PythonMHD: A New Simulation Code for Astrophysical Magnetohydrodynamics

By Delica Leboe-McGowan,
PhD Student,
University of Manitoba

Version 2024.1 (First Public Version) Released on GitHub on January 1, 2024

User Guide

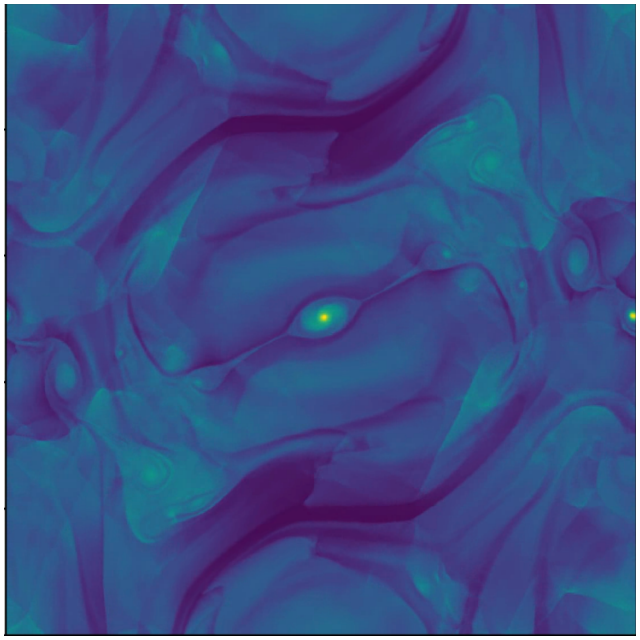

Table of Contents

PythonMHD Mission Statement

PythonMHD is a new software package for astrophysical magnetohydrodynamic (MHD) simulations. Although it is a widely understood programming language in the physical sciences, Python has never previously been used to develop a comprehensive, research-oriented open-source MHD simulation code. All of the most widely used MHD simulation codes are written in lower-level languages, such as C, C++, and FORTRAN. These programming languages are difficult to interpret and, thereby, exacerbate the learning curves associated with MHD software packages. When I first decided to build PythonMHD as a Joint Honours student in Computer Science & Physics at the University of Manitoba, my goal was to create an MHD simulation code that anyone with a basic understanding of Python could easily run. By creating an accessible MHD simulation code written in Python, it is my greatest hope that PythonMHD will attract more individuals into the exciting field of astrophysical MHD. I believe that maximizing the number of people who can actively engage with this area of research is the most effective way to advance scientific progress. In the coming weeks, months, and years, I look forward to building upon this first version of PythonMHD, striving to ensure that the user's experience is always productive, enjoyable, and straightforward.

Acknowledgements

Here I would like to express my appreciation for everyone who has helped PythonMHD become a reality over the past three years.

I would first like to thank my supervisor, Dr. Jason Fiege, who has provided me with endless guidance and support since I entered the Department of Physics & Astronomy at the University of Manitoba as an undergraduate student. One of the best decisions of my bachelor's degree was challenging an introductory computer science course so I could take Dr. Fiege's first year course on astrophysics. That course fostered a passion for computational astrophysics that continues to motivate me today, and I will always be grateful to Dr. Fiege for giving me the opportunity to pursue this passion as one of his graduate students.

I also would like to thank Dr. Jayanne English for all of times that she has advised me on how I could improve the scientific visualization tools in PythonMHD. In particular, I am incredibly grateful to Dr. English for the invaluable feedback that she has given me on PythonMHD's line integral convolution (LIC) algorithm for visualizing magnetic field lines, which will become available on the PythonMHD GitHub page within the next few weeks.

William Grafton is another person who has provided critical support and advice since the earliest stages of PythonMHD's development. The inspiration for PythonMHD arose from the challenges we both encountered while attempting to simulate molecular cloud filaments with the existing MHD software options during a special topics course. Since I began programming PythonMHD as a solution to these challenges, Will has provided countless suggestions that have turned PythonMHD into a more intuitive, user-friendly piece of software.

More recently, Aaron Nédélec helped me test an earlier version of PythonMHD as part of their Honours project at the University of Manitoba. It took Aaron less than thirty minutes to

become a complete master of PythonMHD. I will always be grateful for Aaron's dedication and enthusiasm as one of the first people to use PythonMHD for their own research.

I would be remiss not to thank the students of PHYS 4250 (Computational Physics) at the University of Manitoba, who have played a key role in helping me develop the educational applications of PythonMHD. Despite having no or little previous experience with MHD simulations, students in two offerings of PHYS 4250 bravely experimented with the MATLAB version of PythonMHD. I am still deeply impressed by how quickly they learned to program complex simulation algorithms. Any teacher would appreciate their open-mindedness, work ethic, and persistence.

Lastly, I would like to acknowledge all of the computational physicists who have developed essential algorithms for astrophysical MHD simulations. The existing MHD simulation code that most directly informed PythonMHD's development is Athena, which is why I feel particularly indebted to Dr. James Stone and colleagues for their tremendous contributions to this field.

Installation and Setup

Downloading PythonMHD

Step 1: Go to the PythonMHD GitHub page: https://github.com/DelicaLM/PythonMHD

Step 2: Download PythonMHD as a zip folder.

Step 3: Unzip/decompress PythonMHD wherever you want it to live on your machine.

Opening PythonMHD in PyCharm

(Note: I wrote these instructions for a PyCharm IDE because it is the environment that I used throughout PythonMHD's development. You can definitely run PythonMHD in another IDE, such as Visual Studio, but these particular instructions will not apply. If your preferred IDE is not PyCharm, you can skip to the Required Python Libraries section below.)

Step 1: Download the free community edition of PyCharm:

https://www.jetbrains.com/help/pycharm/installation-guide.html

Step 2: Open PyCharm and select the "Open" folder option shown in Figure 1. (Note: These screenshots might not exactly match your version of PyCharm. The buttons and drop-down menus might be in different locations, but the same instructions otherwise apply.)

Step 3: Select the PythonMHD folder (wherever it exists on your machine). You will now have a PythonMHD project in PyCharm. In the next section, there are instructions on how to run PythonMHD after you have created this project.

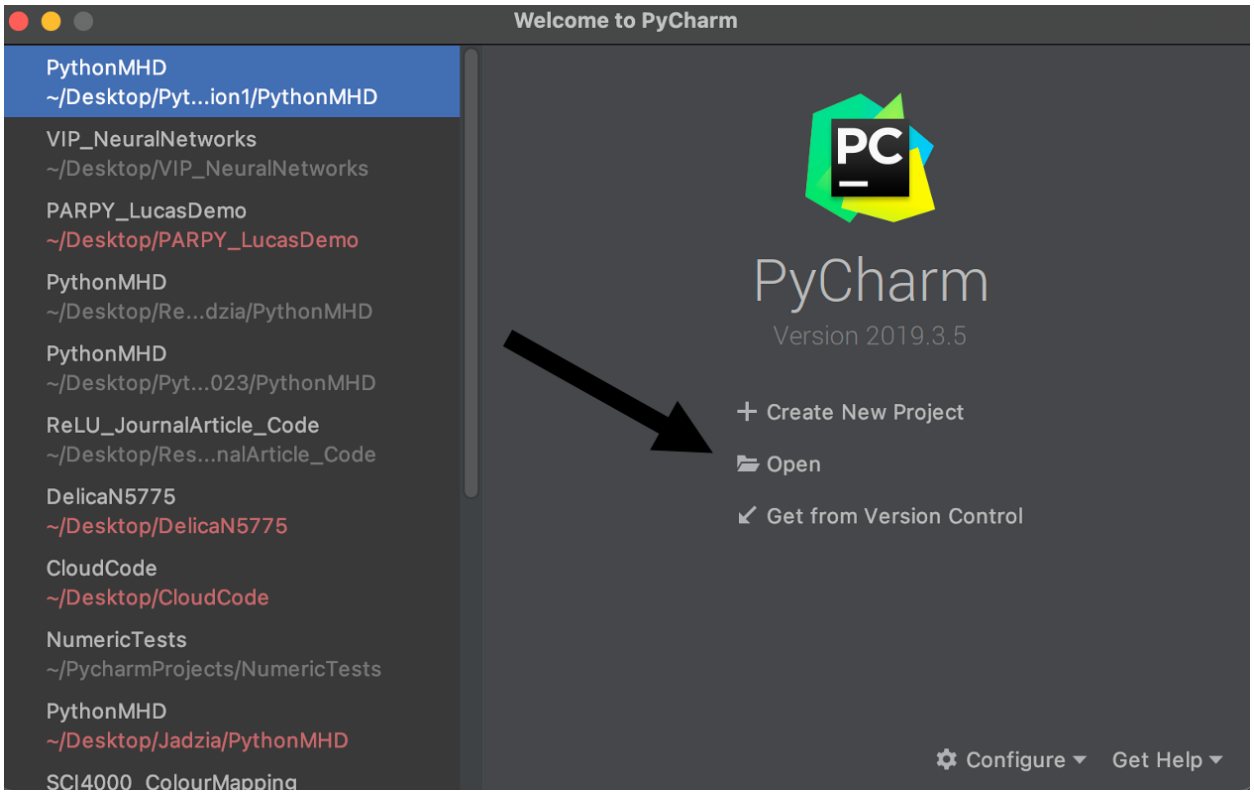

Figure 1: Screenshot of first PyCharm window, where you should pick the "Open" folder option if you do not already have a PythonMHD project set up in PyCharm.

Running a PythonMHD Initialization File

A PythonMHD initialization file is the script that creates your PythonMHD simulation. More details are provided in the "How to Run a PythonMHD Simulation" section below. The initialization files are the only part of the code that you should directly modify (unless you want to change default values in the PythonMHD constants file). The source files in PythonMHD, which you should avoid changing, will evolve your simulation after you set its parameters and initial conditions. The steps in this section will show you how to run an initialization file that you create or any of the sample initialization files that come with your PythonMHD download.

Step 1: Press the runtime configuration bar in the top-right corner of PyCharm when you are in your PythonMHD project, as shown in Figure 2.

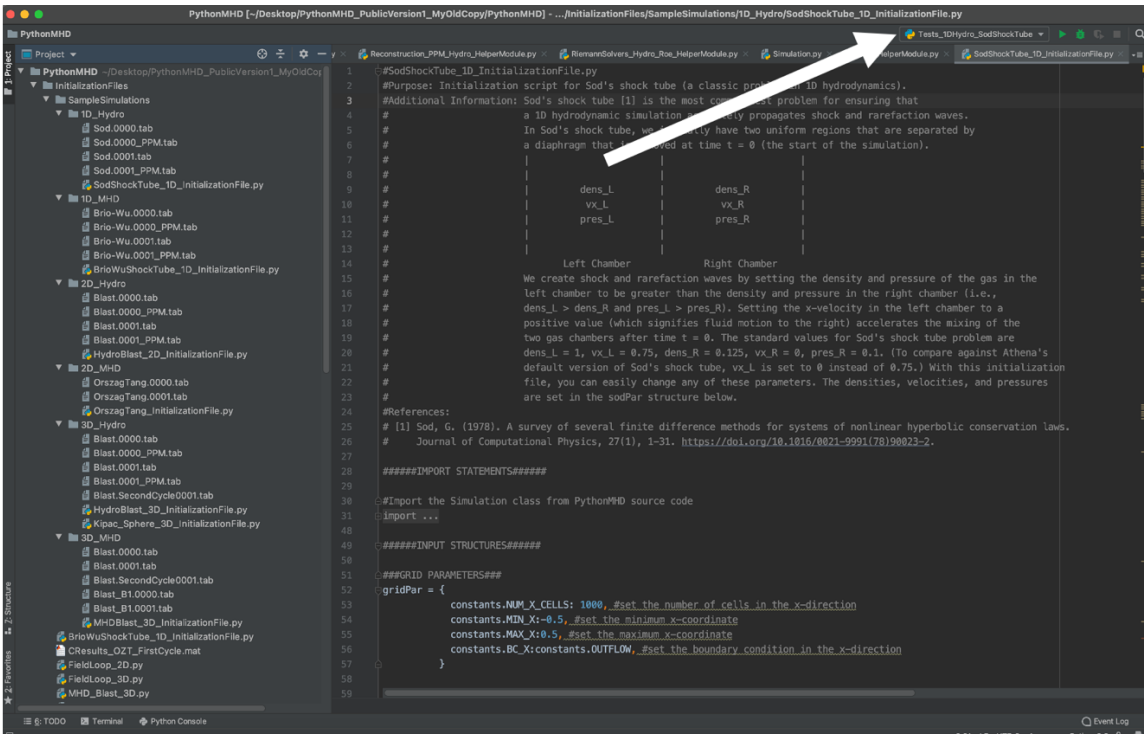

Figure 2: Screenshot of selecting the runtime configuration drop-down menu in PyCharm.

Step 2: Pick the "Edit Configurations…" option from the drop-down menu shown in Figure 3.

