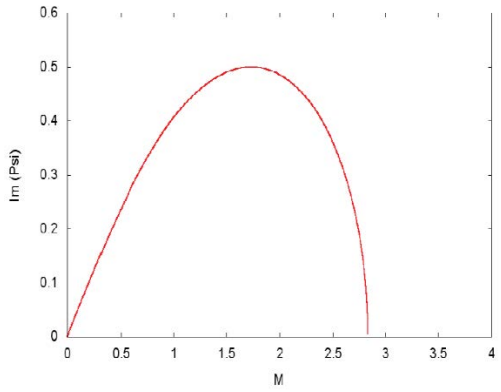


Figure 3.2: Analitical Dispersion for a=0

3.2.3 Linear Theory: the Finite Shear Layer ($a \neq 0$)

When the shear layer has finite width $a > 0$, the dispersion relation cannot be obtained analytically and the eigenvalue problem must be solved numerically. For the profile $u_x(y) = (M/2)\tanh(y/a)$, the instability region in the (k, M) plane (defined by $-\mathrm{Im}(\omega a/c_s) > 0$) exhibits several important quantitative features:

- **Peak growth rate.** At $M = 1$ and $B = 0$, the maximum growth rate is

$$\mathrm{Im}(\omega)_{\max} \approx 0.071\,\frac{c_s}{a}, \quad \text{achieved at} \quad ka \approx 0.396.$$

  This is the reference value used to validate numerical simulations.
- **Short-wavelength cut-off.** Perturbations with $ka \gtrsim 1$ are stable: the finite shear width suppresses very small-scale modes that the vortex sheet allows.
- **Supersonic stabilization.** Flows with $M > 1$ tend to become stable as compressibility increasingly damps the instability.

The role of finite shear width

In the vortex-sheet limit, all wavenumbers are unstable (up to the supersonic cut-off). A finite shear width a introduces a physical length scale into the problem. Perturbations with wavelength $\lambda \ll a$ "fit entirely inside" the shear layer and feel neither the upper nor the lower bulk flow, so they are not driven by the velocity difference — hence the short-wavelength cut-off at $ka \approx 1$.

3.2.4 Linear Theory: Effect of a Magnetic Field

When a uniform magnetic field **B** is included in the vortex-sheet problem, the linearization leads to a 10th-degree polynomial dispersion relation in ω. The impact of the field on the KHI depends critically on its **orientation** relative to the flow direction. Defining θ as the angle between **B** and $\hat{x}$:

Case $\theta = \pi/2$ (field perpendicular to flow). The magnetic field lies in the y-direction, perpendicular to the flow. In this geometry, the field does not inhibit the horizontal shear motion and has a relatively minor quantitative effect: the instability features are essentially unchanged if the Mach number is redefined as

$$M_{\rm eff} = \frac{v}{\sqrt{v_A^2 + c_s^2}},$$

where $v_A = B_0/\sqrt{\rho}$ is the Alfvén speed. The instability is still present and the vortex roll-up proceeds in the same way.

Case $\theta = 0$ (field parallel to flow). The magnetic field lies along $\hat{x}$, parallel to the bulk velocity. In this case the field has a strong **stabilizing** effect. Field lines that are advected along with the flow resist bending by the perturbation (magnetic tension opposes transverse displacement). The key results are:

- For $M < 2\,v_A/c_s$, the flow is **completely stable**; magnetic tension suppresses all perturbations.
- The supersonic stability cut-off decreases from $M = \sqrt{8}$ in the pure HD case to $M = 2$ in the MHD case.
- For $v_A/c_s > 1$ (strongly magnetized), the two stability limits coincide and the flow is always stable against the KHI.

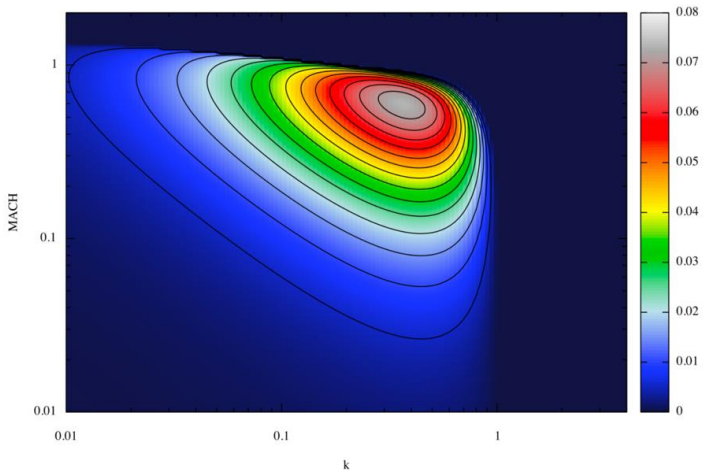


Figure 3.3: KHI: Magnetic field effect

Why parallel fields stabilize and perpendicular fields do not

Consider a field line parallel to the flow ($\theta = 0$). When a KHI perturbation displaces a fluid parcel transversely (y-direction), it must bend the frozen-in field line. The resulting magnetic tension acts as a restoring force, opposing the displacement — just like a stretched rubber band. A sufficiently strong field prevents the displacement from growing. For a perpendicular field ($\theta = \pi/2$), the field lines run vertically. A horizontal perturbation simply advects them horizontally without bending them, so there is no restoring tension and the instability is not inhibited.

3.2.5 Nonlinear Evolution

Once perturbations grow beyond the linear regime, the KHI enters a rich nonlinear phase characterized by several successive stages:

1. **Linear growth.** Small-amplitude perturbations grow exponentially at the rate predicted by linear theory. The interface is slightly deformed but retains its basic horizontal structure.
2. **Roll-up.** The interface folds over itself due to the vorticity concentrated in the shear layer. Each wavelength of the initial perturbation rolls up into a distinct vortex ("cat's eye" structure).
3. **Vortex merging.** Neighbouring vortices of the same sign attract each other and merge, forming progressively larger structures. Each merger dissipates kinetic energy of ordered flow into disordered (turbulent) motions and enhances mixing of fluid from the two regions.
4. **Turbulent mixing.** In the fully nonlinear regime, the two originally distinct fluids become thoroughly mixed and the kinetic energy of the shear flow is redistributed into a broad spectrum of eddies.

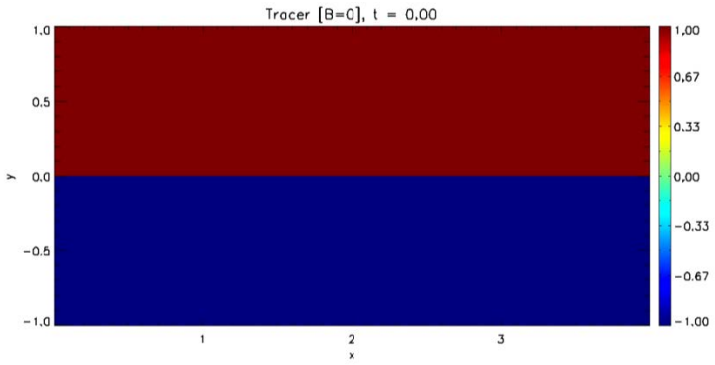


Figure 3.4: KHI: initial tracer distribution

The passive scalar tracer (enabled via `NTRACER = 1`) provides the clearest visualization of this process: it is initialized to a different value on each side of the interface (+1 above, −1 below) and can only change by advection and mixing, making the roll-up and merging stages immediately visible in a colour map.

3.3 Setting Up the Problem

3.3.1 Navigating to the Test Problem

```
cd $PLUTO_DIR/Test_Problems/MHD/Kelvin_Helmholtz
python3 $PLUTO_DIR/setup.py
```

Note that this problem does not require you to copy a pre-made `definitions.hpp` first — the one already in the directory is the correct starting point. Just run the script.

3.3.2 The Setup Menu

The *Setup problem* screen shows the following configuration. Compare it carefully to the blast wave (Chapter 2): the physics module is still `MHD`, but now we have `NTRACER = 1` (a passive scalar tracer is advected through the domain) and `USER_DEF_PARAMETERS = 5` (five independent parameters control the problem).

Option	Value	Comment
PHYSICS	MHD	MHD equation set
DIMENSIONS	2	Two-dimensional
GEOMETRY	CARTESIAN	Cartesian (x, y)
BODY_FORCE	NO	
COOLING	NO	
RECONSTRUCTION	LINEAR	Second-order
TIME_STEPPING	RK2	Second-order Runge–Kutta
NTRACER	1	One passive scalar tracer
USER_DEF_PARAMETERS	5	Five runtime parameters

What is a passive scalar tracer?

A **tracer** (`NTRACER = 1`) is an additional scalar quantity tr_1 that is advected with the fluid velocity but has no effect on the dynamics (it is "passive"). It is initialized to mark one fluid region (e.g., $\mathrm{tr}_1 = +1$ above the interface) versus another ($\mathrm{tr}_1 = -1$ below). Because it is purely advected, its spatial distribution at later times directly shows where fluid elements from each side have been transported. It is the standard diagnostic for visualizing mixing.

3.3.3 User-Defined Parameters

This problem uses five runtime parameters, which you will find in the *User-defined Parameters* sub-menu and later set in `pluto.ini`:

Index	Name	Physical Meaning
0	AMPL	Perturbation amplitude — amplitude of the initial y-velocity perturbation that seeds the instability
1	MACH	Mach number of the shear flow, $M = (\Delta v)/c_s = v_{\mathrm{top}}/c_s$
2	WIDTH	Shear layer half-width a (see eq. (3.4))
3	VALF	Alfvén speed $v_A = B_0/\sqrt{\rho}$ of the background magnetic field. Set to 0 for a purely hydrodynamic run
4	THETA	Angle θ (in degrees) between the magnetic field $\mathbf{B}$ and the x-axis

These five parameters give you direct experimental control over all the physical ingredients discussed in the linear theory section: the shear strength (M), the layer thickness (a), and the field strength and orientation (v_A, θ).

3.4 Specifying Initial Conditions: the InitDomain() Function

3.4.1 Two Functions for Initial Conditions

In all previous chapters we used the `Init()` function to set initial conditions. gPLUTO provides a second, complementary function: `InitDomain()`. The function is used here because we want to loop explicitly over the domain to set the tracer and perturb the velocity field.

The difference between the two functions is summarized in the following table:

Feature	`Init()`	`InitDomain()`
Called once per	Cell	Entire domain
Receives	(x_1, x_2, x_3) coordinates of one cell	Pointer to the full `Data` structure
Best for	Pointwise, closed-form ICs	ICs requiring loops, neighbour information, or global reductions

For this problem we use `InitDomain()` to show how to write loops over the computational grid — a skill that becomes essential for more complex setups.

3.4.2 The InitDomain() Function

The full implementation for the KHI problem is:

```
void InitDomain(Data *d, Grid *grid)
{
  /*
   * Assign initial conditions by looping over the
   * entire computational domain.
   * Values set here overwrite those from Init().
   */

  RunConfig run_config = *(grid->run_config);
  double *inputParam   = run_config.inputParam;

  /* -- Read user-defined parameters -- */
  double M     = inputParam[MACH];   /* Mach number                    */
  double a     = inputParam[WIDTH];  /* Shear layer half-width          */
  double vA    = inputParam[VALF];   /* Alfven speed                    */
  double theta = inputParam[THETA]*CONST_PI/180.0; /* Angle [rad]      */
  double eps   = inputParam[AMPL];   /* Perturbation amplitude          */
  double M2    = M/2.0;

  /* -- Grid loop index bounds -- */
  int ibeg = grid->lbeg[IDIR],  iend = grid->lend[IDIR];
  int jbeg = grid->lbeg[JDIR],  jend = grid->lend[JDIR];
  int kbeg = grid->lbeg[KDIR],  kend = grid->lend[KDIR];

  /* -- Precompute box height for normalization -- */
  double Ly  = grid->xend[JDIR] - grid->xbeg[JDIR];
  double xC  = grid->xend[IDIR]; /* Box length in x              */

  /* -- Domain loop -- */
  for (int k = kbeg; k <= kend; k++) {
    for (int j = jbeg; j <= jend; j++) {
      double y = grid->xC[JDIR][j];     /* Cell-centre y coordinate  */
      for (int i = ibeg; i <= iend; i++) {
        double x = grid->xC[IDIR][i];   /* Cell-centre x coordinate  */
```

```
        /* --- Background velocity profile --- */
        d->Vc[VX1][k][j][i] = M2 * tanh(y/a);

        /* --- Seed perturbation in vy --- */
        /* Two sinusoidal modes with different wavelengths */
        d->Vc[VX2][k][j][i] = eps * (
             sin(2.0*CONST_PI*x/xC)       /* 1st mode (1 wavelength) */
           + sin(4.0*CONST_PI*x/xC) );    /* 2nd mode (2 wavelengths)*/

        /* --- Magnetic field (uniform, tilted at angle theta) --- */
        d->Vc[BX1][k][j][i] = vA * cos(theta);
        d->Vc[BX2][k][j][i] = vA * sin(theta);
        d->Vc[BX3][k][j][i] = 0.0;

        /* --- Vector potential for constrained transport --- */
        /* A_z such that curl(A) = B_x hat_x + B_y hat_y     */
        d->Vc[AX3][k][j][i] = -vA*sin(theta)*x + vA*cos(theta)*y;

        /* --- Tracer: marks upper (y>0) vs lower (y<0) fluid --- */
        d->Vc[TR1][k][j][i] = (y > 0.0) ? 1.0 : -1.0;
      }
    }
  }
}
```

3.4.3 Key Points

Accessing grid data. Inside `InitDomain()`, cell-centre coordinates are accessed as `grid->xC[IDIR][i]` (for x) and `grid->xC[JDIR][j]` (for y). The loop runs from `lbeg` to `lend` in each direction — these are the indices of the first and last interior cells in the local MPI subdomain.

Accessing variables. Primitive variables are stored in the 4D array `d->Vc[VAR][k][j][i]`, indexed by (variable, z-index, y-index, x-index). The variable names are the same as in `Init()`: `VX1`, `VX2`, `BX1`, `BX2`, `AX3`, etc.

Seeding the perturbation. The instability is triggered by a small sinusoidal perturbation in v_y. We superimpose two modes (1 and 2 wavelengths across the box) to break the initial left-right symmetry and produce a richer nonlinear evolution. The amplitude `eps` ($\approx 10^{-4}$ to 10^{-2}) should be small enough to ensure that the simulation starts in the linear regime.

The tracer field. The tracer `TR1` is initialized to $+1$ in the upper half ($y > 0$) and -1 in the lower half ($y < 0$). It is purely advected and provides the most intuitive visualization of the interface roll-up.

The vector potential. For a uniform field at angle θ ($B_x = v_A \cos\theta$, $B_y = v_A \sin\theta$), the vector potential component `AX3` $= A_z$ satisfying $\mathbf{B} = \nabla \times \mathbf{A}$ is:

$$A_z = -B_y\, x + B_x\, y = -v_A \sin\theta\; x + v_A \cos\theta\; y.$$

3.5 Runtime Parameters: pluto.ini

The complete `pluto.ini` file for this problem, with the default values for the HD reference run:

```
[Grid]
X1-grid    0.0   128   1.0
X2-grid   -1.0   256   1.0

[Time]
CFL          0.4
CFL_max_var  1.1
```

```
tstop        20.0
first_dt     1.e-6

[Solver]
Solver       roe

[Boundary]
X1-beg       periodic
X1-end       periodic
X2-beg       reflective
X2-end       reflective
X3-beg       outflow
X3-end       outflow

[Static Grid Output]
uservar      0
dbl          -1.0   -1   single_file
flt          -1.0   -1   single_file
tab          -1.0   -1
ppm          -1.0   -1
png          -1.0   -1
log           1
analysis     0.1    -1

[Parameters]
AMPL         1.e-4
MACH         1.0
WIDTH        0.02
VALF         0.0
THETA        0.0
```

3.5.1 Notable Features of This File

Asymmetric 2D grid. The domain is $[0,1] \times [-1,1]$: one unit wide in x and two units tall in y. The grid uses 128 cells in x and 256 cells in y, giving square cells with $\Delta x = \Delta y = 1/128$. This elongated domain ensures the shear layer at $y = 0$ is far from the reflective top and bottom boundaries.

Mixed boundary conditions. A crucial feature of this problem is the **periodic** boundary condition in x: the domain wraps around horizontally, so the KHI vortices are not influenced by lateral walls. The **reflective** condition at the top and bottom (`X2-beg` and `X2-end`) mirrors all quantities through the boundary, which effectively doubles the simulation domain and suppresses boundary-generated waves.

Long simulation time. The stop time is `tstop = 20.0` (in units of a/c_s). This is much longer than in Chapters 1 and 2 because the instability grows slowly at first and we want to observe the full nonlinear evolution including vortex merging.

The `analysis` output line. The line `analysis 0.1 -1` tells gPLUTO to call the `Analysis()` function every $\Delta t = 0.1$ simulation time units and append the result to a file named `khi.dat`. This is the mechanism for measuring the growth rate.

Comparing HD and MHD. The same `pluto.ini` is used for both runs:

- **HD reference run:** `VALF = 0.0` (no magnetic field)
- **MHD run:** `VALF = 0.3`, `THETA = 0.0` (parallel field with Alfvén speed $v_A = 0.3\,c_s$)

No recompilation is needed between runs — only editing these two lines.

Run time at full resolution

A 128×256 grid evolved to $t = 20$ takes several minutes in serial. Consider using `mpirun -np 4 ./pluto` to speed it up, or run a lower resolution test (64×128) first to verify the setup before committing to the full run.

3.6 The Analysis() Function: On-the-Fly Diagnostics

3.6.1 What the Analysis() Function Does

One of gPLUTO's most powerful features for quantitative analysis is the `Analysis()` **function** in `init.cpp`. Unlike the output files produced by the `[Static Grid Output]` block, which write full snapshots of the entire domain (potentially large), `Analysis()` is designed for *lightweight, high-cadence* diagnostics: it computes a small number of scalar quantities from the current state of the simulation and appends them as a single line to a user-named ASCII file.

This function is called by gPLUTO at every analysis interval specified in `pluto.ini` (`analysis 0.1 -1` means every $\Delta t = 0.1$). Because it writes only numbers rather than full 2D arrays, you can afford to call it very frequently — even every time step if needed — giving you a dense time-series of diagnostics without filling your disk.

3.6.2 Output Format: khi.dat

For the KHI problem, `Analysis()` produces the file `khi.dat` with four columns:

```
# [0]            [1]            [2]            [3]
# t              max-min        max(By^2)      pm
  0.000000e+00   9.975486e-07   0.000000e+00   0.000000e+00
  9.928043e-02   2.572954e-07   0.000000e+00   0.000000e+00
  1.992804e-01   1.941973e-07   0.000000e+00   0.000000e+00
  2.992804e-01   1.848784e-07   0.000000e+00   0.000000e+00
  ...
  6.992804e-01   2.698428e-07   0.000000e+00   0.000000e+00
  7.992804e-01   3.626834e-07   0.000000e+00   0.000000e+00
  8.992804e-01   4.847207e-07   0.000000e+00   0.000000e+00
```

The columns are:

Col.	Quantity	Physical interpretation
0	t	Simulation time
1	$\max(v_y) - \min(v_y)$	Amplitude of the transverse velocity perturbation across the domain. This quantity grows exponentially during the linear phase, allowing the numerical growth rate to be measured.
2	$\max(B_y^2)$	Maximum magnetic energy in the y-component of the field. This is zero for HD runs and grows during MHD runs as the instability stretches and amplifies field lines.
3	$p_m = \langle B^2/2 \rangle$	Volume-averaged magnetic pressure. Measures the global build-up of magnetic energy.

3.6.3 Measuring the Growth Rate

During the linear phase, column 1 (the transverse velocity amplitude Δv_y) grows as $\Delta v_y \propto e^{\omega_i t}$, where $\omega_i = \mathrm{Im}(\omega)$ is the growth rate. To measure it, plot column 1 on a **logarithmic** scale as a

function of time. During the linear phase you will see a straight line; the slope of this line is the numerical growth rate.

Compare your measured slope to the analytical prediction. For the default parameters ($M = 1$, $a = 0.02$):

$$\omega_i^{\text{theory}} \approx 0.071\,\frac{c_s}{a} = 0.071\,\frac{1}{0.02} = 3.55 \text{ (in units of } c_s/\text{length}).$$

The numerical growth rate from the simulation should agree to within a few percent for a well-resolved run.

Quick Gnuplot command for the growth rate

```
gnuplot> set logscale y
gnuplot> plot "khi.dat" u 1:2 w lp
```

The linear segment appears as a straight line on the log-scale plot.

3.7 Compiling and Running

Compilation is identical to previous chapters:

```
make -j
```

For a serial run:

```
./pluto
```

For a parallel run with MPI (recommended for full resolution):

```
mpirun -np <n> ./pluto
```

As the simulation runs, you will see the standard output log. Because `tstop = 20` and the time steps are small (set by the CFL condition on the Alfvén speed for the MHD case), expect several thousand time steps.

3.8 Visualization

3.8.1 Gnuplot: Animation with khi.gp

A pre-made Gnuplot animation script is included in the test problem directory. Set the library path and load the script:

```
export GNUPLOT_LIB=$PLUTO_DIR/Tools/Gnuplot
gnuplot> load "khi.gp"
```

This produces a frame sequence that animates the tracer field `TR1`, making the vortex roll-up and merging immediately visible as a time evolution.