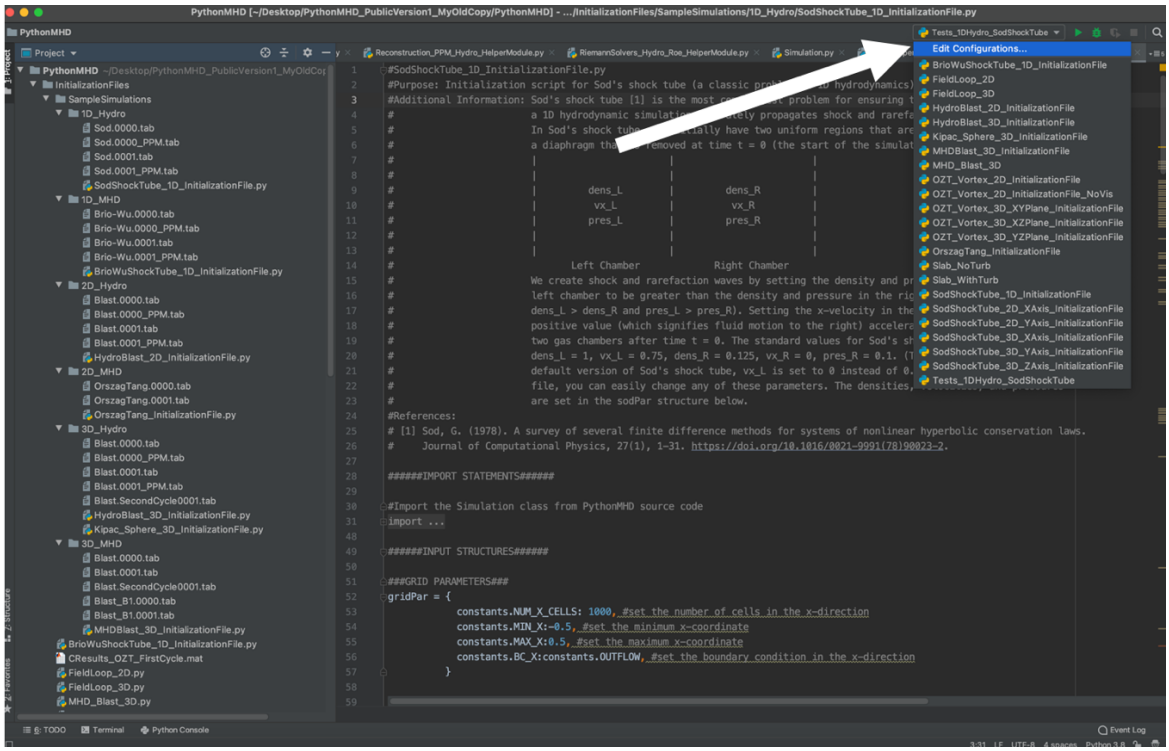

Figure 3: Screenshot of selecting the "Edit Configurations…" option from the runtime configuration drop-down menu in PyCharm.

Step 3: In the runtime configurations window shown in Figure 4, press the "+" button in the top-left corner to create a new runtime configuration.

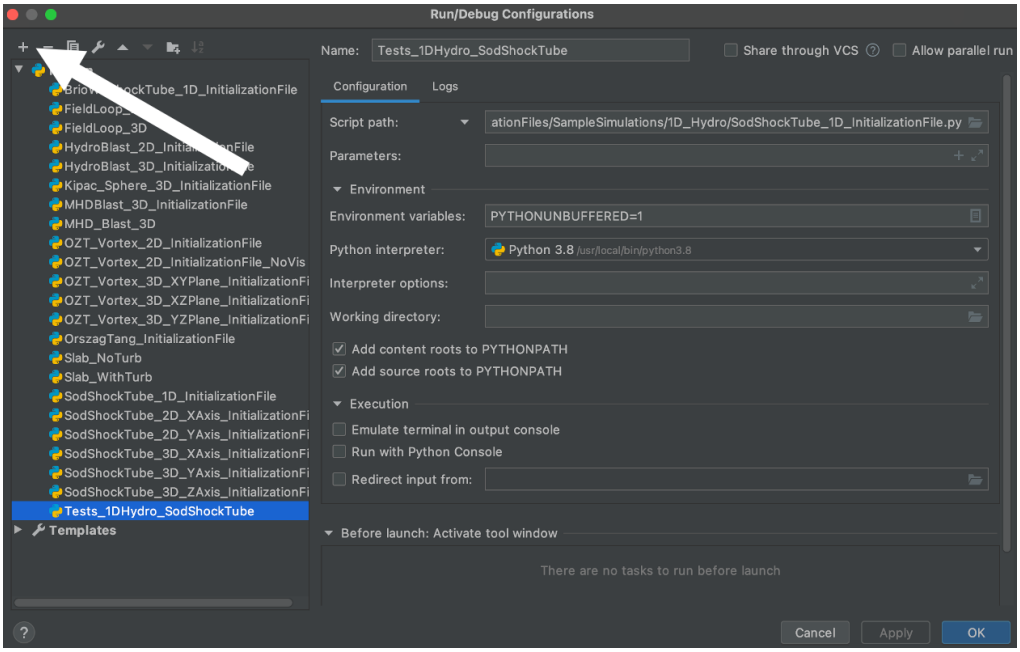

Figure 4: Screenshot of selecting the "+" button to create a new runtime configuration in PyCharm.

Step 4: Choose "Python" from the Figure 5 drop-down menu of possible runtime configuration types.

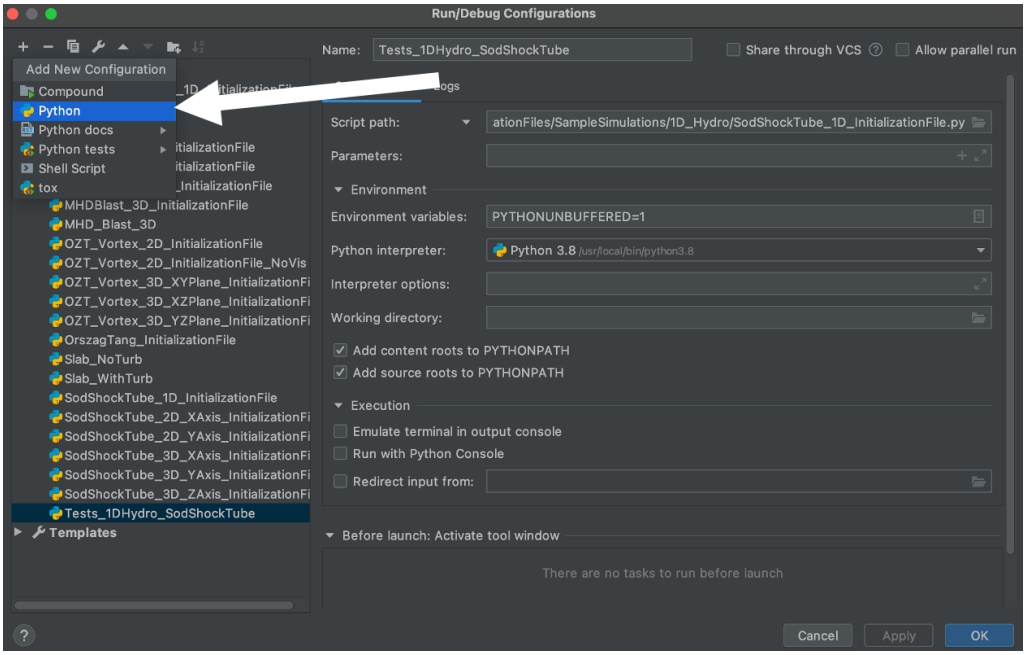

Figure 5: Screenshot of selecting Python for a new PyCharm runtime configuration.

Step 5: Press the folder icon next to “Script Path” (see Figure 6) to choose the PythonMHD initialization script that you want to run (e.g., SodShockTube_1D_InitializationFile.py in the InitializationFiles/SampleSimulations/1D_Hydro directory). (Note: For the Python interpreter (a few options below “Script Path”), I recommend using at least Python 3.8. This Python version is the one that I have used for all PythonMHD testing thus far, so I can’t guarantee that all imported libraries will be happy if you try to use an older iteration of Python. In the future, I will post an update on the absolute minimum Python version that PythonMHD requires.)

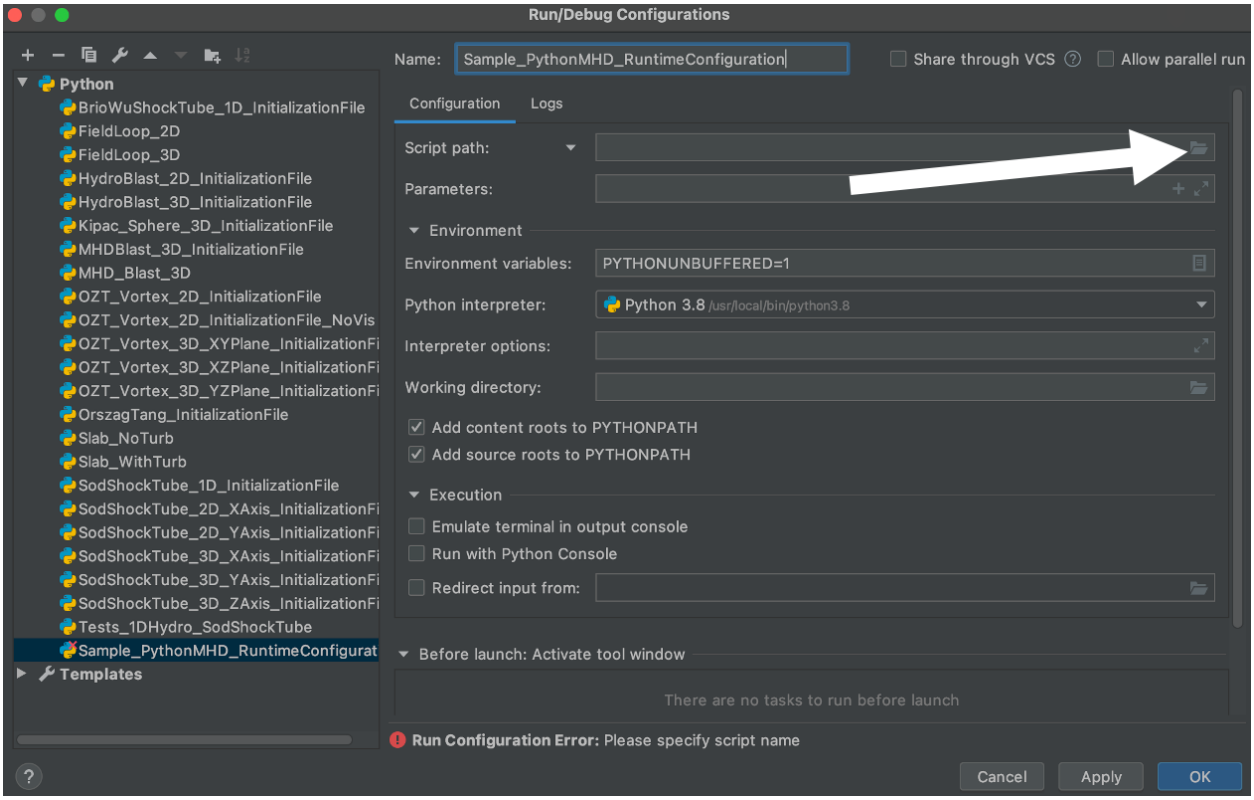

Figure 6: Screenshot of pressing the folder icon next to “Script Path” in order to choose the Python script that you want to run.

Step 5: Now when you press the runtime configuration bar shown in Figure 2, your new runtime configuration will be one of the options in the drop-down menu. Select this runtime configuration, and then run the simulation by pressing the green arrow shown in Figure 7.

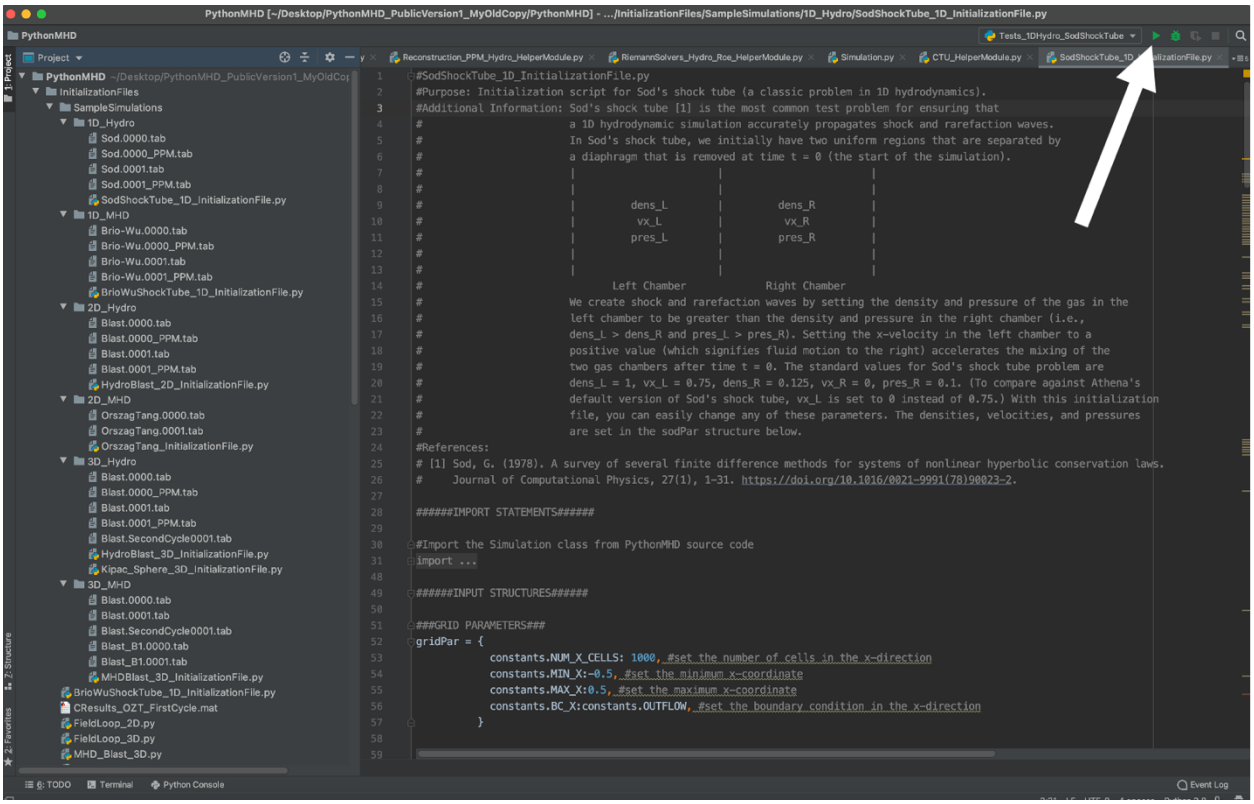

Figure 7: Screenshot of pressing the green arrow/"Run" button next to the runtime configurations bar, which should be set to the runtime configuration that you want to use for the PythonMHD simulation.