3.8.2 PyPLUTO: Interactive Animation

PyPLUTO provides a particularly convenient workflow for the KHI because of its `interactive()` and `animate()` methods, which allow you to explore the full time-series without writing separate plotting code for each output.

Loading All Outputs and Creating an Interactive View

```
import pyPLUTO as pp
```

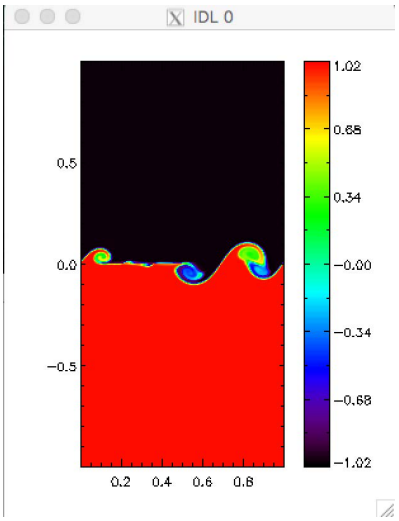


Figure 3.5: Simulation output plot with IDL

```
# Load all available outputs, keeping only the tracer variable
D = pp.Load('all', vars='tr1')

# Create an Image and launch interactive mode
I = pp.Image()
I.interactive(D.tr1, x1=D.x1, x2=D.x2,
              cpos='right', aspect='equal', cmap='RdBu')
```

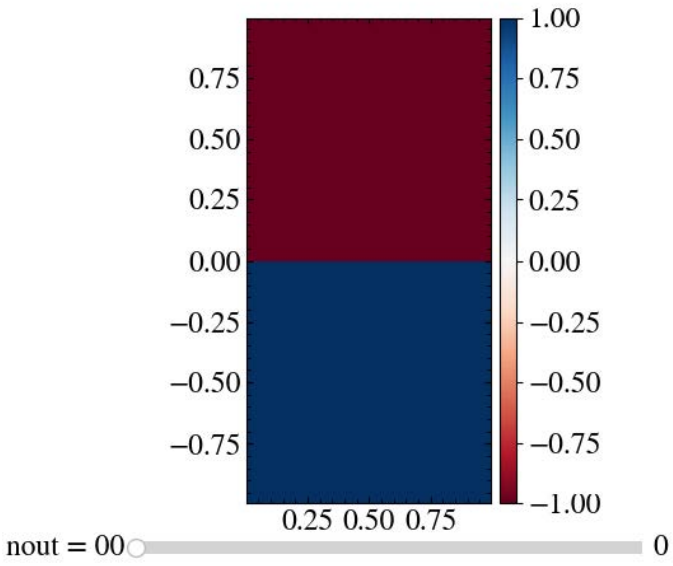


Figure 3.6: Simulation animation with pyPLUTO

The `pp.Load('all', vars='tr1')` call loads all output times at once, requesting only the tracer variable to save memory. The resulting object `D.tr1` is a 3D NumPy array of shape `(nout, nx1, nx2)`, where `nout` is the number of output snapshots.

The `interactive()` method creates a scrollable plot: you can step forward and backward through the output times using the keyboard or a slider, without having to write a loop yourself. The `cmap='RdBu'` colour map naturally shows the tracer: red ($\mathrm{tr}_1 = +1$) for the upper fluid and

blue ($\mathrm{tr}_1 = -1$) for the lower fluid, with intermediate colours (white/purple) where the two have mixed.

Creating an Animation File

```
# Create an animated GIF or MP4
I.animate('KHI.gif', interval=100)   # interval in ms between frames
```

Passing a filename to `animate()` saves the animation to disk as a GIF (or MP4 if `ffmpeg` is available). The `interval` parameter controls the display speed (100 ms per frame is a good starting point).

Loading and Plotting the khi.dat Diagnostics File

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

# Load the khi.dat file (skip=1 skips the header line)
res = pp.Load(None).read_file('khi.dat', skip=1)

# res is a dictionary; keys are column indices
t    = res[0]   # time
dvy  = res[1]   # max - min of vy
by2  = res[2]   # max(By^2)
pm   = res[3]   # mean magnetic pressure

# Plot the growth on a log scale
plt.figure()
plt.semilogy(t, dvy, 'k.-', label=r'$\Delta v_y$ (simulation)')

# Overlay the theoretical exponential growth
omega_theory = 0.071 / 0.02   # = 3.55 for our parameters
# Normalise to the simulation value at t=t_start
t_start = 5.0
i0 = np.argmin(np.abs(t - t_start))
A0 = dvy[i0]
plt.semilogy(t, A0 * np.exp(omega_theory * (t - t_start)),
             'r--', label=r'$\propto e^{\omega_i t}$ (theory)')

plt.xlabel('Time')
plt.ylabel(r'$\Delta v_y$')
plt.legend()
plt.title('KHI Growth Rate Comparison')
plt.tight_layout()
plt.savefig('khi_growth.pdf')
plt.show()
```

What pp.Load(None).read_file() returns

The `read_file()` method reads a plain ASCII column file and returns a Python dictionary where the key is the column index (0, 1, 2, ...) and the value is a NumPy array of that column's data. The `skip=1` argument skips the first line (the header comment).

3.9 Analysis and Exercises

3.9.1 Identifying the Phases of the KHI

After running the HD case (`VALF = 0`), produce:

1. A tracer colour map at several times (e.g., $t = 2,\ 5,\ 10,\ 20$) to visualize the roll-up and merging.
2. A log-scale plot of column 1 of `khi.dat` to identify the linear, saturation, and turbulent mixing phases.

   Can you identify the following phases?

- **Linear phase:** the exponential growth segment in the log-scale plot; the tracer map shows a barely-distorted interface.
- **Roll-up phase:** the tracer curls into discrete vortices and the growth rate begins to deviate from the analytical exponential.
- **Merging phase:** smaller vortices coalesce into larger ones; you may see two distinct merging events if the domain contains two initial perturbation wavelengths.
- **Turbulent mixing:** the interface is thoroughly disrupted and the tracer is distributed throughout the domain.

3.9.2 Measuring and Comparing the Growth Rate

From the log-scale plot of $\Delta v_y(t)$:

1. Identify the straight-line (linear) segment.
2. Estimate its slope $\omega_i^{\rm num}$ by fitting a line or reading off two points.
3. Compare to the theoretical value $\omega_i^{\rm theory} \approx 0.071\, c_s/a = 3.55$ for $M = 1$, $a = 0.02$.

3.9.3 Suppression by a Parallel Magnetic Field

Rerun with `VALF = 0.3`, `THETA = 0` (Alfvén speed $= 0.3 c_s$, field parallel to the flow). Since $M = 1$ and $2v_A/c_s = 0.6 < 1$, the flow is still unstable, but the field should slow the growth. Verify this by comparing the log-scale Δv_y plot to the HD case. Now try progressively stronger fields:

VALF	v_A/c_s	Expected behaviour
0.0	0	Unstable (HD reference)
0.3	0.3	Unstable but slower growth
0.5	0.5	Near the stability threshold ($2v_A/c_s = 1 = M$)
0.8	0.8	Stable ($2v_A/c_s = 1.6 > M = 1$)
1.2	1.2	Always stable ($v_A > c_s$)

Confirm that the flow becomes stable when $v_A \gtrsim M\, c_s/2 = 0.5$.

3.9.4 Effect of Field Geometry: Perpendicular Field

Change to `THETA = 90` (field perpendicular to the flow, $\mathbf{B} = v_A\, \hat{y}$). According to linear theory, this geometry does not suppress the instability. Verify that for `VALF = 0.8`, `THETA = 90` the KHI still grows, in stark contrast to the `THETA = 0` case with the same field strength.

3.9.5 Growth Rate vs. Resolution and Riemann Solver

The numerical growth rate depends on how well the code resolves the shear layer. Run the simulation at three resolutions and compare the measured growth rates:

$N_x \times N_y$	$\Delta x/a$	**Expected ω_i^{num}**
32×64	1.56	Underestimated (shear layer poorly resolved)
128×256	0.39	Close to theory
256×512	0.20	Very close to theory

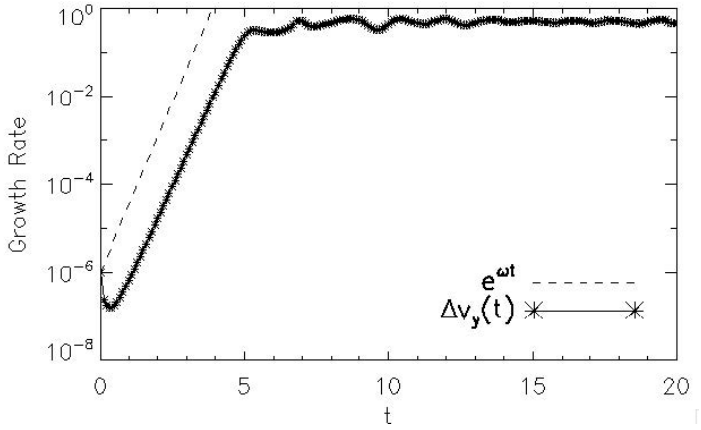


Figure 3.7: Growth Rate vs Resolution

Also try switching from the `roe` solver to the more diffusive `hll` solver (in `pluto.ini`). The HLL solver adds more numerical dissipation near discontinuities, which acts like an artificial viscosity on the shear layer and reduces the effective growth rate.

3.10 Summary

This chapter covered the Kelvin–Helmholtz Instability in both its theoretical and practical numerical aspects. The key takeaways are:

- **The KHI** develops at any shear-layer interface. In the vortex-sheet limit, the growth rate follows the analytical dispersion relation (3.5); for a finite shear layer of width a, the maximum growth rate is $\mathrm{Im}(\omega) \approx 0.071\, c_s/a$ at $ka \approx 0.396$ for $M = 1$.
- **Magnetic field geometry matters crucially.** A field parallel to the flow ($\theta = 0$) stabilizes the KHI for $v_A > Mc_s/2$ and completely suppresses it for $v_A > c_s$. A perpendicular field ($\theta = \pi/2$) leaves the instability qualitatively unchanged.
- `InitDomain()` allows you to set initial conditions by looping over the domain, accessing cell-centre coordinates and primitive variables through the `Data` and `Grid` structures.
- `NTRACER = 1` activates a passive scalar tracer that provides the most intuitive visualization of mixing and roll-up.
- **The `Analysis()` function** computes lightweight on-the-fly diagnostics (time series of scalar quantities) and appends them to an ASCII file (`khi.dat`) at user-specified intervals. This is the standard way to measure growth rates without writing full snapshots.
- **PyPLUTO's `interactive()` and `animate()` methods** provide easy time-series visualization; `read_file()` loads arbitrary ASCII data files into NumPy arrays for quantitative analysis.

What comes next?

Chapter 4 covers Session #4: the **Orszag–Tang Vortex**, a celebrated 2D MHD benchmark that tests the code's ability to handle the complex interaction between multiple MHD shocks and the emergence of large-scale magnetic reconnection. It introduces the concept of the

current sheet and provides a standard test for the robustness of the MHD solver in fully nonlinear, multi-wave scenarios.

Chapter 4

MHD Jet Propagation

4.1 Introduction

Collimated supersonic outflows — **jets** — are among the most spectacular and energetically significant phenomena in astrophysics. They emerge from accreting compact objects ranging from young stellar objects (YSOs) to active galactic nuclei (AGN), spanning twelve orders of magnitude in spatial scale and power. Despite this diversity, the underlying physics is governed by the same magnetohydrodynamic processes: the interplay between the jet's ram pressure, the ambient medium's inertia, and the magnetic field that both collimates the flow and mediates its stability.

This chapter introduces the numerical simulation of supersonic jet propagation in gPLUTO. It is the most complex problem in this tutorial series, requiring a non-trivial coordinate system (**cylindrical** geometry), a **custom boundary condition** function to continuously inject the jet, and consideration of both 2D axisymmetric and 3D Cartesian configurations.

Learning Objectives

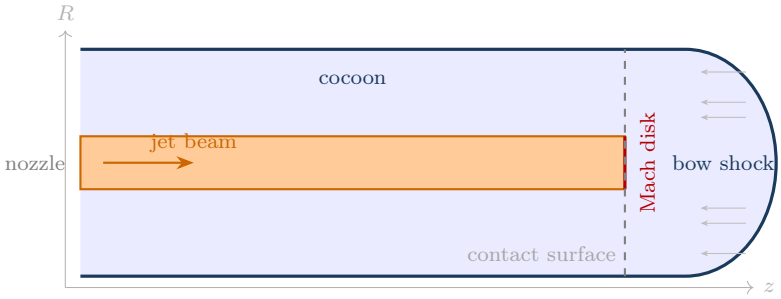
By the end of this chapter you will be able to:

- Set up a 2D axisymmetric MHD simulation in cylindrical coordinates using gPLUTO
- Understand the role of the magnetization parameters σ_z and σ_ϕ and the density contrast η in shaping jet morphology
- Implement a continuous supersonic injection boundary condition using `UserDefBoundary()` and the `GetJetValues()` helper function
- Interpret the standard features of a supersonic jet: bow shock, Mach disk, cocoon, and jet head
- Understand how varying η and σ_ϕ produces qualitatively different jet morphologies
- Extend the simulation to a full 3D Cartesian run to study the current-driven kink instability
- Use VisIt or ParaView for 3D volumetric rendering and slicing

4.2 Physics Background: Supersonic Jet Propagation

4.2.1 Anatomy of a Supersonic Jet

When a supersonic flow is injected into a denser ambient medium, a well-defined shock structure develops. The main features, illustrated schematically below, are:



- **Jet beam.** The supersonic flow injected through the nozzle. As long as the magnetic field is strong enough, the beam remains collimated over many jet radii.
- **Mach disk (terminal shock).** The jet is decelerated from supersonic to subsonic speed at the head. The Mach disk is a strong standing shock that converts kinetic energy into thermal energy.
- **Bow shock.** The ambient medium cannot adjust subsonically to the approaching jet. A forward bow shock precedes the jet head and pushes ambient material sideways into the cocoon.
- **Cocoon.** The region between the jet beam and the bow shock, filled with shocked jet and ambient material that has been displaced laterally. Backflow from the Mach disk inflates the cocoon.
- **Contact surface (jet head).** The interface between the post-Mach-disk jet material and the post-bow-shock ambient material. In the presence of density contrasts and velocity shear, this surface is subject to Rayleigh–Taylor and Kelvin–Helmholtz instabilities.

4.2.2 Key Dimensionless Parameters

The jet morphology and dynamics are controlled by a small set of dimensionless parameters. Quantities in the jet are subscripted j; quantities in the ambient medium are subscripted a.

Density contrast η.

$$\eta = \frac{\rho_j}{\rho_a}$$

This is the most important morphological parameter. A heavy jet ($\eta > 1$) is **ballistic**: ram pressure dominates and the beam propagates nearly unimpeded, drilling a long narrow channel. A light jet ($\eta < 1$) is **under-dense**: the ambient medium offers significant resistance, the jet is decelerated more efficiently, and the cocoon becomes broader and more turbulent.

Magnetic magnetization parameters. The magnetic field carried by the jet is described by two dimensionless magnetization parameters, defined as the ratio of magnetic pressure to ambient gas pressure:

$$\sigma_z = \frac{B_z^2}{2p_a}, \qquad \sigma_\phi = \frac{\langle B_\phi^2 \rangle}{2p_a}.$$

Here B_z is the axial (poloidal) field and B_ϕ is the toroidal (azimuthal) field. The angle brackets in σ_ϕ denote the appropriate spatial average over the jet cross-section. The total magnetization is $\sigma = \sigma_z + \sigma_\phi$.

The initial ambient field is characterized by:

$$\sigma_z = \frac{B_z^2}{2p_a}, \qquad B_z^2 = 2\sigma_z p_a.$$

Inside the jet, the toroidal field profile follows:

$$B_\phi(R) = \begin{cases} B_m\,(R/a) & R \le a \\ B_m\,(a/R) & R > a \end{cases} \qquad \text{where} \quad B_m^2 = \frac{2\sigma_\phi p_a}{a^2(1/2 - 2\log a)}.$$

This profile rises linearly for $R \leq a$ (solid-rotation-like) and falls off as R^{-1} outside, ensuring that the total toroidal flux is finite and the field is smooth at the axis.

Why toroidal fields matter

A toroidal (azimuthal) magnetic field B_ϕ in a jet produces a *pinch force*: the magnetic pressure and tension both act radially inward, squeezing the plasma. This **hoop stress** provides additional collimation beyond what the axial field can achieve and can dominate the jet dynamics at high σ_ϕ. However, a current-carrying jet with a dominant toroidal field is susceptible to the **current-driven kink instability** ($|m| = 1$ mode), which can disrupt the beam on a timescale of a few Alfvén crossing times. This is the motivation for the 3D extension of this problem.

Mach number. The jet Mach number $\mathcal{M}_j = u_j/c_{s,a}$ (velocity of the jet divided by the ambient sound speed) controls how strongly shocked the jet material is at the Mach disk and how fast the bow shock advances. In all cases here, $u_j/c_{s,a} = 20$, placing the jet well in the supersonic regime.

4.3 Physical Setup and Coordinate System

4.3.1 Cylindrical Coordinates

This problem is most naturally formulated in **cylindrical coordinates** (R, ϕ, z), where the jet propagates along the z-axis and the domain is axisymmetric about $R = 0$. In gPLUTO the mapping is:

$$x_1 = R, \quad x_2 = \phi, \quad x_3 = z.$$

The 2D axisymmetric run sets `DIMENSIONS = 3` but makes the ϕ-direction cyclic (one cell in x_2), effectively reducing the problem to 2D in the (R, z) plane. This is a standard trick for exploiting axisymmetry in gPLUTO without writing a separate 2D code.

4.3.2 Domain and Boundary Conditions

Boundary	Direction	Condition	Reason
$R = 0$ (X1-beg)	Radial inner	`axisymmetric`	Axis of symmetry
$R = R_{\max}$ (X1-end)	Radial outer	`outflow`	Free escape sideways
ϕ (X2-beg/end)	Azimuthal	`periodic`	Cyclic (1 cell in 2D)
$z = 0$ (X3-beg)	Jet inlet	`userdef`	Nozzle injection
$z = z_{\max}$ (X3-end)	Downstream	`outflow`	Free escape

The `axisymmetric` condition at $R = 0$ handles the coordinate singularity of cylindrical geometry: it reflects the vector components appropriately so that the axis remains smooth. The `userdef` condition at $z = 0$ is the nozzle: inside the jet radius $R \leq R_j = 1$ the supersonic jet values are injected; outside, ambient values are maintained.

4.3.3 Initial and Ambient Conditions

The entire domain is initially filled with the ambient medium:

$$\rho_a = 1, \quad p_a = \frac{1}{\Gamma}, \quad \mathbf{u}_a = \mathbf{0}.$$

A uniform vertical field $B_z = \sqrt{2\sigma_z p_a}$ threads the entire domain (jet + ambient) from the start. The jet material is injected continuously through the lower boundary, starting at $t = 0$.

4.4 Setting Up the Problem

4.4.1 Navigating to the Test Problem and Running Setup

```
cd $PLUTO_DIR/Test_Problems/MHD/Jet
cp definitions_01.hpp definitions.hpp
cp pluto_01.ini      pluto.ini
python3 $PLUTO_DIR/setup.py
```

The `definitions_01.hpp` / `pluto_01.ini` pair configures the **2D axisymmetric** run. The `definitions_10.hpp` / `pluto_10.ini` pair configures the **3D Cartesian** run for the kink instability study (see Section 4.9).

4.4.2 The Setup Menu

Option	Value	Comment
PHYSICS	MHD	Full MHD
DIMENSIONS	3	Three coordinates (R, ϕ, z); the ϕ direction is set cyclic in `pluto.ini` for the 2D run
GEOMETRY	CYLINDRICAL	Non-Cartesian: $x_1 = R$, $x_2 = \phi$, $x_3 = z$
MAPPED_GRID	NO	Uniform grid in each direction
BODY_FORCE	NO	
COOLING	NO	
RECONSTRUCTION	LINEAR	
TIME_STEPPING	RK2	
NTRACER	1	Tracer to distinguish jet from ambient
USER_DEF_PARAMETERS	4	η, u_j, σ_z, σ_ϕ

4.4.3 User-Defined Parameters

Index	Name	Physical Meaning
0	ETA	Density contrast $\eta = \rho_j/\rho_a$
1	JET_VEL	Jet velocity u_j (in units of $c_{s,a}$)
2	SIGMA_Z	Axial magnetization $\sigma_z = B_z^2/(2p_a)$
3	SIGMA_PHI	Toroidal magnetization $\sigma_\phi = \langle B_\phi^2 \rangle/(2p_a)$

4.5 Specifying Initial and Boundary Conditions

4.5.1 The Init() Function: Ambient Values

The `Init()` function sets the *ambient* (background) values that fill the domain at $t = 0$. It is deliberately simple because the interesting physics — the jet injection — is handled separately through the boundary condition:

```
void Init(double *v, double x1, double x2, double x3,
          RunConfig &run_config)
{
  /* Set ambient units and floor values */
  double pa = 1.0/run_config.gamma;   /* Ambient pressure p = 1/gamma */
  double cs = sqrt(run_config.gamma * pa);  /* Sound speed = 1          */

  /* -- Ambient conditions -- */
  v[RHO] = 1.0;           /* Uniform density                          */
  v[VX1] = 0.0;
  v[VX2] = 0.0;
  v[VX3] = 0.0;           /* Zero velocity everywhere                 */
  v[PRS] = pa;            /* Thermal pressure 1/gamma                 */