Installing Required Python Libraries

If errors occurred when you tried to run the sample PythonMHD initialization script in the last step, you most likely do not have all of the required python libraries installed on your machine. Below is a list of all of the libraries that are required to take full advantage of PythonMHD:

- numpy (for matrix operations)
- math (for retrieving the value of π)
- matplotlib (for visualizing simulation data) (Note: In order to fully use PythonMHD visualizations, you also need to have an interactive matplotlib backend (https://matplotlib.org/stable/users/explain/figure/backends.html) installed on your machine. I highly recommend using the TkAgg backend if possible, because it is the

backend that I used for all of my PythonMHD testing. In the future, I will post an update on how PythonMHD reacts to other matplotlib backends.)

- mpl_toolkits (for animating colorbars and making 3D data plots)
- scipy.io (for creating .mat files of simulation data)
- pyevtk (for creating .vtu (unstructured points vtk files) files of simulation data)
- glob (for retrieving the image files that will be the frames of simulation movies)
- cv2 (for combining the output images to make a .mp4 movie file)

All of these libraries can be downloaded with "pip install" statements.

How to Run a PythonMHD Simulation

For each type of PythonMHD simulation (1D Hydro, 2D Hydro, 3D Hydro, 1D MHD, 2D MHD, 3D MHD), there is a sample initialization file (i.e., the PythonMHD script you use to create and run a PythonMHD simulation) that demonstrates all of the steps that we will explore in this section. As was mentioned in the "Running a PythonMHD Initialization File" section, the initialization file is where you set the parameters and initial conditions for a PythonMHD simulation.

At the top of the initialization file, you should import numpy for setting up your gas variables (see the "Initial Conditions" section below) and some PythonMHD modules that will help you easily set up your simulation.

The "from Source.Simulation import Simulation" line in Figure 8 imports the PythonMHD simulation class. When you create a PythonMHD simulation, you are actually creating an instance of this class.

The "import Source.Initialization_HelperModule as initHelper" imports functions that will help you create the (x, y, z) coordinate for your simulation.

Finally, the “import Source.PythonMHD_Constants as constants” line imports PythonMHD constants that will let you set your input parameters without having to remember their exact names. For example, you won’t need to worry about whether the name for the parameter that sets the number of cells in the x-direction is “numXCells”, “num_x_cells”, or “NumXCells”; you can make sure that you have the correct string by using the constants.NUM_X_CELLS parameter string constant (see the following section for more details).

```
#Import numpy for matrix operations
import numpy as np

#Import the Simulation class from PythonMHD source code
from Source.Simulation import Simulation

#Import the initialization helper module to find the coordinates of every cell in your simulation grid
import Source.Initialization_HelperModule as initHelper

#Import the module that contains all PythonMHD constants (convenient for setting input parameters)
import Source.PythonMHD_Constants as constants
```

Figure 8: Screenshot of import statements at the top of a PythonMHD initialization file/script.

Setting Up Your Simulation Grid

The first step is to define the grid for the simulation. PythonMHD uses a finite volume algorithm, which means that any simulation is divided into cells/pixels with a finite width, height, and depth. This first version only supports Cartesian simulation grids, which means that all of the cells are rectilinear (i.e., they are all rectangular cuboids/parallelepipeds) and that we define their locations with x-, y-, and z-coordinates. Future versions of PythonMHD will include support for polar, cylindrical, and spherical coordinates.

The Table below defines all of the parameters that you can use for creating your Cartesian simulation grid.

PythonMHD Grid Parameters (gridPar)			
Parameter Name	Purpose	Acceptable Inputs	Required or Optional
constants.NUM_X_CELLS or "numXCells"	Integer for the number of cells in the x-direction	Any int greater than zero (e.g., 100, 200, etc.)	Required for all simulations (because any simulation is at least 1D)
constants.MIN_X or "minX"	Numeric value for the minimum x-position/coordinate	Any int or float less than the maximum x-position	Required for all simulations
constants.MAX_X or "maxX"	Numeric value for the maximum x-position/coordinate	Any int or float greater than the minimum x-position	Required for all simulations
constants.NUM_Y_CELLS or "numYCells"	Integer for the number of cells in the y-direction	Any int greater than zero (e.g., 100, 200, etc.)	Required for 2D and 3D simulations
constants.MIN_Y or "minY"	Numeric value for the minimum y-position/coordinate	Any int or float less than the maximum y-position	Required for 2D and 3D simulations
constants.MAX_Y or "maxY"	Numeric value for the maximum y-position/coordinate	Any int or float greater than the minimum y-position	Required for 2D and 3D simulations
constants.NUM_Z_CELLS or "numZCells"	Integer for the number of cells in the z-direction	Any int greater than zero (e.g., 100, 200, etc.)	Required for 3D simulations
constants.MIN_Z or "minZ"	Numeric value for the minimum z-position/coordinate	Any int or float less than the maximum z-position	Required for 3D simulations
constants.MAX_Z or "maxZ"	Numeric value for the maximum z-position/coordinate	Any int or float greater than the minimum z-position	Required for 3D simulations

Now we will look at how these grid parameters are set in a PythonMHD initialization file. Figures 9-11, respectively, show the PythonMHD grid parameter structures that correspond to 1D, 2D, and 3D simulations.

```
###GRID PARAMETERS###
gridPar = {
            constants.NUM_X_CELLS: 1000, #set the number of cells in the x-direction
            constants.MIN_X:-0.5, #set the minimum x-coordinate
            constants.MAX_X:0.5, #set the maximum x-coordinate
            constants.BC_X:constants.OUTFLOW, #set the boundary condition in the x-direction
          }
```

Figure 9: Screenshot of the grid parameters/gridPar structure for a 1D simulation.

```
###GRID PARAMETERS###
gridPar = {
            constants.NUM_X_CELLS: 200, #set the number of cells in the x-direction
            constants.MIN_X:-0.5, #set the minimum x-coordinate
            constants.MAX_X:0.5, #set the maximum x-coordinate
            constants.NUM_Y_CELLS: 300, #set the number of cells in the y-direction
            constants.MIN_Y:-0.75, #set the minimum y-coordinate
            constants.MAX_Y:0.75, #set the maximum y-coordinate
            constants.BC_X:constants.PERIODIC, #set the boundary condition in the x-direction
            constants.BC_Y:constants.PERIODIC, #set the boundary condition in the y-direction
          }
```

Figure 10: Screenshot of the grid parameters/gridPar structure for a 2D simulation.

```
###GRID PARAMETERS###
gridPar = {
            constants.NUM_X_CELLS: 100, #set the number of cells in the x-direction
            constants.MIN_X:-0.5, #set the minimum x-coordinate
            constants.MAX_X:0.5, #set the maximum x-coordinate
            constants.NUM_Y_CELLS: 150, #set the number of cells in the y-direction
            constants.MIN_Y:-0.75, #set the minimum y-coordinate
            constants.MAX_Y:0.75, #set the maximum y-coordinate
            constants.NUM_Z_CELLS: 100, #set the number of cells in the z-direction
            constants.MIN_Z: -0.5, #set the minimum z-coordinate
            constants.MAX_Z: 0.5, #set the maximum z-coordinate
            constants.BC_X:constants.PERIODIC, #set the boundary condition in the x-direction
            constants.BC_Y:constants.PERIODIC, #set the boundary condition in the y-direction
            constants.BC_Z:constants.PERIODIC, #set the boundary condition in the z-direction
          }
```

Figure 11: Screenshot of the grid parameters/gridPar structure for a 3D simulation.

Simulation Parameters

Most details about your simulation (except the initial gas state) are specified in a Python dictionary of simulation parameters, which is referred to as simPar in PythonMHD. These parameters tell PythonMHD if your simulation has magnetic fields, when it should stop the simulation, the specific heat ratio/gamma for the ideal gas, whether you want to visualize the gas

while the simulation is still running, whether you want to save the visualizations that are created, where you want to save output files, whether you want to save numerical data from your simulation, and many other pieces of information that will help PythonMHD satisfy all of your simulation needs. All of the parameters that you can set in simPar are described in the table below.

PythonMHD Simulation Parameters (simPar)			
Parameter Name	Purpose	Acceptable Inputs	Required or Optional
constants.IS_MHD or "isMHD"	Boolean flag for whether the simulation has magnetic fields	True (if the simulation has magnetic fields) False (if the simulation does not have magnetic fields)	Optional for hydrodynamics (because the default value is False) Required (and must be set to True) for MHD simulations
constants.TLIM or "tLim"	Numeric value (int or float) for when the simulation should end (As long as no issues occur, PythonMHD will simulate your gas system from time t = 0 to this time limit.)	Any integer or float greater than 0 (e.g., 1.0, 3, 0.1, etc.)	Required for all simulations
constants.MAX_CYCLES or "maxCycles"	Max number of simulation cycles (i.e., number of times PythonMHD calculates a new simulation state) (useful for ensuring that any simulation will eventually end, even if numerical instabilities make	Any integer greater than 0 (e.g., 1000, 20000, etc.)	Optional (default value is 10000)

	the timesteps excessively small)		
constants.GAMMA or “gamma”	Specific heat ratio/gamma value for the ideal gas in your simulation	Any integer or float greater than 0 (make sure that you choose a specific heat ratio that makes sense for the type of gas you want to simulate)	Optional (default value is 1.4)
constants.CFL or “cfl”	Courant-Friedrichs-Lewy (CFL) number (i.e., fraction of a grid cell that the fastest wave should travel in one simulation cycle) for calculating timestep sizes	Any float greater than 0 and less than 1 for 1D and 2D simulations (e.g., 0.8, 0.9, etc.) Any float greater than 0 and less than 0.5 for 3D simulations (e.g., 0.4, 0.45, etc.)	Optional (default value for 1D and 2D is 0.8; default value for 3D is 0.4)
constants. RECONSTRUCT_ORDER or “reconstructOrder”	Integer value for which spatial reconstruction order PythonMHD should use before calculating intercell fluxes	Accepted values are 0 for no spatial reconstruction and 2 for second-order/PPM reconstruction (first-order/PLM reconstruction will be available in a future version of PythonMHD)	Optional (default value is 0 for no spatial reconstruction)
constants.ENTROPY_FIX or “entropyFix”	Boolean flag for whether an entropy fix should be applied to small wavespeeds when calculating intercell flux (sometimes helpful for smoothing out numerical instabilities; see textbook by Toro [1] for additional information)	True (if you want to apply an entropy fix) False (if you do not want to apply an entropy fix)	Optional (default is False, because most simulations in PythonMHD do not require an entropy fix to remain numerically stable)

constants.EPSILON or “epsilon”	Numeric value for the wavespeed threshold that should be used in a simulation with an entropy fix (i.e., all wavespeeds below this epsilon threshold will be changed by the entropy fix)	Any numeric value (int or float) greater than zero (but you will probably want to keep it well under 1, in order to avoid applying the entropy fix to most waves in your simulation)	Optional (default value is 0.5)
constants.MIN_DENSITY or “minDens”	Numeric value for the density minimum/floor in your simulation (all densities below this value will be set to the minimum density you define with this parameter)	Any numeric value (int or float) greater than zero (but you will want to keep it very small (~10^-9 or lower) so that you never change density values by a significant amount)	Optional (default is 10^-12)
constants.MIN_PRESSURE or “minPres”	Numeric value for the pressure minimum/floor in your simulation (all pressures below this value will be set to the minimum pressure you define with this parameter)	Any numeric value (int or float) greater than zero (but you will want to keep it very small (~10^-9 or lower) so that you never change pressure values by a significant amount)	Optional (default is 10^-12)
constants.MIN_ENERGY or “minEnergy”	Numeric value for the energy minimum/floor in your simulation (all energies below this value will be set to the minimum density you define with this parameter)	Any numeric value (int or float) greater than zero (but you will want to keep it very small (~10^-9 or lower) so that you never change energy values by a significant amount)	Optional (default is 10^-12)
constants.PLOT_DATA or “plotData”	Boolean flag for whether the gas state should be	True (if you want to visualize simulation data)	Optional (default is False)