  /* -- Magnetic field: uniform vertical field -- */
  double *inputParam = run_config.inputParam;
  double sigma_z = inputParam[SIGMA_Z];
  double Bz      = sqrt(2.0 * sigma_z * pa);

  #if GEOMETRY == CYLINDRICAL
    v[BX3] = Bz;          /* Axial field (z-direction in cylindrical) */
    v[BX1] = 0.0;
    v[BX2] = 0.0;
  #elif GEOMETRY == CARTESIAN && (DIMENSIONS == 3)
    v[BX3] = Bz;
    v[BX1] = 0.0;
    v[BX2] = 0.0;
  #endif

  /* -- Vector potential (for CT initialization) -- */
  /* For B_z uniform: A_phi = (Bz/2)*R in cylindrical => AX2 = 0.5*Bz*x1 */
  v[AX1] = 0.0;
  v[AX2] = 0.5*Bz*x1;     /* A_phi such that curl A = Bz z-hat         */
  v[AX3] = 0.0;

  /* -- Tracer: 0 in ambient medium -- */
  v[TR1] = 0.0;
}
```

4.5.2 The Boundary Condition Architecture

The supersonic jet injection cannot be handled through the standard `Init()` function because it is a **continuous inflow boundary** rather than a one-time initial condition. gPLUTO implements this through two functions in `init.cpp`:

Function	Role
`GetJetValues()`	Helper function that computes and stores all jet primitive variables (density, velocity, pressure, magnetic field components) as a function of radius R. Called once to set up the jet profile.
`UserDefBoundary()`	The actual boundary condition function called by gPLUTO at every time step. It loops over ghost cells at $z = 0$ and fills them with jet values (if $R \leq R_j$) or ambient values (if $R > R_j$).

4.5.3 The GetJetValues() Function

```
void GetJetValues(double *vjet, double x1, double x2, double x3,
                  RunConfig &run_config)
{
  /* Compute the jet primitive variables at position (R=x1, phi=x2, z=x3) */

  double *inputParam = run_config.inputParam;
  double pa   = 1.0/run_config.gamma;
  double eta  = inputParam[ETA];
  double vj   = inputParam[JET_VEL] * sqrt(run_config.gamma * pa);
  double sz   = inputParam[SIGMA_Z];
  double sphi = inputParam[SIGMA_PHI];

  double Bz   = sqrt(2.0 * sz  * pa);
  double Rj   = 1.0;          /* Jet radius in code units          */
  double a    = 0.5 * Rj;     /* Toroidal field scale radius       */
  double R    = x1;           /* Cylindrical radius of ghost cell  */

  /* -- Toroidal field profile -- */
  double Bm2 = 2.0 * sphi * pa /
               (a*a * (0.5 - 2.0*log(a)));  /* Peak B_phi^2         */
  double Bphi;
  if (R <= a) {
    Bphi = sqrt(Bm2) * (R / a);             /* Linear rise to axis  */
  } else if (R <= Rj) {
    Bphi = sqrt(Bm2) * (a / R);             /* 1/R fall-off          */
  } else {
    Bphi = 0.0;                              /* Outside nozzle = 0   */
  }

  /* -- Jet primitive variables (inside nozzle) -- */
  vjet[RHO] = eta;          /* Jet density = eta * rho_a            */
  vjet[VX1] = 0.0;          /* No radial velocity                   */
  vjet[VX2] = 0.0;          /* No azimuthal velocity                */
  vjet[VX3] = vj;           /* Supersonic axial velocity            */
  vjet[PRS] = pa;           /* Pressure equilibrium with ambient    */
  vjet[BX3] = Bz;           /* Axial field (same as ambient)        */
  vjet[BX1] = 0.0;
  vjet[BX2] = Bphi;         /* Toroidal field component             */
  vjet[TR1] = 1.0;          /* Tracer = 1 marks jet material        */
}
```

Key Points in GetJetValues()

Pressure equilibrium. The jet pressure equals the ambient pressure ($p_j = p_a = 1/\Gamma$). This is the simplest choice and avoids launching a pressure wave at $t = 0$; the jet dynamics is driven entirely by ram pressure and magnetic forces.

The toroidal field profile. The B_ϕ profile is designed to be smooth at both the axis ($R = 0$) and the jet edge ($R = R_j = 1$). It rises linearly for $R \leq a$ and falls off as R^{-1} for $a < R \leq R_j$. Outside the nozzle ($R > R_j$) it is zero. This profile approximates a current-carrying wire of finite radius and naturally satisfies $\nabla \cdot \mathbf{B} = 0$ in cylindrical coordinates provided the corresponding vector potential is used for CT initialization.

Tracer value. The jet tracer is set to $\mathrm{tr}_1 = 1$, while the ambient was initialized to $\mathrm{tr}_1 = 0$ in `Init()`. Any intermediate value between 0 and 1 in the simulation indicates a mixture of jet and ambient material, providing a direct measure of mixing.

4.5.4 The UserDefBoundary() Function

```
void UserDefBoundary(const Data *d, RBox *box, int side,
                     Grid *grid)
{
  int   i, j, k, nv;
  RunConfig run_config = *(grid->run_config);
  double t = d->run_config->time;
  double vjet[NVAR], vout[NVAR];

  if (side == X3_BEG) {
    /* -- Apply boundary at z = 0 (lower z face) -- */

    /* Parallel loop over all ghost cells at z = 0 */
    #pragma acc loop gang
    BOX_LOOP(box, k, j, i) {
      double R = grid->xC[IDIR][i];  /* Cylindrical radius of this cell */

      /* Get jet values at this (R, phi) position */
      GetJetValues(vjet, grid->xC[IDIR][i],
                         grid->xC[JDIR][j],
                         grid->xC[KDIR][k],
                   run_config);

      if (R <= 1.0) {
        /* -- Inside nozzle: inject the jet -- */
        for (nv = 0; nv < NVAR; nv++) {
          d->Vc[nv][k][j][i] = vjet[nv];
        }
      } else {
        /* -- Outside nozzle: ambient outflow (copy from first interior cell) */
        for (nv = 0; nv < NVAR; nv++) {
          d->Vc[nv][k][j][i] = d->Vc[nv][k+1][j][i];
        }
      }
    }
  }
}
```

Key Points in UserDefBoundary()

The `side` parameter. `UserDefBoundary()` is called for every boundary that has `userdef` in `pluto.ini`. The `side` argument identifies which boundary is being processed: `X3_BEG` is the lower

z boundary (the nozzle). The function checks `if (side == X3_BEG)` before doing anything, so it only acts on the correct face.

The `BOX_LOOP` macro. This macro generates a triple nested loop over the ghost cells of the current boundary face. It automatically handles the correct index ranges for the specific boundary being processed.

Supersonic injection. When $R \leq R_j = 1$, the ghost cell is filled with all jet values from `GetJetValues()`. The velocity $v_z = u_j \gg c_s$ means information travels from the ghost cell into the domain — no characteristic extrapolation is needed because the flow is supersonic and information only propagates downstream.

Ambient outflow. When $R > R_j$, the ambient boundary is handled as a simple zero-gradient (outflow) condition: the first ghost cell copies the first interior cell. This prevents artificial reflections from the domain corner.

The `#pragma acc loop` directive. This OpenACC pragma enables GPU acceleration (when the code is compiled with the NVIDIA `nvc++` compiler and the `linux.nvc++.acc.defs` makefile). In standard CPU builds it is simply ignored.

4.6 Runtime Parameters: pluto.ini

The complete `pluto.ini` for the 2D axisymmetric run:

```
[Grid]
X1-grid    0.0   160   10.0
X2-grid    0.0     1    1.0
X3-grid    0.0   480   30.0

[Time]
CFL          0.4
CFL_max_var  1.1
tstop        1.8
first_dt     3.e-7

[Solver]
Solver       hllc

[Boundary]
X1-beg       axisymmetric
X1-end       outflow
X2-beg       periodic
X2-end       periodic
X3-beg       userdef
X3-end       outflow

[Static Grid Output]
uservar      0
dbl         -1.0   -1   single_file
flt         -1.0   -1   single_file
tab         -1.0   -1
ppm         -1.0   -1
png         -1.0   -1
log           1
analysis    -1.0   -1

[Parameters]
ETA          10.0
JET_VEL      20.0
SIGMA_Z       0.0
```

```
SIGMA_PHI     0.01
```

4.6.1 Notes on the Grid and Parameters

Grid dimensions. The radial domain extends from $R = 0$ to $R = 10$ (ten jet radii), with 160 cells giving $\Delta R = 0.0625\, R_j$. The axial domain extends from $z = 0$ to $z = 30$ (thirty jet radii), with 480 cells giving a similar axial resolution. The ϕ direction has a single cell (the cyclic direction for 2D runs).

Very small `first_dt`. The first time step is set to 3×10^{-7} — much smaller than in previous chapters. This is because the jet is injected supersonically at $t = 0$ into a stationary ambient, creating very large initial pressure gradients in the ghost cells. A tiny first step allows the code to "ramp up" stably.

The three parameter cases. All three workshop cases use $u_j = 20$ (Mach 20 jet) and $\sigma_z = 0$ (no axial field in the initial run). Only η and σ_ϕ change:

Case	Physical regime	ETA	SIGMA_PHI
1	Ballistic heavy jet	10.0	0.01
2	Under-dense, weakly magnetized	0.05	0.01
3	Under-dense, strongly magnetized	0.05	10.0

4.7 Compiling and Running

Compilation is the same as in previous chapters:

```
make -j
```

For a serial run:

```
./pluto
```

For a parallel MPI run:

```
mpirun -np <n> ./pluto
```

The 2D axisymmetric run at 160×480 resolution takes a few minutes in serial. Because the domain is elongated in z, decomposing it along z (the x_3 direction) with MPI gives a good load balance.

4.8 Comparing Jet Morphologies

After running all three cases and producing density maps ($\log_{10}\rho$), the following qualitatively distinct morphologies emerge.

4.8.1 Case 1: Ballistic Heavy Jet ($\eta = 10$, $\sigma_\phi = 0.01$)

A heavy jet has a density contrast $\eta = 10$: the jet material is ten times denser than the ambient. The ram pressure $\rho_j u_j^2 = 10 \times 20^2 = 4000$ far exceeds the ambient pressure $p_a = 1/\Gamma \approx 0.71$ (for $\Gamma = 1.4$). The jet is therefore almost unaffected by the ambient medium and propagates like a nearly rigid beam — a **ballistic jet**.

Key morphological features visible in the density map:

- A long, narrow beam extending nearly the full domain length.

- A bright, compact Mach disk at the jet head with a small working surface.
- A thin, elongated bow shock and cocoon, barely wider than the beam.
- Internal oblique shocks (recollimation shocks) visible as alternating bright and dark bands along the beam.

4.8.2 Case 2: Under-dense, Weakly Magnetized ($\eta = 0.05$, $\sigma_\phi = 0.01$)

An under-dense jet ($\eta = 0.05$, i.e. twenty times lighter than the ambient) has much lower ram pressure and is decelerated more efficiently. The magnetic field is weak ($\sigma_\phi = 0.01$) and plays a minor role.

Key differences from Case 1:

- The beam is shorter (slower advance speed) and **broader**, with a more developed cocoon.
- The Mach disk is broader and less stable; the jet head is more irregular.
- The cocoon is turbulent and fills a much larger volume.
- The bow shock is more rounded and slower.

This is the regime typical of **FR-II radio jets** in the extragalactic context, where the jet is lighter than the intracluster medium.

4.8.3 Case 3: Under-dense, Strongly Magnetized ($\eta = 0.05$, $\sigma_\phi = 10$)

With the same low density but very high toroidal magnetization ($\sigma_\phi = 10$), the magnetic hoop stress completely changes the morphology.

Key differences from Case 2:

- The beam is strongly **collimated** by the magnetic pinch force, remaining much narrower than the weakly magnetized case.
- The cocoon is very narrow and elongated — the toroidal field effectively collimates the backflow as well.
- The bow shock is much more elongated ("pencil"-shaped rather than round), because the jet advances faster along the axis.
- Recollimation shocks inside the beam are more pronounced due to the oscillating magnetic tension.

Summary of morphological trends

Case	η	σ_ϕ	Morphology
1	10	0.01	Narrow ballistic beam, thin cocoon, fast advance
2	0.05	0.01	Short turbulent beam, broad irregular cocoon
3	0.05	10.0	Narrow magnetically collimated beam, fast advance despite low density

The progression illustrates how the toroidal field can partially "substitute" for the ram pressure in providing beam collimation, enabling an under-dense jet to propagate much further than it would without magnetic support.

4.9 The 3D Jet and the Kink Instability

4.9.1 Motivation

The 2D axisymmetric setup assumes perfect rotational symmetry about the z-axis. This suppresses a whole class of perturbations with azimuthal mode number $|m| \geq 1$. In particular, the **current-driven kink instability** (the $|m| = 1$ mode, also called the kink or $m = 1$ mode) can only develop if the azimuthal symmetry is broken.

The kink mode is driven by the axial current density $j_z \propto \nabla \times \mathbf{B}_\phi$ associated with the toroidal field. When the Kruskal–Shafranov criterion is satisfied — roughly when the field lines wind more than once around the jet — the current-carrying configuration is unstable and the beam will undergo a lateral displacement that grows exponentially with time. This can ultimately disrupt the jet and is thought to be responsible for the observed bending and disruption of some astrophysical jets.

4.9.2 Setting Up the 3D Run

```
cp definitions_10.hpp definitions.hpp
cp pluto_10.ini      pluto.ini
python3 $PLUTO_DIR/setup.py
```

The 3D run uses **3D Cartesian coordinates** (x, y, z) rather than cylindrical. gPLUTO treats the jet as propagating along z, the nozzle is a circular region in the xy-plane. The key differences from the 2D setup are:

- `GEOMETRY = CARTESIAN` and `DIMENSIONS = 3`.
- The domain is smaller (fewer radial jet radii) to keep the computational cost tractable.
- `TIME_STEPPING = RK3` and `RECONSTRUCTION = PPM` (piecewise parabolic method) are used: third-order Runge–Kutta time integration and higher-order spatial reconstruction. This less dissipative combination resolves the kink mode better.
- The domain is deliberately set to favour the $|m| = 1$ mode by choosing a domain length comparable to the kink wavelength.

4.9.3 Recommended Makefile and HPC Options

For the 3D run, the serial makefile is too slow. Select one of:

- `linux.mpicxx.defs`: standard multi-core MPI run. Use `mpirun -np <n> ./pluto` with at least 4–8 cores.
- `linux.nvc++.acc.defs`: NVIDIA GPU-accelerated run using OpenACC. If a GPU is available, this typically gives a factor of 10–50 speedup over a single CPU core. Requires the NVIDIA HPC SDK (`nvc++`) compiler.

3D run time

A full 3D run at sufficient resolution to observe the kink instability can take tens of minutes to hours depending on hardware. If time is limited, run the simulation to an earlier stop time (`tstop = 5` or `10`) and look for the onset of the lateral displacement of the jet beam.

4.10 Visualization

4.10.1 2D Runs: PyPLUTO Density Maps

For the 2D axisymmetric runs, PyPLUTO colour maps of $\log_{10}(\rho)$ in the (R, z) plane are the standard visualization:

```
import pyPLUTO as pp
import numpy as np
```

```
# Load the last output
D = pp.Load()

# Log density colour map in the (R, z) plane
I = pp.Image()
I.display(np.log10(D.rho), x1=D.x3, x2=D.x1,
          cpos='right', cmap='inferno',
          label1='z', label2='R',
          title=r'$\log_{10}(\rho)$ at $t = $' + f'{D.t:.2f}')
pp.show()
```

Note that in cylindrical geometry, `D.x1` is the radial coordinate R and `D.x3` is the axial coordinate z, so the axes are swapped relative to the natural (R, z) plot.

To produce a mirrored plot (showing both the $R > 0$ and the reflected $R < 0$ side for a more intuitive cylindrical view), construct the reflected array explicitly:

```
import numpy as np
import pyPLUTO as pp

D = pp.Load()

# Stack the mirrored and original arrays
rho_full = np.vstack([D.rho[:, ::-1, :], D.rho])  # Reflect in R
R_full   = np.concatenate([-D.x1[::-1], D.x1])

I = pp.Image()
I.display(np.log10(rho_full.T), x1=D.x3, x2=R_full,
          cpos='right', cmap='inferno')
pp.show()
```

4.10.2 3D Runs: VisIt and ParaView

For the 3D run, gPLUTO can output data in formats compatible with **VisIt** and **ParaView**, two standard open-source scientific visualization packages. Enable HDF5 output in `pluto.ini`:

```
[Static Grid Output]
dbl.h5  -1.0  -1  single_file
```

Then load the `.h5` files into VisIt or ParaView. Useful operations for jet visualization include:

- **Pseudocolor slice.** Take a 2D slice through the xz or yz plane and colour it by $\log_{10}(\rho)$ or the tracer `tr1` to see the jet cross-section.
- **Volume rendering.** Assign an opacity transfer function to the tracer so that $\mathrm{tr}_1 \approx 1$ (jet material) is rendered opaque and $\mathrm{tr}_1 \approx 0$ (ambient) is transparent. This gives a 3D view of the jet beam and its distortions due to the kink instability.
- **Isosurface.** Extract an isosurface of $\mathrm{tr}_1 = 0.5$ to show the boundary between jet and ambient material. Kink-unstable jets show a clear lateral displacement of this surface.

Using the tracer as a mask

The tracer variable `tr1` provides the cleanest way to separate jet from ambient material in 3D visualizations. Because it is purely advected with the flow, regions with $\mathrm{tr}_1 \approx 1$ contain predominantly jet material, while $\mathrm{tr}_1 \approx 0$ indicates ambient. A threshold of $\mathrm{tr}_1 > 0.5$ is a standard choice for the "jet surface" isosurface.

4.11 Exercises

1. **Reproduce the three morphology cases.** Run all three parameter sets in the table (Cases 1–3) to $t = 1.8$ and produce side-by-side $\log_{10}\rho$ colour maps. Identify the bow shock, Mach disk, beam, and cocoon in each case. How do the advance speed of the bow shock and the cocoon width change across the three cases?
2. **Measure the jet head advance speed.** From two output snapshots, estimate the z-coordinate of the Mach disk (the head of the jet beam) and divide by the time interval to get the advance speed $v_{\rm head}$. Compare to the analytical estimate from ram-pressure balance:
$$\frac{v_{\rm head}}{u_j} \approx \frac{\sqrt{\eta}}{1+\sqrt{\eta}}.$$
Do all three cases agree with this formula? Why might Case 3 deviate from it?
3. **Vary the density contrast.** Run a fourth case with $\eta = 1$ (matched-density jet, $\sigma_\phi = 0.01$) and compare the morphology to Cases 1 and 2. At what value of η does the transition between ballistic and decelerated regimes occur?
4. **Add an axial field.** Set $\sigma_z = 1.0$ (strong axial field, $\sigma_\phi = 0$, $\eta = 0.05$). How does the axial field change the morphology compared to Case 2? Does it collimate the jet in the same way as the toroidal field?
5. **3D kink instability.** Run the 3D Cartesian configuration (`definitions_10.hpp`, `pluto_10.ini`). Load the output in VisIt or ParaView and produce a 3D volume rendering of the tracer. Identify the onset of the $|m| = 1$ kink mode by looking for lateral displacement of the jet beam. At what time does the kink become visible? Does it eventually disrupt the jet?
6. **Resolution study for the 2D case.** Double the radial and axial resolution (320×960 cells) for Case 2 and compare the density maps. Are the fine structures in the cocoon and at the jet head converged at the lower resolution?

4.12 Summary

This chapter introduced the simulation of supersonic MHD jet propagation, the most complex problem in this tutorial series. The key points are:

- **Cylindrical geometry** ($x_1 = R$, $x_2 = \phi$, $x_3 = z$) is the natural choice for an axisymmetric jet. A 2D axisymmetric run uses `DIMENSIONS = 3` with a single cyclic ϕ-cell.
- **Continuous injection** is implemented via `UserDefBoundary()`, which fills ghost cells at $z = 0$ with jet values (if $R \le R_j$) or ambient outflow conditions (if $R > R_j$) at every time step.
- `GetJetValues()` computes the radially-varying jet profile, including the toroidal field $B_\phi(R)$ that rises linearly to a peak and then falls off as R^{-1}.
- **The density contrast η and toroidal magnetization σ_ϕ** are the primary morphological parameters. A heavy jet ($\eta \gg 1$) is ballistic; a light jet ($\eta \ll 1$) is decelerated. A strong toroidal field ($\sigma_\phi \gg 1$) collimates even a light jet via magnetic hoop stress.
- **The 3D extension** uses Cartesian coordinates, PPM reconstruction, and RK3 time-stepping to study the current-driven kink instability — an intrinsically 3D phenomenon that cannot be captured in the axisymmetric approximation.
- **3D visualization** with VisIt or ParaView uses pseudocolour slices, volume rendering, and isosurfaces of the tracer variable to characterize the 3D jet structure.

What comes next?

With the four core sessions complete, you have a working knowledge of gPLUTO spanning 1D HD, 2D MHD, and 3D MHD problems in both Cartesian and cylindrical coordinates, with multiple boundary condition types, custom diagnostics, and parallel execution. Advanced topics include relativistic MHD (`PHYSICS = RMHD`), adaptive mesh refinement via

Chombo, GPU-accelerated runs via OpenACC, and radiation transport modules. The gPLUTO user guide and the test problem library (`$PLUTO_DIR/Test_Problems/`) provide the starting point for exploring these extensions.

Chapter 5

HPC Environment Setup: NVIDIA HPC-SDK, GPU Acceleration, and Cluster Job Submission

5.1 Overview

The main tutorial chapters (1–4) describe the physics and gPLUTO workflow in a way that is largely independent of the computing environment. This appendix collects all the environment-specific steps required to run gPLUTO with **GPU acceleration via OpenACC** (using the NVIDIA HPC Software Development Kit) and to **submit batch jobs** to three HPC systems used during the IT4I workshop:

- **Karolina** — the primary IT4I flagship supercomputer (AMD processors, 576 NVIDIA A100 GPUs).
- **Barbora** — the secondary IT4I cluster (Intel Cascade Lake processors, 32 NVIDIA Tesla V100 GPUs).
- **Jureca** — the JÜLICH Supercomputing Centre system used for additional GPU runs.
- **Local GPU workstation** — the four interactive nodes at the University of Torino (UniTo) HPC4AI facility, each equipped with an NVIDIA V100-SXM2 32 GB GPU.

The GPU acceleration in gPLUTO is implemented via **OpenACC** pragmas in the source code. The NVIDIA HPC-SDK provides the `nvc++` compiler that understands these pragmas and generates optimized GPU code. Without this compiler, gPLUTO runs on CPUs only and the OpenACC directives are simply ignored.

Do I need GPU acceleration?

GPU acceleration is *optional* for all the problems in Chapters 1–4. The 1D and 2D simulations (Chapters 1–3) run comfortably on a laptop CPU in minutes. The 2D cylindrical jet (Chapter 4) takes a few minutes serially. The 3D Cartesian jet run is the first problem where GPU acceleration gives a significant practical benefit (10–50× speedup over a single core), making it feasible within a workshop session.

5.2 Installing the NVIDIA HPC-SDK on a Local Machine

5.2.1 Downloading and Installing

The NVIDIA HPC-SDK (version 24.5 at the time of the workshop) can be obtained from the NVIDIA developer portal:

https://developer.nvidia.com/nvidia-hpc-sdk-245-downloads

Navigate to the page, accept the license agreement, select **Linux x86_64**, and choose the **Ubuntu (apt)** installation method for Debian-based systems. The installation instructions on that page read:

```
curl https://developer.download.nvidia.com/hpc-sdk/ubuntu/DEB_SIGNING_KEY | \
     sudo gpg --dearmor -o /usr/share/keyrings/nvidia.gpg
echo 'deb [signed-by=/usr/share/keyrings/nvidia.gpg] \
     https://developer.download.nvidia.com/hpc-sdk/ubuntu/amd64 /' | \
     sudo tee /etc/apt/sources.list.d/nvhpc.list
sudo apt-get update -y
sudo apt-get install -y nvhpc-24-5
```

For other Linux distributions (RHEL/Rocky, SLES, tar file), equivalent instructions are shown on the download page — select the appropriate tab.

5.2.2 Configuring the Environment

After installation, add the following lines to your `~/.bashrc` (or `~/.bash_profile`) so that the `nvc++` compiler and the bundled MPI are found automatically at every login:

```
NVARCH=`uname -s`_`uname -m`; export NVARCH
NVCOMPILERS=/opt/nvidia/hpc_sdk; export NVCOMPILERS

MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/24.5/compilers/man
export MANPATH

PATH=$NVCOMPILERS/$NVARCH/24.5/compilers/bin:$PATH
export PATH

export PATH=$NVCOMPILERS/$NVARCH/24.5/comm_libs/mpi/bin:$PATH
export MANPATH=$MANPATH:$NVCOMPILERS/$NVARCH/24.5/comm_libs/mpi/man
```

After editing `.bashrc`, reload it without logging out:

```
source ~/.bashrc
```

5.2.3 Verifying the Installation

Checking the nvc++ Compiler

Type `which nvc` at the terminal. A successful installation prints:

Compiler check: which nvc

```
nvc 24.5-0 64-bit target on x86-64 Linux -tp haswell Copyright (c) 2024, NVIDIA
CORPORATION & AFFILIATES. NVIDIA Compilers and Tools All rights reserved.
```

If you see an error such as `nvc: command not found`, check that the `PATH` line in `.bashrc` is correct and that you have sourced it in the current session.

Checking GPU Visibility

Type `nvidia-smi` to confirm that the NVIDIA driver can see your GPU. A successful output looks like:

If the GPU name, driver version, and CUDA version appear, the installation is successful and gPLUTO can use GPU acceleration.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 535.183.01             Driver Version: 535.183.01   CUDA Version: 12.2     |
|-------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                     |                      |               MIG M. |
|=====================================+======================+======================|
|   0  NVIDIA GeForce GTX 1650 Ti     Off | 00000000:01:00.0 Off |                  N/A |
| N/A   47C    P0              5W /  50W |      6MiB /  4096MiB |      0%      Default |
|                                     |                      |                  N/A |
+-------------------------------------+----------------------+----------------------+
```

Figure 5.1: nvidia-smi output on the terminal window

Troubleshooting: Missing CUDA Toolkit or Driver

If `nvidia-smi` or `nvaccelinfo` returns an error, the GPU driver or CUDA toolkit may not be installed. Go to:

https://developer.nvidia.com/cuda-toolkit

Click "Download now", select your operating system and architecture, and follow the instructions. The page provides both the **CUDA Toolkit Installer** (required for compilation) and the **Driver Installer** (required for runtime). Install both if you are setting up a machine from scratch.

Driver and CUDA version compatibility

The NVIDIA driver version must be compatible with the CUDA version bundled in the HPC-SDK. The HPC-SDK 24.5 ships with CUDA 12.4. Check the NVIDIA compatibility matrix at https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/ to confirm your driver version is sufficient. As a general rule, always install the latest driver available for your GPU.

5.3 The IT4I HPC Systems

5.3.1 Hardware Summary

The workshop provided access to two IT4I supercomputers. Their key specifications are summarized below.

	Karolina	Barbora
Operational since	Summer 2021	Autumn 2019
Peak performance	15 690 Tflop/s	849 Tflop/s
OS	Rocky Linux 8.9	CentOS 7.x
CPU compute nodes	720× 2× AMD 7H12 (64 cores, 2.6 GHz) + 72× 2× AMD 7763 (64 cores, 2.45 GHz) + 32× Intel Xeon SC 8628 (24 cores, 2.9 GHz)	201× 2× Intel Cascade Lake (18 cores, 2.6 GHz)
GPU accelerators	576× NVIDIA A100	32× NVIDIA Tesla V100
Documentation	https://docs.it4i.cz	https://docs.it4i.cz
Workshop CPU hours	1 000	200
Workshop GPU hours	500	200

5.4 Connecting to Karolina and Barbora

5.4.1 Initial SSH Login

Both systems use key-based SSH authentication. Your workshop account username follows the pattern `atr-25-5-XXX` where `XXX` is your assigned user number.

Karolina:

```
ssh -i /path/to/.ssh/key atr-25-5-XXX@login1.karolina.it4i.cz
```

Barbora:

```
ssh -i /path/to/.ssh/key atr-25-5-XXX@login1.barbora.it4i.cz
```

5.4.2 Enabling X11 Forwarding for GUI Applications

Some visualization tools (e.g., Gnuplot interactive windows) require X11 forwarding. Enable it by first creating the `.Xauthority` file and then reconnecting with the `-X` flag:

```
# First login: create the X authority file
ssh -i /path/to/.ssh/key atr-25-5-XXX@login1.karolina.it4i.cz
touch ~/.Xauthority
exit

# Second login: enable X11 forwarding
ssh -i -X /path/to/.ssh/key atr-25-5-XXX@login1.karolina.it4i.cz
```

5.5 Setting Up the Software Environment on Karolina and Barbora

After logging in, the following steps prepare your environment for gPLUTO. The commands differ slightly between Karolina (NVHPC 25.3 / CUDA 12.8) and Barbora (NVHPC 24.3 / CUDA 12.3).

5.5.1 Step 1: Download gPLUTO

```
git clone https://gitlab.com/PLUTO-code/gPLUTO.git
```

This creates a `gPLUTO/` directory in your home folder containing the full source tree.

5.5.2 Step 2: Set the PLUTO_DIR Environment Variable

Add this to your `.bashrc` so it is available in every session (replace the path with your actual username):

```
export PLUTO_DIR=/home/atr-25-5-XXX/gPLUTO
```

For the current session, also source it immediately:

```
source ~/.bashrc
```

5.5.3 Step 3: Create a Python Virtual Environment and Load Modules

A Python virtual environment isolates PyPLUTO and its dependencies from the system Python installation, avoiding version conflicts.

On Karolina (NVHPC 25.3, CUDA 12.8):

```
# Create the virtual environment
python3 -m venv workshop
source workshop/bin/activate

# Load the required modules
ml Python/3.12.3-GCCcore-14.2.0
ml NVHPC/25.3-CUDA-12.8.0

# Set the NVHPC version path for library exports
export ver=/apps/all/NVHPC/25.3-CUDA-12.8.0/Linux_x86_64/25.3

# Install PyPLUTO from the gPLUTO source tree
cd gPLUTO/Tools/PyPLUTO/
pip install ./
```

On Barbora (NVHPC 24.3, CUDA 12.3):

```
# Create the virtual environment
python3 -m venv workshop
source workshop/bin/activate

# Load the required modules
ml NVHPC/24.3-CUDA-12.3.0
ml Python/3.12.3-GCCcore-13.3.0

# Set the NVHPC version path for library exports
export ver=/apps/all/NVHPC/24.3-CUDA-12.3.0/Linux_x86_64/24.3

# Install PyPLUTO from the gPLUTO source tree
cd gPLUTO/Tools/PyPLUTO/
pip install ./
```

5.5.4 Step 4: Export NCCL Library Paths

The NCCL library (NVIDIA Collective Communications Library, used for multi-GPU MPI communication) must be explicitly added to the search paths. The path structure differs slightly

between Karolina and Barbora due to differing NVHPC versions.

On Karolina:

```
export CPATH="$CPATH:$ver/comm_libs/nccl/include"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$ver/comm_libs/nccl/lib"
export LIBRARY_PATH="$LIBRARY_PATH:$ver/comm_libs/nccl/lib"
```

On Barbora:

```
export CPATH="$CPATH:$ver/comm_libs/nccl/12.3/include"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:$ver/comm_libs/12.3/nccl/lib"
export LIBRARY_PATH="$LIBRARY_PATH:$ver/comm_libs/12.3/nccl/lib"
```

Adding environment exports to .bashrc

To avoid repeating these `export` commands at every login, append the module loads and `export` lines to your `~/.bashrc`. The module loads should come after the virtual environment activation line to ensure the correct Python and compiler are always available.

5.6 Slurm Job Submission on Karolina and Barbora

Both Karolina and Barbora use the **Slurm** workload manager for scheduling batch jobs. Interactive use on the login nodes is only for editing and compiling — actual simulations must be submitted through Slurm.