	plotted/visualized during the simulation Visualization details are provided in the visualization parameters/visPar structure (see next user guide section)	False (if you don't want to visualize simulation data)	
constants.PLOT_DATA_Dt or "plotDt"	Numeric value for the amount of simulation time between visualization outputs (i.e., should data be plotted every 0.1 time units in the simulation, every 0.5, etc.)	Any numeric value (int or float) greater than zero	Required (if plotData is set to True)
constants.SAVE_FIGS or "saveFigs"	Boolean flag for whether the visualizations generated by the simulation should be saved on the user's computer	True (if you want to save the visualizations) False (if you don't want to save the visualizations)	Optional (default is False)
constants. SAVE_FIGS_FORMAT or "figFormat"	String for the image file format that should be used for saving PythonMHD visualizations	Any string (e.g., ".png" or "png", ".pdf") that corresponds to an image file type that is supported by your matplotlib backend (see matplotlib backend parameter below for more information) Note: You can use the provided IMAGE_FORMAT constants in PythonMHD if you don't want to look up the correct string for the	Optional (default is PNG files)

		file type that you want to use.	
constants. SAVE_FIGS_DPI or “saveFigsDPI”	Integer for the resolution of saved images in dots-per-inch/dpi	Any integer greater than zero	Optional (default is 300 dots per inch)
constants. MAKE_MOVIE or “makeMovie”	Boolean flag for whether you want to make a movie of your visualization outputs	True (if you want PythonMHD to make a movie with your output images) False (if you don’t want PythonMHD to make a movie of the visualizations)	Optional (default is False)
constants. MATPLOTLIB_BACKEND or “matplotlibBackend”	String for the matplotlib backend that should be used for PythonMHD visualizations	Any string for a matplotlib backend that is installed on your machine Note: This version of PythonMHD has only been tested on the TkAgg backend, so unexpected behaviours may occur if you pass a different backend for this parameter. In the future I will post information about how well PythonMHD visualizations work with other matplotlib backends.	Optional (default is “TkAgg”)
constants. SAVE_DATA or “saveData”	Boolean flag for whether PythonMHD should save numerical data (i.e., all of the gas variables at a particular time) during the simulation	True (if you want to save numerical data) False (if you don’t want to save numerical data)	Optional (default is False)
constants. SAVE_DATA_DT or “saveDt”	Numeric value for the amount of simulation time between	Any numeric value (int or float) greater than zero	Required (if saveData is set to True)

	numerical data saves (i.e., should data be saved every 0.1 time units in the simulation, every 0.5, etc.)		
constants.SAVE_DATA_FORMAT or “dataFormat”	String for the file type (.mat or .vtk) that should be used for saving numerical data	constants.DATA_FORMAT_MAT or “mat” for creating .mat files constants.DATA_FORMAT_VTK or “vtk” for creating .vtk files	Optional (default is .mat files)
constants.OUTPUT_FOLDER_PATH or “outputFolderPath”	String for the absolute path to the directory where you want to create your outputs folder (which will be where any images and data files are saved)	Any string for a path that exists on your computer	Optional (default is the Outputs folder in the PythonMHD directory)
constants.OUTPUT_FOLDER_NAME or “outputFolderName”	String for the name of the output folder where all images and data files will be saved	Any non-empty string with fewer than 50 characters	Optional (default folder name includes the date and time to help you identify the simulation)
constants.OUTPUT_FILE_NAME or “outputFileName”	String for the name that should be at the start of the file names for all images and data files that are saved during your simulation	Any non-empty string with fewer than 50 characters	Optional (default file name prefix is the name of the folder)

Figure 12 provides a screenshot of how the simPar structure looks in one of the sample PythonMHD initialization files.

```
###SIMULATION PARAMETERS###
simPar = {
            constants.IS_MHD:False, #boolean flag for whether the simulation has magnetic fields
            constants.GAMMA: 1.6666666667, #specific heat ratio for the ideal gas
            constants.TLIM: 1.0, #time at which the simulation should terminate
            constants.CFL: 0.8, #CFL number for calculating timestep sizes
            constants.MAX_CYCLES:10000, #max number of simulation cycles
            constants.RECONSTRUCT_ORDER:constants.NO_SPATIAL_RECONSTRUCTION, #spatial reconstruction order
            constants.ENTROPY_FIX:False, #boolean flag for whether we should apply an entropy fix to small wavespeeds
            constants.EPSILON:0.5, #smallest wavespeed that will not be affected by the entropy fix
            constants.PLOT_DATA:True, #boolean flag for whether we want to visualize the simulation data
            constants.PLOT_DATA_DT:0.1, #amount of simulation time between visualization outputs
            constants.SAVE_FIGS:False, #boolean flag for whether we should save the visualization outputs
            constants.SAVE_FIGS_FORMAT: constants.IMAGE_FORMAT_PNG, #file type for images of any visualization outputs
            constants.SAVE_FIGS_DPI: 300, #resolution in dots-per-inch/dpi for output images
            constants.MATPLOTLIB_BACKEND: constants.MATPLOTLIB_BACKEND_TKAGG, #matplotlib backend (for visualizations)
            constants.SAVE_DATA:False, #boolean flag for whether we should save the numerical data for the simulation
            constants.SAVE_DATA_DT:0.05, #amount of simulation time between numerical data saves
            constants.SAVE_DATA_FORMAT: constants.DATA_FORMAT_MAT, #file type for numerical data saves
            constants.OUTPUT_FOLDER_NAME: "HydroBlast_2D", #name for the outputs folder
            constants.OUTPUT_FILE_NAME: "HydroBlast_2D", #prefix name for the output files
        }
```

Figure 12: Screenshot of the grid parameters/simPar structure for a PythonMHD simulation.

Visualization Parameters

If you want PythonMHD to create visualizations of your simulation data, you need to define the visualization parameters in another Python dictionary, which PythonMHD refers to as visPar. An example of visPar for a 3D simulation is shown in Figure 13.

The first parameter (constants.FIGURES) is an array that contains the dictionaries for each matplotlib figure that you want to generate. The Figure 13 example only has one figure dictionary in this array, which means that only one figure will be generated. In the figure dictionary, the constants.PLOTS parameter is an array of the dictionaries for all of the data plots that you want to display on this figure. PythonMHD will let you have a maximum of nine plots on a single matplotlib figure. In each plot dictionary, you can define the gas variable that should be visualized, the color or colormap that should be used, and the type of plot. In PythonMHD, you can visualize simulation data with either a 1D line plot, a 2D colormap, or a 3D transparent

scatter plot. If your simulation is 1D, you can only visualize data with line plots. If the simulation is 2D, the plot type can be either a full 2D colormap of the simulation area or a 1D line plot of a particular line through the simulation (e.g., the horizontal line y = 0, the vertical line x = 1, etc.). If the simulation is 3D, you can choose between a 3D transparent scatter plot of the entire simulation volume, a 2D colormap of a particular plane in the simulation grid, or a 1D plot of a line through the 3D space.

```
###VISUALIZATION PARAMETERS###
visPar = {
            constants.FIGURES: #array of Figure structures
            [
                #Figure 1
                {
                    constants.FIGURE_TITLE: "3D Hydro Blast", #title for the figure
                    constants.PLOTS: #array of plots to display on this figure
                     [
                        {
                            constants.PLOT_VAR:constants.DENSITY #gas variable we want to plot (density)
                        },
                        {
                            constants.PLOT_VAR:constants.PRESSURE #gas variable we want to plot (pressure)
                        },
                        {
                             constants.PLOT_VAR:constants.DENSITY, #gas variable we want to plot (density)
                             constants.PLOT_TYPE: constants.PLOT_TYPE_2D, #type of plot (2D colormap)
                             constants.PLOT_PLANE: constants.XY_PLANE, #plane that is parallel with our
                                                                       #2D cross-section (xy-plane)
                             constants.Z_POS: 0, #z-position of the cross-section
                        },
                        {
                             constants.PLOT_VAR:constants.PRESSURE, #gas variable we want to plot (pressure)
                             constants.PLOT_TYPE: constants.PLOT_TYPE_1D, #type of plot (1D line plot)
                             constants.PLOT_AXIS: constants.Z_AXIS, #axis that is parallel with our line (z-axis)
                             constants.X_POS: 0, #x-position of the line
                             constants.Y_POS: 0, #y-position of the line
                        }
                    ]
                }
            ]
        }
```

Figure 13: Screenshot of the visualization parameters/visPar structure for a 3D PythonMHD simulation.

The table below describes all of the figure and plot parameters that you can set in visPar.

PythonMHD Visualization Parameters (visPar)			
Parameter Name	Purpose	Acceptable Inputs	Required or Optional
constants. FIGURES or “figures” (visPar level parameter)	Array of dictionaries (one for each matplotlib figure	A list/array with at least one figure dictionary	Required

	that you want to create)		
constants. FIGURE_TITLE or “figureTitle” (figure level parameter)	String for the title that you want to display on the matplotlib figure	Any string with fewer than 50 characters	Optional (default title is Figure #x, where x is the figure’s position in the figures array)
constants.SAVE_FIGS or “saveFigs” (figure level parameter)	Boolean flag for whether images of this particular matplotlib figure should be saved (overrides the simPar value of this parameter)	True (if you want to save the visualizations generated by this figure) False (if you don’t want to save the visualizations on this figure)	Optional (default is the value of saveFigs in simPar (or False, if saveFigs isn’t in simPar))
constants. SAVE_FIGS_FORMAT or “figFormat” (figure level parameter)	String for the image file format that should be used for saving images of this particular figure	Any string (e.g., “.png” or “png”, “.pdf”) that corresponds to an image file type that is supported by your matplotlib backend (see matplotlib backend parameter below for more information) Note: You can use the provided IMAGE_FORMAT constants in PythonMHD if you don’t want to look up the correct string for the file type that you want to use.	Optional (default is is the value of figFormat in simPar (or False, if figFormat isn’t in simPar)
constants. SAVE_FIGS_DPI or “saveFigsDPI” (figure level parameter)	Integer for the resolution of saved images in dots-per-inch/dpi for this particular figure	Any integer greater than zero	Optional (default is the value of saveFigsDPI in simPar (or 300 dots per inch, if saveFigsDPI isn’t in simPar)

constants. MAKE_MOVIE or “makeMovie” (figure level parameter)	Boolean flag for whether you want to make a movie of your visualization outputs from this particular figure	True (if you want PythonMHD to make a movie with your output images from this figure) False (if you don’t want PythonMHD to make a movie of the visualizations from this figure)	Optional (default is the value of makeMovies in simper (or False, if makeMovies isn’t in simPar)
constants. NUM_ROWS or “numRowsInFigure” (figure level parameter)	Integer for the number of rows that you want in your figure (useful for when you have multiple data plots in one figure)	Any integer >= 1 and <= 9 (because no more than 9 plots can be displayed in one figure)	Optional (default is calculated based on the number of plots in the figure)
constants. NUM_COLS or “numColsInFigure” (figure level parameter)	Integer for the number of columns that you want in your figure (useful for when you have multiple data plots in one figure)	Any integer >= 1 and <= 9 (because no more than 9 plots can be displayed in one figure)	Optional (default is calculated based on the number of plots in the figure)
constants. PLOTS or “plots” (figure level parameter)	Array of dictionaries (one for each data plot that you want to display on the figure)	A list/array with at least one data plot dictionary	Required
constants. PLOT_VAR or “plotVar” (plot level parameter)	String for the gas variable that should be visualized on the plot	Any string that corresponds to a variable that PythonMHD can plot (density, x-velocity, y-velocity, z-velocity, total velocity, total velocity squared, pressure, and energy for hydrodynamics) (density, x-velocity, y-velocity, z-velocity, total velocity, total velocity	Optional (default is density)

		squared, Bx, By, Bz, B, B squared, pressure, total pressure (magnetic + hydrodynamic), and energy for magnetohydrodynamics) Note: You can use the provided DENSITY, VX, VY, VZ, V, V2, PRESSURE, ENERGY, BX, BY, BZ, B, B2, TOTAL_PRESSURE constants in PythonMHD to ensure that you pass the correct string for each variable.	
constants. PLOT_TYPE or "plotType" (plot level parameter)	String for the type of plot that should be generated	Any string for a plot type that is supported in PythonMHD ("1DPlot" for 1D line plots, "2DPlot" for 2D colormaps, "3DPlot" for 3D transparent scatter plots) Note: You can use the provided PLOT_TYPE_1D, PLOT_TYPE_2D, and PLOT_TYPE_3D constants in PythonMHD to ensure that you pass the correct string for each plot type.	Optional (default is the plot type that corresponds to the number of dimensions in your simulation (i.e., 1D line plots for 1D simulations, 2D colormaps for 2D simulations, and 3D transparent scatter plots for 3D simulations)
constants. PLOT_COLOR or "plotColor" (plot level parameter)	String for the color or colormap that should be used for the plot	Any string that is a recognized color or colormap in matplotlib	Optional (default is blue for 1D line plots, viridis for 2D colormaps, and blues for