5.6.1 Understanding the Partitions

Partition	Hardware	Use case
`qcpu_exp`	CPU nodes	Short CPU jobs for testing (up to 1h)
`qgpu_exp`	GPU nodes	Short GPU jobs for testing (up to 1h)
`qcpu`	CPU nodes	Standard CPU production runs
`qgpu`	GPU nodes	Standard GPU production runs

For workshop exercises, the `_exp` (express) partitions are the right choice: they have short queue times but are limited to 1 hour wall time.

5.6.2 The sbatch Script

A minimal Slurm submission script for gPLUTO looks identical on both systems (only the NVHPC module version number differs):

Karolina sbatch script:

```
#!/usr/bin/bash
#SBATCH --job-name  pluto
#SBATCH --account   ATR-25-5
#SBATCH --partition qcpu_exp    # or qgpu_exp for GPU runs
#SBATCH --ntasks=1              # 1 MPI task
#SBATCH --cpus-per-task=1       # 1 CPU core per task
##SBATCH --gpus=1               # uncomment for GPU partition only
#SBATCH --time 00:01:00         # wall time limit HH:MM:SS

export PLUTO_DIR=/home/atr-25-5-XXX/gPLUTO
ml NVHPC/25.3-CUDA-12.8.0
```

```
./pluto > pluto.log
```

Barbora sbatch script:

```
#!/usr/bin/bash
#SBATCH --job-name  pluto
#SBATCH --account   ATR-25-5
#SBATCH --partition qcpu_exp   # or qgpu_exp for GPU runs
#SBATCH --ntasks=1             # 1 MPI task
#SBATCH --cpus-per-task=1      # 1 CPU core per task
##SBATCH --gpus=1              # uncomment for GPU partition only
#SBATCH --time 00:01:00        # wall time limit HH:MM:SS

export PLUTO_DIR=/home/atr-25-5-XXX/gPLUTO
ml NVHPC/24.3-CUDA-12.3.0

./pluto > pluto.log
```

Submit the script from inside the problem directory (where the compiled `pluto` executable and `pluto.ini` reside):

```
sbatch myjob.sh
```

5.6.3 Key Slurm Directives Explained

Directive	Meaning
`-job-name`	A short human-readable name for the job, shown in the queue listing
`-account`	The project account to charge. For the workshop: `ATR-25-5`
`-partition`	Which queue to submit to (`qcpu_exp` for CPU, `qgpu_exp` for GPU)
`-ntasks`	Number of MPI processes. Set to the number of cores/GPUs you want
`-cpus-per-task`	CPU cores per MPI task. For GPU runs, match to the number of hardware threads per GPU
`-gpus`	Number of GPUs to reserve. **Only uncomment when submitting to a GPU partition**
`-time`	Maximum wall-clock time in `HH:MM:SS`. The job is killed if it exceeds this

5.6.4 Useful Slurm Commands

Command	Purpose
`sbatch myjob.sh`	Submit a batch job
`squeue -u $USER`	Show your jobs currently in the queue
`scancel <jobid>`	Cancel a queued or running job
`sinfo -p qgpu_exp`	Show the status of a specific partition
`sacct -j <jobid>`	Show accounting information after a job completes

5.7 Interactive GPU Nodes at UniTo HPC4AI

During the workshop, four dedicated interactive GPU nodes at the University of Torino **HPC4AI** facility were made available, courtesy of M. Aldinucci and S. Rabellino (Department of Computer Science, UniTo).

5.7.1 Node Specifications

Each of the four nodes is equipped with:

- **GPU:** 1× NVIDIA V100-SXM2-32GB
- **CPU:** 4 cores Intel Xeon (Skylake, IBRS)
- **RAM:** 32 GB

5.7.2 How to Connect

The connection address follows the pattern:

```
plutosymp<a><n>@pluto-symposium-node<N>.hpc4ai.unito.it
```

where `<a>` is a letter (`a`, `b`, `c`, or `d`) identifying your user group, `<n>` is your user number within that group (1 to 4), and `<N>` is the node number (1 to 4). The password was distributed via the workshop Webex chat. Example:

```
ssh -X plutosympb2@pluto-symposium-node3.hpc4ai.unito.it
```

5.7.3 After Connecting

After logging in, the environment is pre-configured. From your `$HOME` directory there is a `../shared` folder containing a `bashrc` file with the correct `PATH` for `nvc++`:

```
# Source the pre-configured environment
source ~/shared/bashrc

# Gnuplot is directly available for 2D plots
gnuplot
```

gPLUTO has already been cloned and the relevant test problems compiled on these nodes. Participants can immediately run simulations and visualize results without any further setup steps.

5.8 Running gPLUTO on Jureca at JSC

The JÜLICH Supercomputing Centre's **Jureca** system provides an additional route to GPU-accelerated runs. The workflow on Jureca is interactive rather than batch-based for the workshop exercises.

5.8.1 Step-by-Step Workflow

The numbered steps below match the workflow shown in the workshop slides:

Step 1 — Download gPLUTO:

```
git clone https://gitlab.com/PLUTO-code/gPLUTO.git
```

Step 2 — Request an interactive GPU node:

```
salloc --account=punch_astro \
       --partition=dc-gpu    \
       --time 01:00:00       \
       --gres=gpu:4          \
       --nodes 1
```

This requests 1 node with 4 GPUs for 1 hour. Once the allocation is granted, Slurm will return a job number and a node name.

Step 3 — Enter the interactive node:

```
sgoto <job_number> <node_number>
# node_number is relative: 0 = first node in your allocation
```

Step 4 — Load modules:

```
module purge
module load NVHPC/24.3-CUDA-12
module load OpenMPI
```

Step 5 — Set PLUTO_DIR:

```
export PLUTO_DIR=/<fullpath>/gPLUTO
# Add this to your .bashrc for persistence
```

Step 6 — Navigate to the problem folder and copy configuration:

```
cd $PLUTO_DIR/Test_Problems/Simple_Tests/Jet
# Copy the desired definitions and ini files as described in Chapter 4
cp definitions_01.hpp definitions.hpp
cp pluto_01.ini      pluto.ini
```

Step 7 — Problem setup:

```
python3 $PLUTO_DIR/setup.py
```

Step 8 — Compile:

```
make -j
```

Step 9 — Run (serial or parallel):

```
# Serial run (1 GPU)
./pluto

# Parallel run (4 GPUs, one MPI task per GPU)
srun -w $HOSTNAME --ntasks 4 --cpus-per-task 32 --gres=gpu:4 ./pluto
```

The `srun` command inside an `salloc` allocation launches the job on the same node that was al-

located, using 4 MPI tasks (one per GPU) with 32 CPU threads each. gPLUTO automatically assigns one GPU per MPI task when compiled with the OpenACC makefile (`linux.nvc++.acc.defs`).

5.9 Choosing the Right Makefile for GPU Runs

gPLUTO ships with several pre-configured makefiles in `$PLUTO_DIR/Config/`. The choice of makefile determines the compiler and acceleration mode:

Makefile	Use case
`Linux.gcc.defs`	Serial CPU run using GCC. No MPI, no GPU. Simplest option for testing on a laptop.
`linux.mpicxx.defs`	Multi-core CPU run using MPI and the system `mpicxx` wrapper. No GPU.
`linux.nvc++.acc.defs`	GPU-accelerated run using NVIDIA `nvc++` with OpenACC. Requires HPC-SDK.
`linux.nvc++.acc.mpi.defs`	GPU + MPI: multiple GPUs across one or more nodes. Requires HPC-SDK and a compatible MPI.

To select a makefile, copy it to the problem directory:

```
cp $PLUTO_DIR/Config/linux.nvc++.acc.defs makefile
make -j
```

Or run `python3 $PLUTO_DIR/setup.py` and select the makefile interactively from the *Select makefile* menu.

Checking GPU utilization during a run

While gPLUTO is running on a GPU, open a second terminal on the same node and type:

```
watch -n 1 nvidia-smi
```

This refreshes the GPU status every second. During an active GPU run you should see `GPU-Util` at or near 100% and `Memory-Usage` increasing as the simulation proceeds.

5.10 Quick-Reference Checklist

The table below provides a concise checklist for getting gPLUTO running in each environment:

Task	Command / action
Install HPC-SDK (local)	Follow apt instructions from developer.nvidia.com/nvidia-hpc-sdk-245-downloads
Add compiler to PATH	Edit ~/.bashrc with the NVARCH/NVCOMPILERS/PATH lines
Verify compiler	which nvc
Verify GPU	nvidia-smi
Clone gPLUTO	git clone https://gitlab.com/PLUTO-code/gPLUTO.git
Set PLUTO_DIR	export PLUTO_DIR=~/gPLUTO (add to .bashrc)
Create Python venv	python3 -m venv workshop && source workshop/bin/activate
Install PyPLUTO	pip install $PLUTO_DIR/Tools/PyPLUTO/
Load modules (Karolina)	ml Python/3.12.3-GCCcore-14.2.0 && ml NVHPC/25.3-CUDA-12.8.0
Load modules (Barbora)	ml Python/3.12.3-GCCcore-13.3.0 && ml NVHPC/24.3-CUDA-12.3.0
Select GPU makefile	Copy linux.nvc++.acc.defs to makefile in problem dir
Compile	make -j
Run (serial)	./pluto
Run (MPI/parallel)	mpirun -np <n> ./pluto
Run (Jureca multi-GPU)	srun -w $HOSTNAME --ntasks 4 --cpus-per-task 32 --gres=gpu:4 ./pluto
Submit batch job	sbatch myjob.sh
Monitor queue	squeue -u $USER
Monitor GPU	watch -n 1 nvidia-smi

5.11 Summary

This appendix has covered all the environment-specific steps needed to run gPLUTO beyond a standard laptop installation:

- The **NVIDIA HPC-SDK** provides the nvc++ compiler and OpenACC runtime that enable GPU acceleration in gPLUTO. It installs via apt on Ubuntu/Debian and requires both an NVIDIA driver and the CUDA toolkit.
- **Karolina** and **Barbora** differ only in the NVHPC module version (25.3 vs. 24.3) and the corresponding NCCL library path structure. The Python virtual environment and PyPLUTO installation steps are otherwise identical.
- **Slurm** is the workload manager on all IT4I systems. The key directives for gPLUTO jobs are -ntasks (MPI ranks), -gpus (GPU count, GPU partition only), and -partition (qcpu_exp or qgpu_exp for workshop-scale jobs).
- The **UniTo HPC4AI interactive nodes** provide immediate access to a pre-configured environment with nvc++ and an NVIDIA V100-SXM2 32 GB GPU, suitable for all four tutorial problem sets.
- On **Jureca**, the interactive salloc workflow combined with srun allows multi-GPU parallel runs (up to 4 GPUs) within a single interactive session, without writing a batch script.