			3D transparent scatter plots)
constants. PLOT_ALPHA or “plotAlpha” (plot level parameter)	Numeric value for the transparency of 3D scatter plots	Any float greater than zero and less than one (e.g., 0.001, 0.01, etc.) (You will typically want a small alpha value, in order to see through the entire simulation volume)	Optional (default is 0.005)
constants. MIN_PLOT_VAL or “minPlotVal” (plot level parameter)	Numeric value for the minimum/floor value that should be used in the plot’s colormap	Any numeric value (int or float) that is less than the MAX_PLOT_VAL parameter below (if MAX_PLOT_VAL is in your plot dictionary)	Optional (default is the smallest value in the data we are plotting)
constants. MAX_PLOT_VAL or “maxPlotVal” (plot level parameter)	Numeric value for the maximum/floor value that should be used in the plot’s colormap	Any numeric value (int or float) that is more than the MIN_PLOT_VAL parameter below (if MIN_PLOT_VAL is in your plot dictionary)	Optional (default is the largest value in the data we are plotting)
constants. PLOT_PLANE or “plane” (plot level parameter)	String for the plane that is parallel with the 2D cross-section that you want to plot in a 3D simulations	constants.XY_PLANE (if you want the plane to be the xy-plane) constants.XZ_PLANE (if you want the plane to be the xz-plane) constants.YZ_PLANE (if you want the plane to be the yz-plane)	Required if you are making a 2D plot of a 3D simulation
constants. PLOT_AXIS or “axis” (plot level parameter)	String for the axis that is parallel with the 1D line of data that you want to plot in a 3D simulation	constants.X_AXIS (if you want the axis to be the x-axis) constants.Y_AXIS (if you want the axis to be the y-axis) constants.Z_AXIS (if you want the axis to be the z-axis)	Required if you are making a 1D plot of a 2D or 3D simulation
constants. PLOT_X_POS or “xPos” (plot level parameter)	Numeric value for the x-position of the plane or line of	Any numeric value (int or float) for an x-position that falls within your simulation grid	Required if you are plotting a yz-plane of a 3D

	data you want to plot		simulation, if you are plotting a y-axis line in a 2D simulation, or if you are plotting a y-axis or z-axis line in a 3D simulation
constants. PLOT_Y_POS or “yPos” (plot level parameter)	Numeric value for the y-position of the plane or line of data you want to plot	Any numeric value (int or float) for a y-position that falls within your simulation grid	Required if you are plotting an xz-plane of a 3D simulation, if you are plotting an x-axis line in a 2D simulation, or if you are plotting a x-axis or z-axis line in a 3D simulation
constants. PLOT_Z_POS or “zPos” (plot level parameter)	Numeric value for the z-position of the plane or line of data you want to plot	Any numeric value (int or float) for a z-position that falls within your simulation grid	Required if you are plotting an xy-plane, an x-axis line, or a y-axis line in a 3D simulation

Initial Conditions

Next we need to set the initial conditions for the PythonMHD simulation. To make your gas system as flexible as possible, you will probably want to define another dictionary where you can set some parameters for the initial state of the system. Figure 14 shows the initial conditions

parameter structure that is used for the 3D MHD blast problem in the MHDBlast_3D_InitializationFile.py sample initialization script.

```
###MHD BLAST PARAMETERS###
#This parameter structure allow you to customize
#the ambient pressure, blast pressure, ambient density,
#blast density, magnetic field strength, magnetic field direction,
#and the initial size of the over-pressurized region.
blastPar = {
            "ambPres":0.1, #initial pressure in the ambient medium/area outside the blast
            "presRatio":100.0, #initial pressure ratio between the blast and the ambient medium
                               #(blast pressure/ambient pressure)
            "ambDens":1.0, #initial density in the ambient medium/area outside the blast
            "densRatio":1.0, #initial density ratio between the blast and the ambient medium
                             #(blast density/ambient density)
            "blastRadius":0.1, #initial size of the blast/over-pressurized region
            "B0": 1.0, #initial magnetic field strength
            "theta": m.pi/4.0, #initial angle of the magnetic field with respect to the x-axis (in radians)
           }
```

Figure 14: Screenshot of the initial conditions parameter structure for a 3D MHD blast simulation.

Below this blast dictionary, we have the function that will build the gas variable matrix that we will pass to PythonMHD as the initial state of the gas system. PythonMHD expects to receive the initial state of the system as a numpy matrix, which we will call primVars. We choose the name primVars because we are defining the initial primitive variables (density, velocities, magnetic field components, and pressure) for the simulation. In PythonMHD, for hydrodynamics, primVars[0] should contain your initial density values for the simulation; primVars[1], the initial x-velocities; primVars[2], the initial y-velocities; primVars[3], the initial z-velocities; and primVars[4], the initial pressures for every cell in the simulation grid. For MHD simulations, primVars[0] again stores the density values; primVars[1], the initial x-velocities; primVars[2], the initial y-velocities; primVars[3], the initial z-velocities; primVars[4], the initial x-components of the magnetic field (Bx); primVars[5], the initial y-components of the magnetic field (By); primVars[6], the initial z-components of the magnetic field (Bz); and primVars[7], the pressures for every cell in the simulation grid. For hydrodynamics and MHD, all of these

parts of primVars should have the same number of values. In 1D, primVars[var][x] is the value of the gas variable var at a cell #x in the x-direction. In 2D, primVars[var][y][x] is the value of the gas variable var at the cell with index y in the y-direction and index x in the x-direction. In 3D, primVars[var][y][x][z] is the value of the gas variable var at the cell with index y in the y-direction, index x in the x-direction, and index z in the z-direction.

Most of the time, how you set the primitive variable initial conditions will vary across the simulation grid. Figure 15 shows the start of the primitive variable function for the 3D MHD blast problem. The line (xCoords, yCoords, zCoords) = initHelper.createCoordinates3D_Cartesian(gridPar) is how you get the (x, y, z) coordinates that correspond to the values that you passed in your gridPar structure. You can then use these values to set different densities, pressures, velocities, etc. at different locations in your simulation.

```
######PRIMITIVE VARIABLE FUNCTION######

#Function: getBlastPrimVars
#Purpose: Creates the primitive variable matrix for the 3D MHD blast,
#         using the parameters in gridPar and blastPar.
#Input Parameters: gridPar (the grid parameters structure)
#                  blastPar (the Sod Shock Tube parameters structure)
#Outputs: primVars (the primitive variable matrix that has been
#                   constructed based on the specifications in
#                   gridPar and blastPar)
def getBlastPrimVars(gridPar, blastPar):
    #Get the number of cells in the x-direction
    numXCells = gridPar[constants.NUM_X_CELLS]
    assert(numXCells > 0)
    #Get the number of cells in the y-direction
    numYCells = gridPar[constants.NUM_Y_CELLS]
    assert (numYCells > 0)
    #Get the number of cells in the z-direction
    numZCells = gridPar[constants.NUM_Z_CELLS]
    assert(numZCells > 0)
    #Calculate the width of each grid cell
    dz = (gridPar["maxZ"] - gridPar["minZ"])/numZCells
    #Find the x-, y-, and z-coordinates for every cell in the simulation grid
    (xCoords, yCoords, zCoords) = initHelper.createCoordinates3D_Cartesian(gridPar)
    #Calculate each cell's distance from the centre of the simulation grid (at coordinates (x = 0, y = 0, z = 0)
    radii = np.sqrt(xCoords*xCoords + yCoords*yCoords + zCoords*zCoords)
```

Figure 15: Screenshot of the primitive variable function that creates the initial conditions for a 3D MHD blast problem.

Figure 16 shows the part of the primitive variables function where we construct the primVars matrix.

```
    blastRadius = blastPar["blastRadius"]
    ambPres = blastPar["ambPres"]
    presRatio = blastPar["presRatio"]
    ambDens = blastPar["ambDens"]
    densRatio = blastPar["densRatio"]

    #Create the density values
    rho = ambDens*np.ones(shape=(numYCells,numXCells,numZCells))
    rho[radii < blastRadius] = densRatio*ambDens
    #Create the x-velocities
    vx = np.zeros(shape=(numYCells,numXCells,numZCells))
    #Set the y-velocities to zero
    vy = np.zeros(shape=(numYCells,numXCells,numZCells))
    #Set the z-velocities to zero
    vz = np.zeros(shape=(numYCells,numXCells,numZCells))

    Bx = blastPar["B0"]*np.cos(blastPar["theta"])*np.ones(shape=(numYCells,numXCells,numZCells))
    By = blastPar["B0"]*np.sin(blastPar["theta"])*np.ones(shape=(numYCells,numXCells,numZCells))
    Bz = np.zeros(shape=(numYCells,numXCells,numZCells))

    #Create the pressure values
    pres = ambPres*np.ones(shape=(numYCells,numXCells,numZCells))
    pres[radii < blastRadius] = presRatio*ambPres
    #Create the primitive variables matrix,
    #which will contain the five hydrodynamic
    #primitive variables in the following
    #order: density, x-velocity, y-velocity,
    #       z-velocity, and pressure
    primVars = np.zeros(shape=(8,numYCells,numXCells,numZCells))
    primVars[0,:] = rho
    primVars[1,:] = vx
    primVars[2,:] = vy
    primVars[3,:] = vz
    primVars[4,:] = Bx
    primVars[5,:] = By
    primVars[6,:] = Bz
    primVars[7,:] = pres
    #Return the primitive variables matrix
    return primVars
```

Figure 16: Screenshot of the setting the initial conditions in the 3D MHD blast primitive variable function.

See other sample PythonMHD initialization files for additional examples of how you can build your primitive variables matrix.

Running the Simulation

At the bottom of the initialization file, as shown in Figure 17, we have the main script where we call the function that builds the primitive variable matrix (blastPrimVars = getBlastPrimVars(gridPar, blastPar)), creates an instance of the PythonMHD Simulation class (blastSim = Simulation(simPar, gridPar, blastPrimVars, visPar)), and calls the run function on the Simulation object that we just created (newGrid = blastSim.run()). The last line is the one that starts your PythonMHD simulation. When the simulation ends, the .run() function will return a new instance of the SimulationGrid class (defined in the SimulationGrid.py PythonMHD source file) that contains the final state of your gas system. If you told PythonMHD to save visualizations or numerical data, all of the created files will be saved on your computer by the time that PythonMHD returns the simulation grid, so you can feel safe terminating the program at this point.

```
######MAIN SCRIPT######

#Build the primitive variables matrix for
#the initial state of the 3D MHD Blast
blastPrimVars = getBlastPrimVars(gridPar, blastPar)

#Create the Simulation object
blastSim = Simulation(simPar,gridPar,blastPrimVars,visPar)

#Run the simulation
newGrid = blastSim.run()
```

Figure 17: Screenshot of the main script at the bottom of the initialization file, where we call the function that builds our primitive variable matrix, create an instant of the PythonMHD Simulation class, and call the run function on our new Simulation object.

Built-In Test Problems

In this part of the user guide, I give an overview of the built-in simulations that you can run as soon as you finish setting up PythonMHD on your machine. For each test problem, I provide visualization outputs so you can confirm that your PythonMHD simulations return the same results.

(Note: The following sections include a substantial amount of content from my master's thesis [2], which is available at https://mspace.lib.umanitoba.ca/items/7b23c9bc-a038-4f2b-a616-ce6393afed8b.)

Sod's Shock Tube (1D Hydrodynamics)

(The script for this test problem is called SodShockTube_1D_InitializationFile.py.)

Sod's shock tube [3] is a fundamental problem in 1D hydrodynamics, because it tests how well a simulation code models the propagation of shock and rarefaction waves in a hydrodynamic system. In Sod's shock tube, we initially have two uniform regions that are separated by a diaphragm that is removed at time t = 0. In the left chamber of the shock tube, we have a density of ρ_L= 1.0, an x-velocity of $v_{x,L}$ = 0.75, and a pressure of p_L= 1.0. In the right chamber, we have a density of ρ_R= 0.125, an x-velocity of $v_{x,R}$ = 0.0, and a pressure of p_R = 0.1. Across the entire one-dimensional system, we treat the gas as ideal with an adiabatic equation of state and a uniform specific heat ratio of γ = 1.4. When we remove the diaphragm, a shock wave travels from the high-pressure, high-density left chamber into the low-pressure, low-density right chamber. While the shock wave travels to the right, a rarefaction wave expands into the left chamber. Between the shock and rarefaction waves, we have a contact discontinuity (also known as an entropy wave) that travels to the right at a slower speed than the shock front. These waves create distinct five regions, as shown in Figure 18.

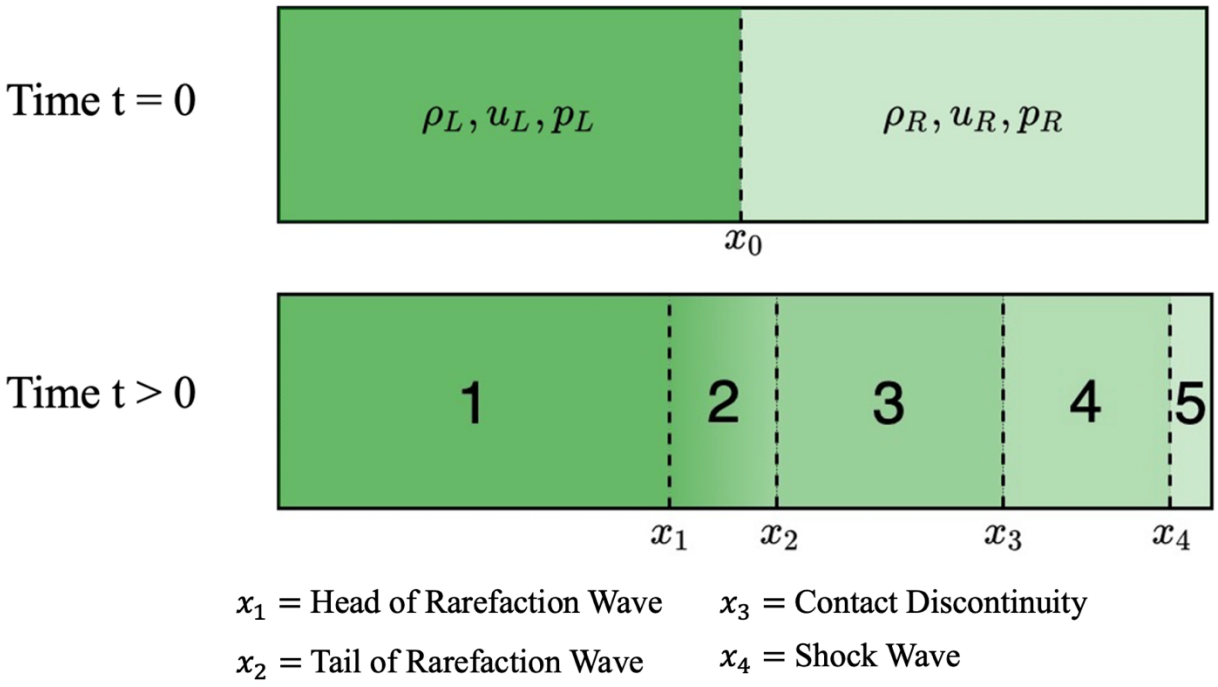

Figure 18: Regions in the Exact Solution of Sod's Shock Tube. At time t = 0.0, there are only two distinct regions. After removing the diaphragm, five distinct regions emerge, with their own values for gas density, velocity, and hydrodynamic pressure.

We can see these regions emerge in a PythonMHD simulation of Sod's shock tube, as shown in Figures 19 and 20. (Note: In order to match the gas system that is used for Sod's shock tube in the 2017 version of Athena [4,5], this PythonMHD simulation sets the left chamber x-velocity to zero instead of 0.75.)

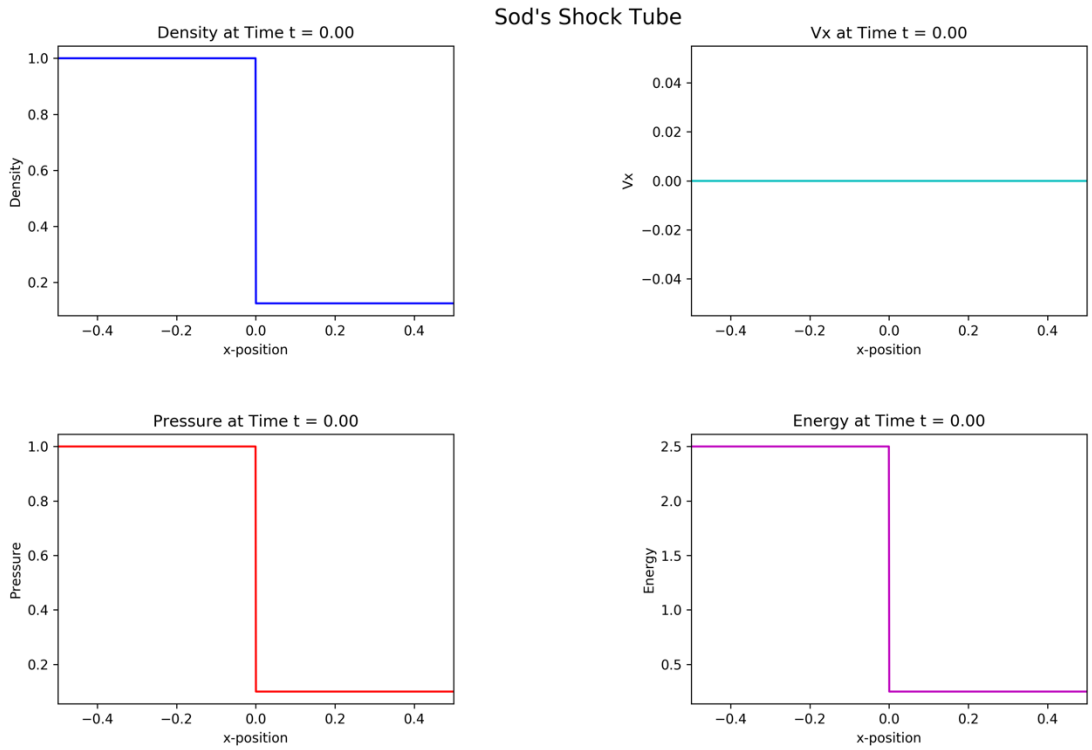

Figure 19: PythonMHD visualization of Sod's shock tube at time t = 0.

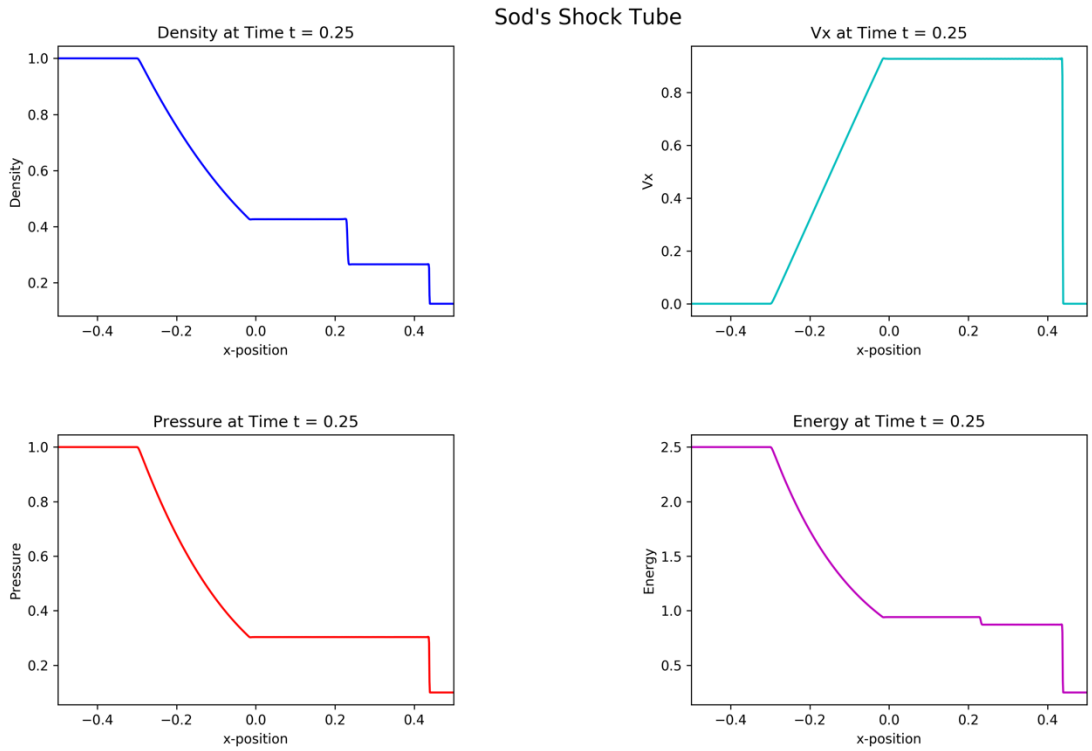

Figure 20: PythonMHD visualization of Sod's shock tube at time t = 0.25.

2D Hydro Blast (2D Hydrodynamics)

(The script for this test problem is called HydroBlast_2D_InitializationFile.py.)

For 2D hydrodynamics, the test problem is the hydrodynamic version of the MHD blast in [6]. The initialization script for this simulation is designed to match the gas system that is created for the 2D hydrodynamic blast in the 2017 version of Athena [4, 5]. At the centre of the simulation grid, we have a disk of gas with a uniform pressure that is 100 times greater than the pressure in the surrounding area. For this version of the blast problem, the gas density is initially uniform at dens = 1 everywhere in the simulation. In order to generate the results shown below in Figures 21-24, the pressure in the ambient medium (i.e., outside the high-pressure disk) is initially 0.1 and the initial radius of the blast disk is 0.1.

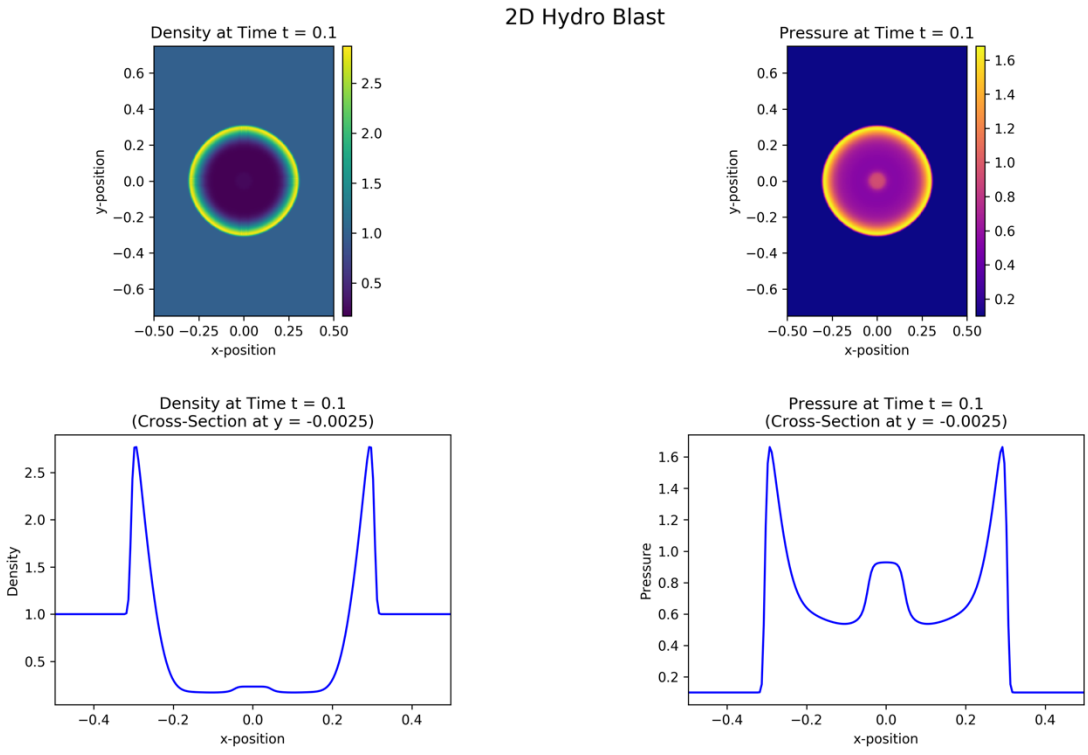

Figure 21: PythonMHD visualization of the 2D hydrodynamic blast at time t = 0.1.

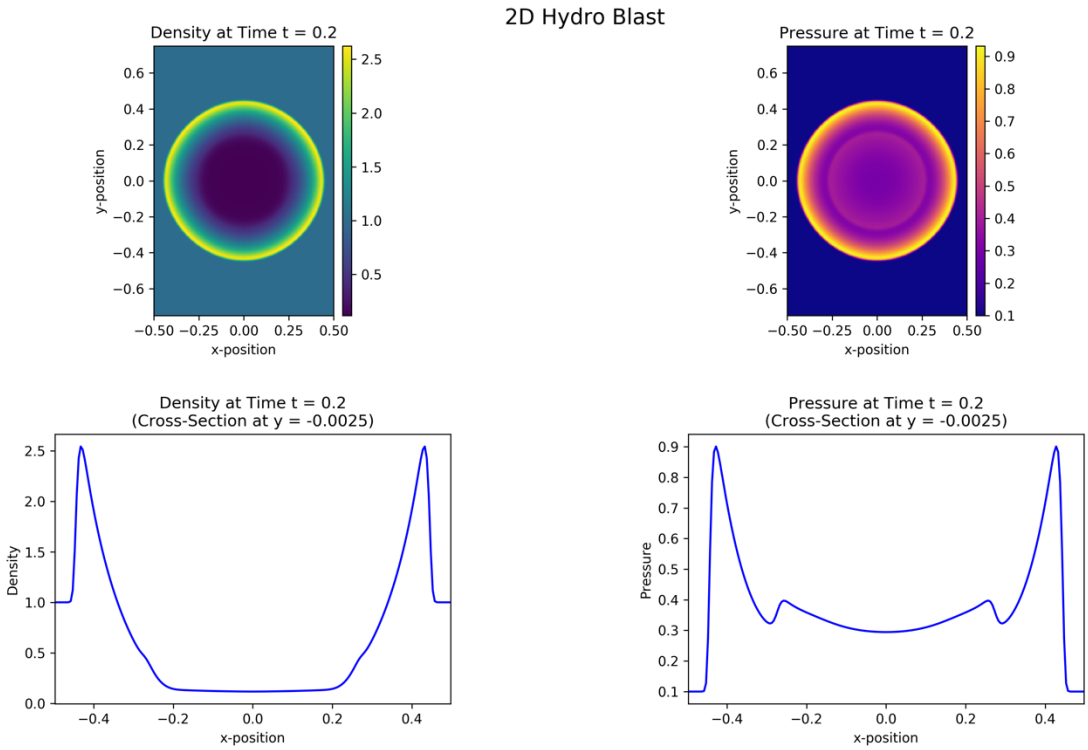

Figure 22: PythonMHD visualization of the 2D hydrodynamic blast at time t = 0.2.

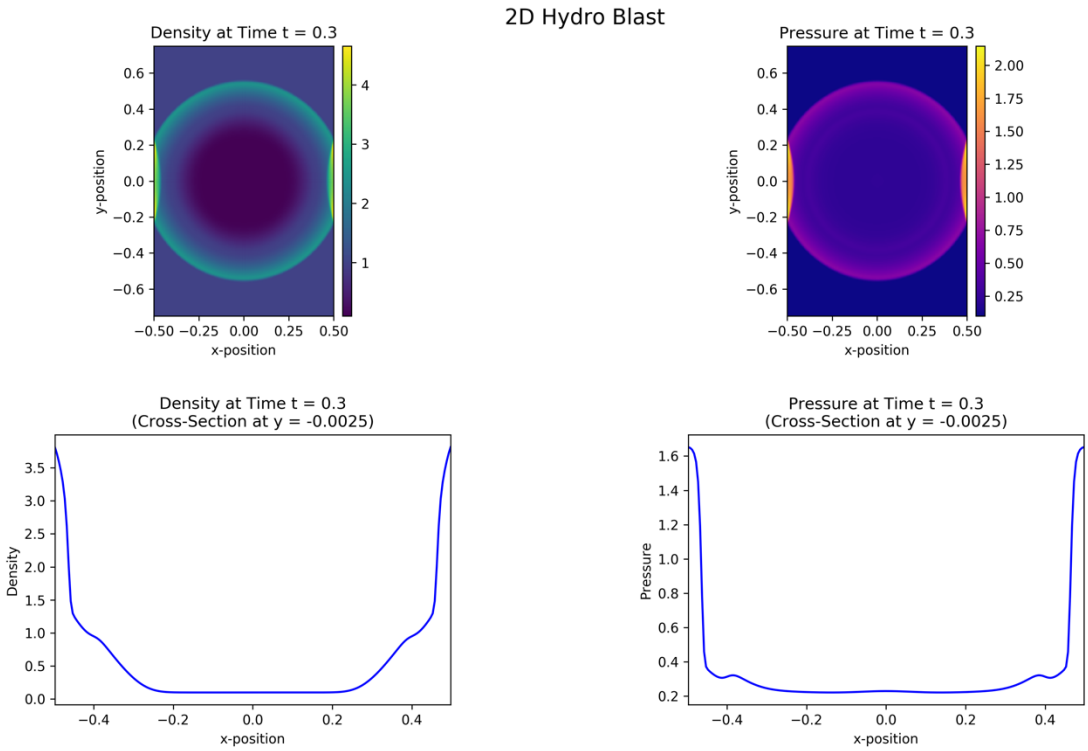

Figure 23: PythonMHD visualization of the 2D hydrodynamic blast at time t = 0.3.

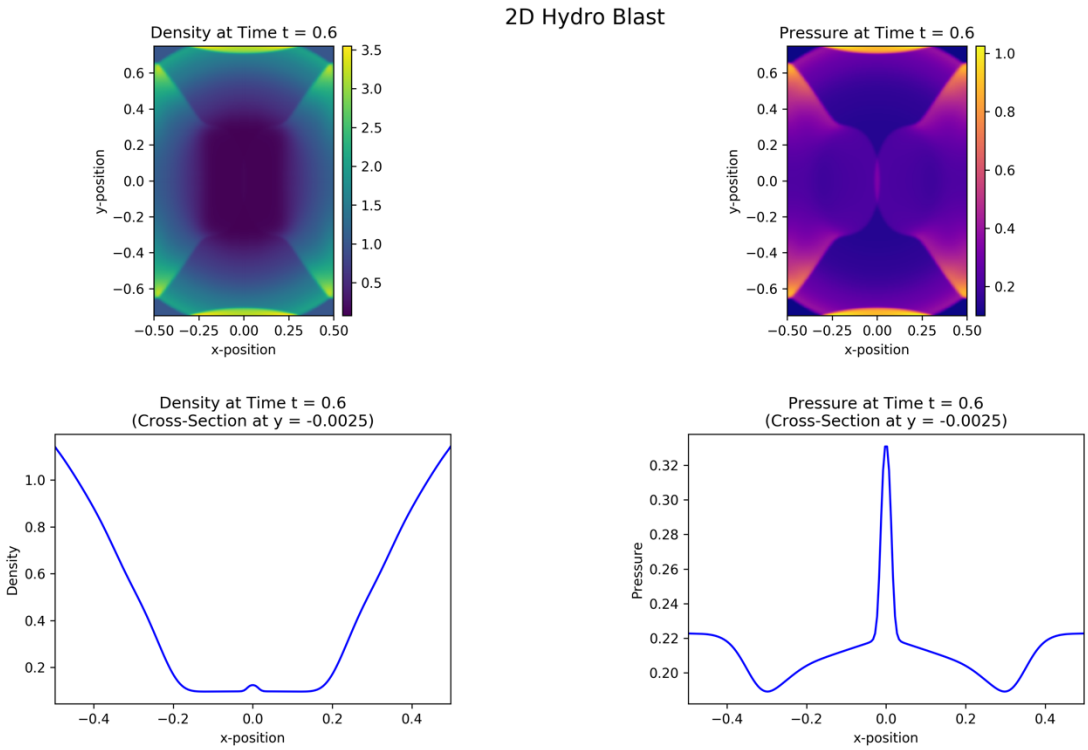

Figure 24: PythonMHD visualization of the 2D hydrodynamic blast at time t = 0.6.

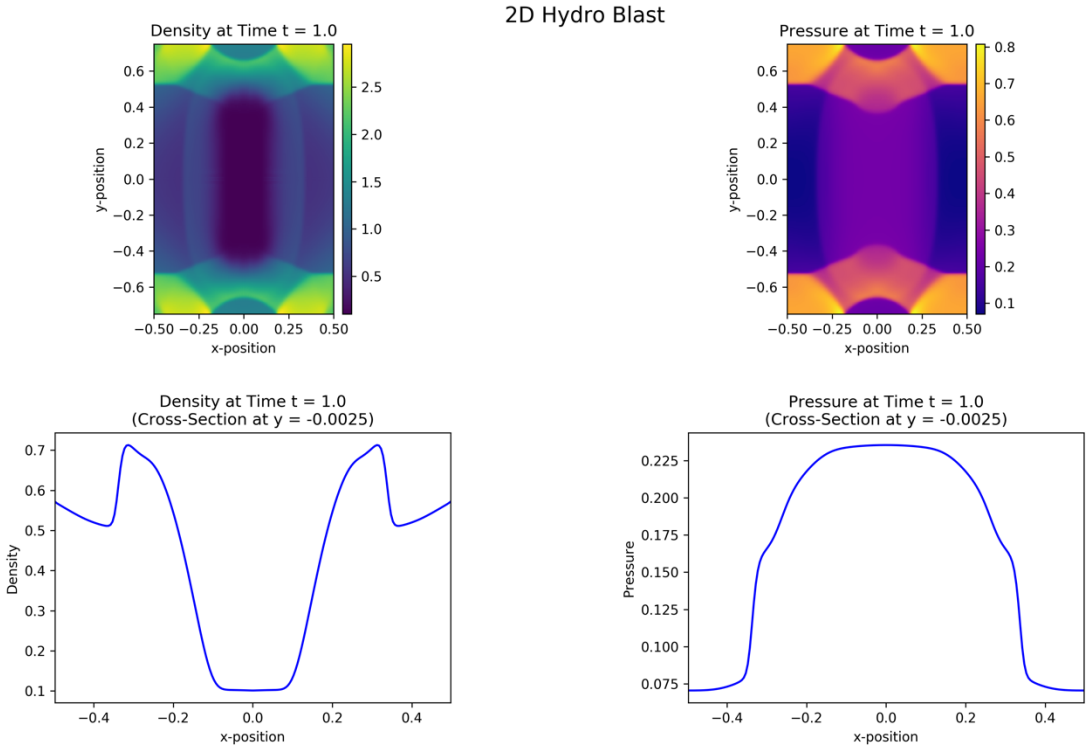

Figure 24: PythonMHD visualization of the 2D hydrodynamic blast at time t = 0.9.

3D Hydro Blast (3D Hydrodynamics)

(The script for this test problem is called HydroBlast_3D_InitializationFile.py.)

For 3D hydrodynamics, the test problem is the 3D version of the 2D hydrodynamic blast. At the centre of the simulation grid, we now have a sphere of gas with a uniform pressure that is 100 times greater than the pressure in the surrounding area. We again have a uniform density of dens = 1 everywhere in the simulation grid, an ambient pressure of 0.1, and a blast radius of 0.1. PythonMHD outputs for this problem are shown in Figures 25-32.

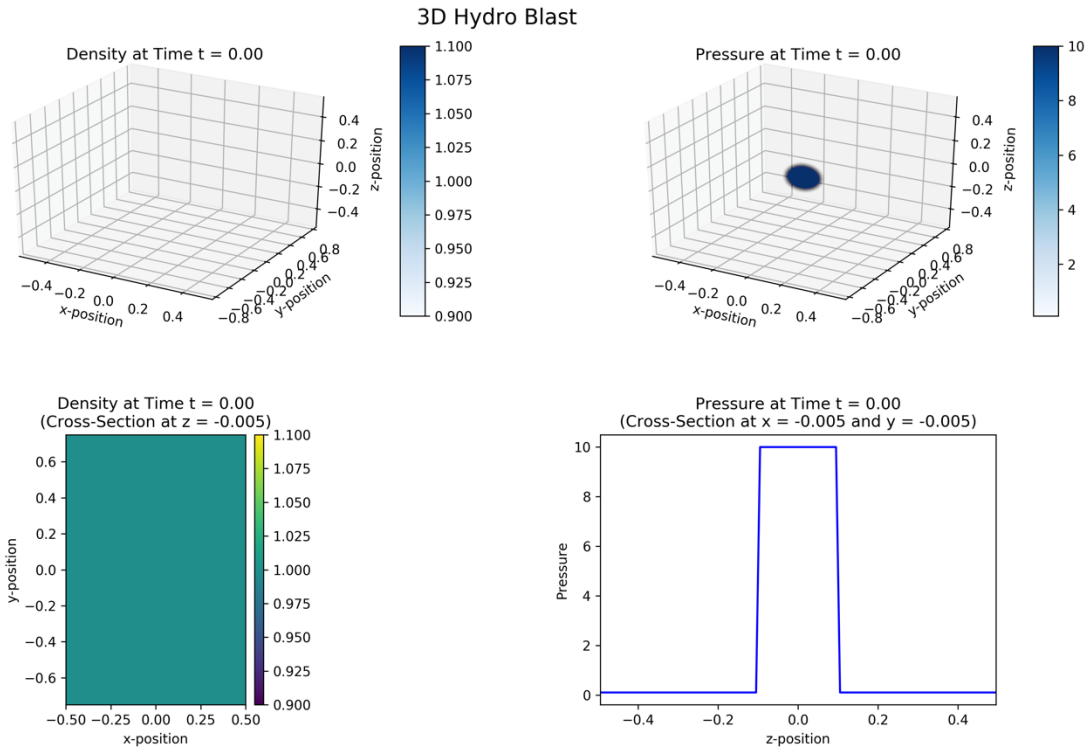

Figure 25: PythonMHD visualization of the 3D hydrodynamic blast at time t = 0.0.

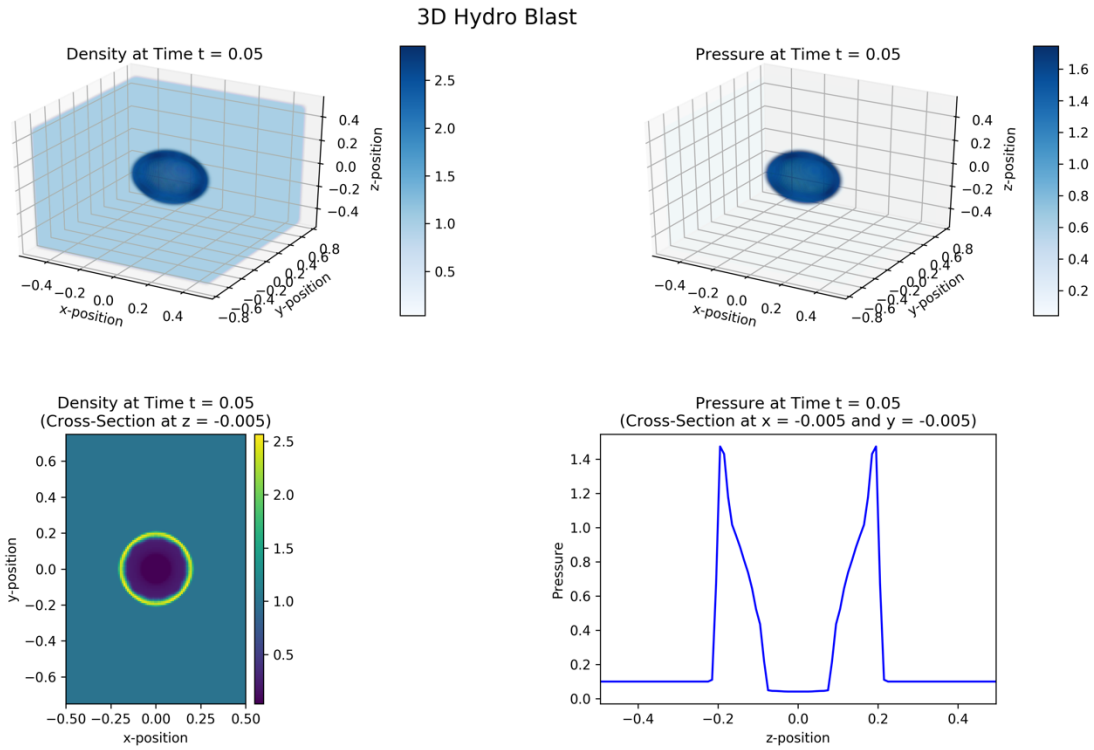

Figure 26: PythonMHD visualization of the 3D hydrodynamic blast at time t = 0.05.

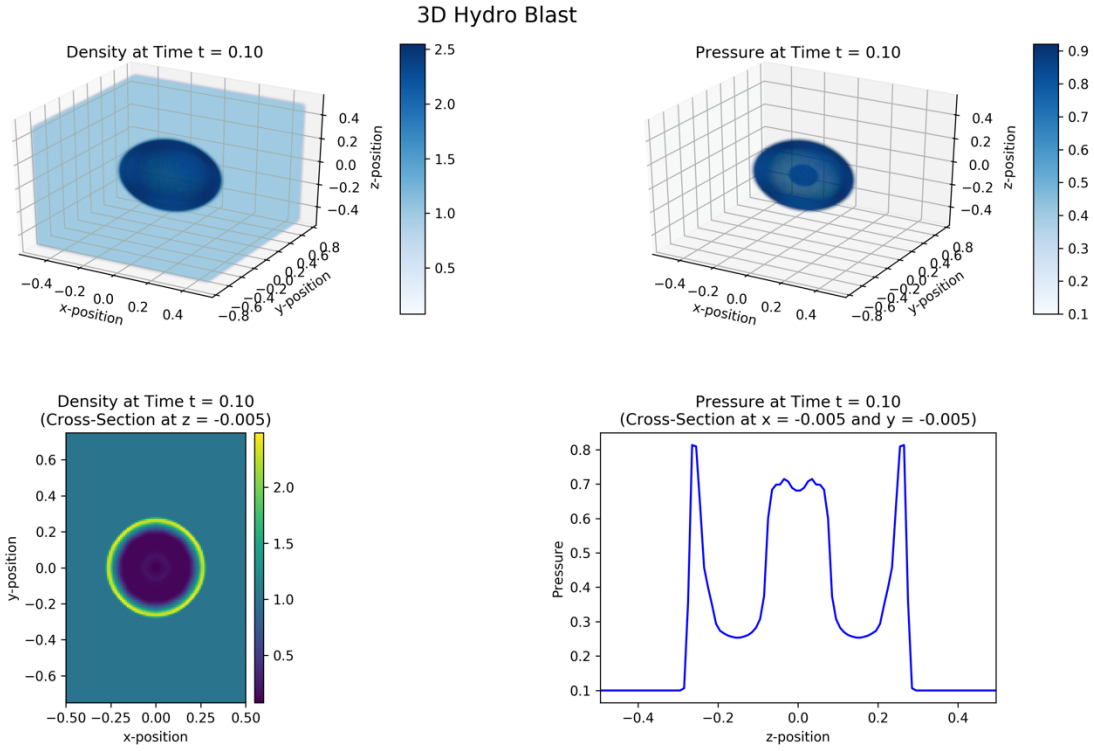

Figure 27: PythonMHD visualization of the 3D hydrodynamic blast at time t = 0.1.

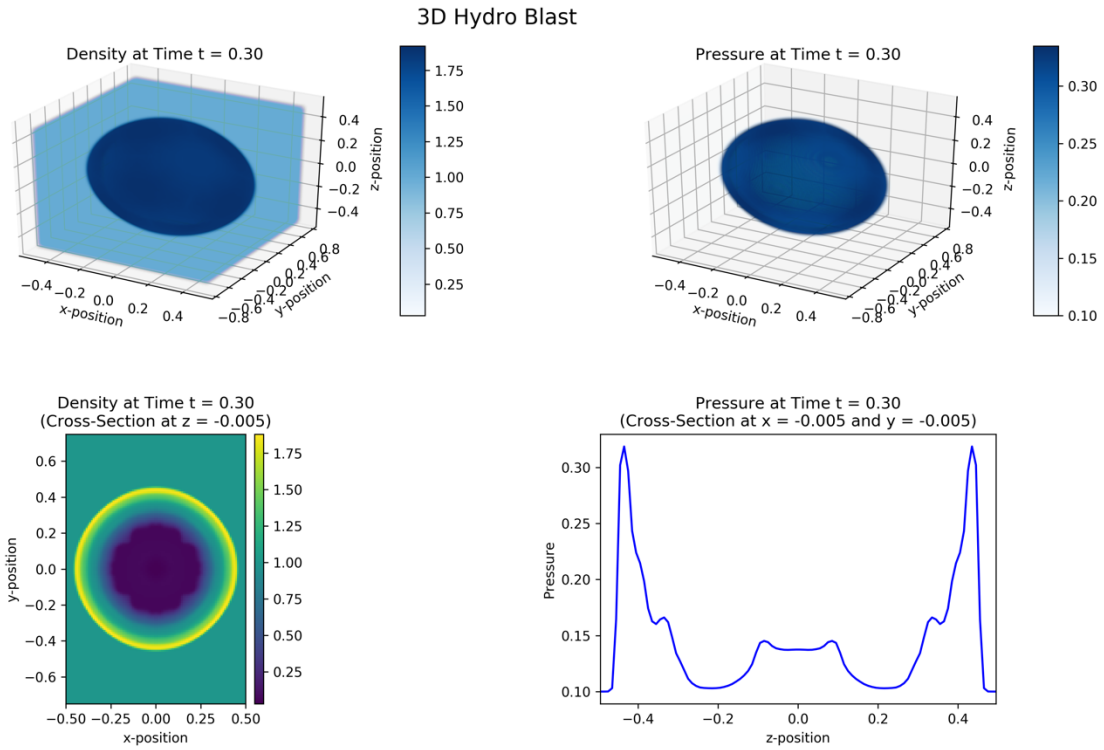

Figure 28: PythonMHD visualization of the 3D hydrodynamic blast at time t = 0.3.

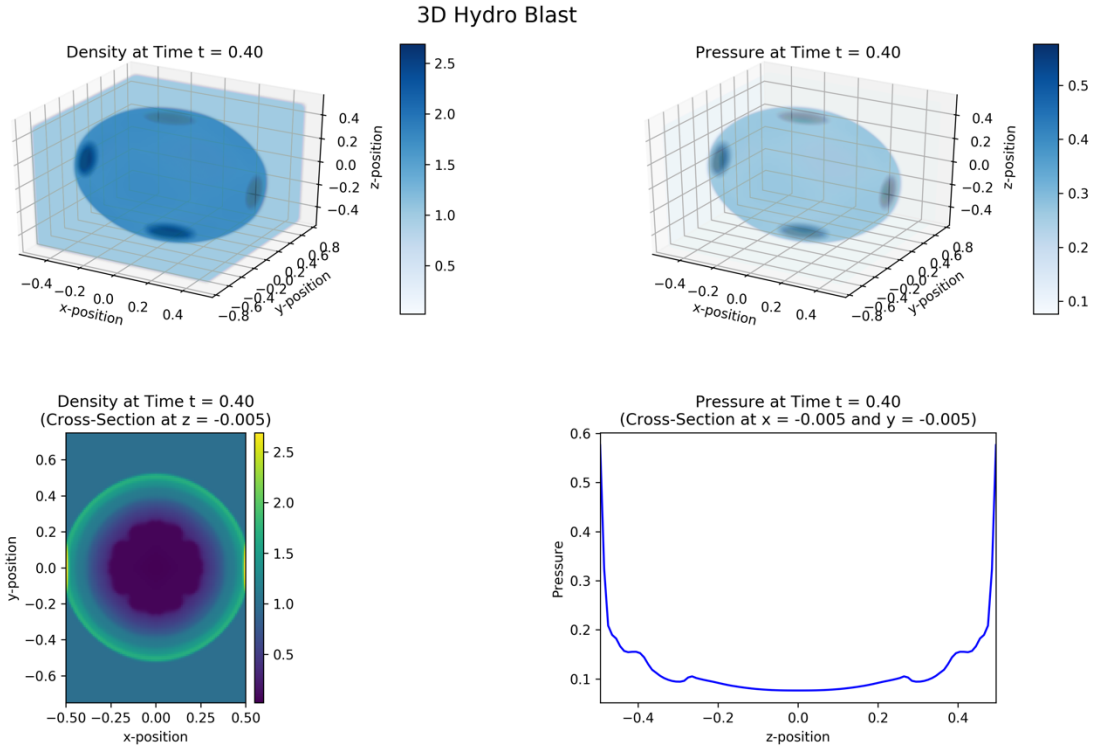

Figure 29: PythonMHD visualization of the 3D hydrodynamic blast at time t = 0.4.

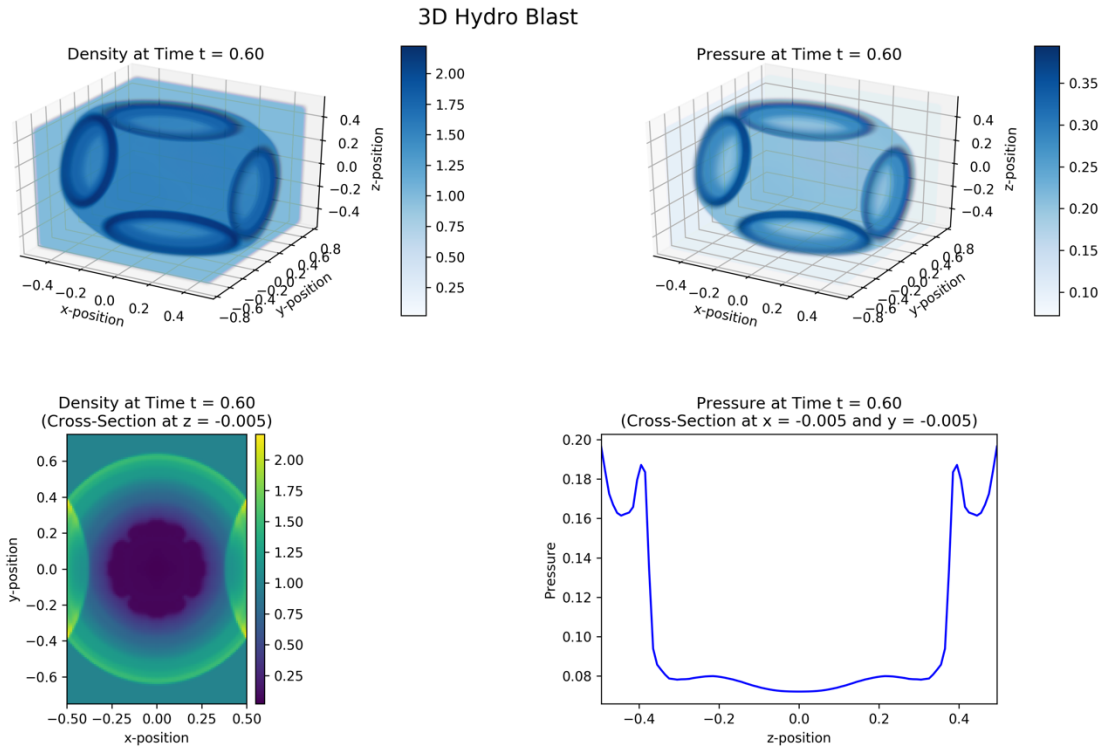

Figure 30: PythonMHD visualization of the 3D hydrodynamic blast at time t = 0.6.

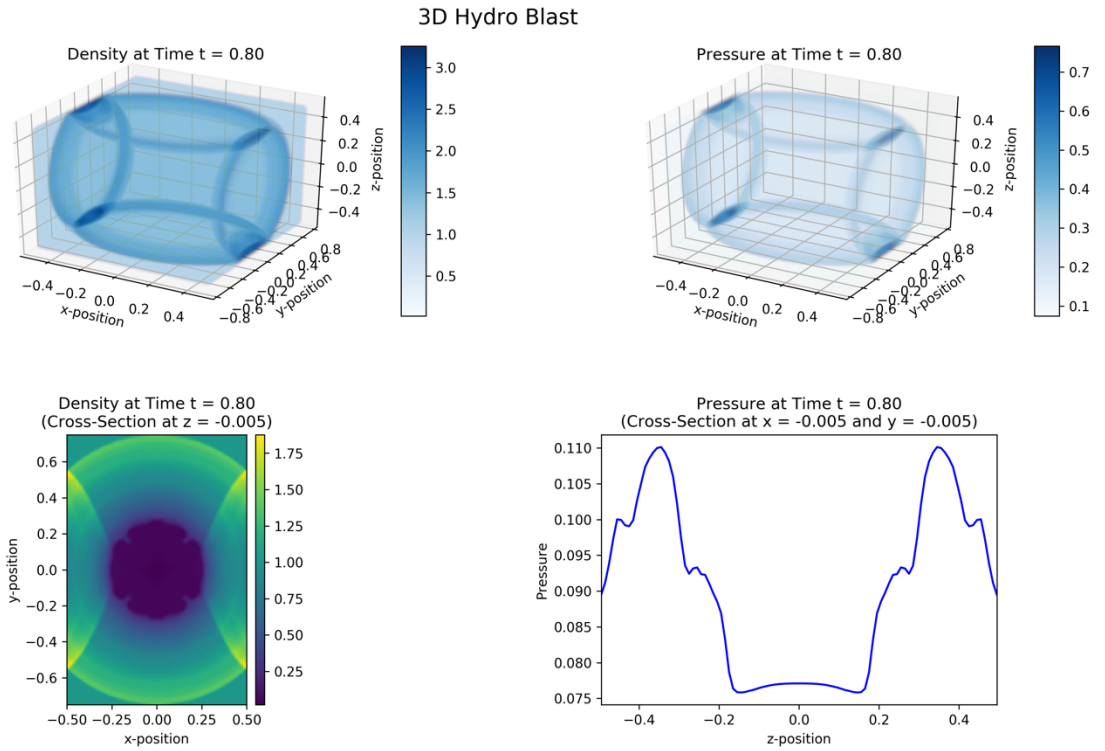

Figure 31: PythonMHD visualization of the 3D hydrodynamic blast at time t = 0.8.

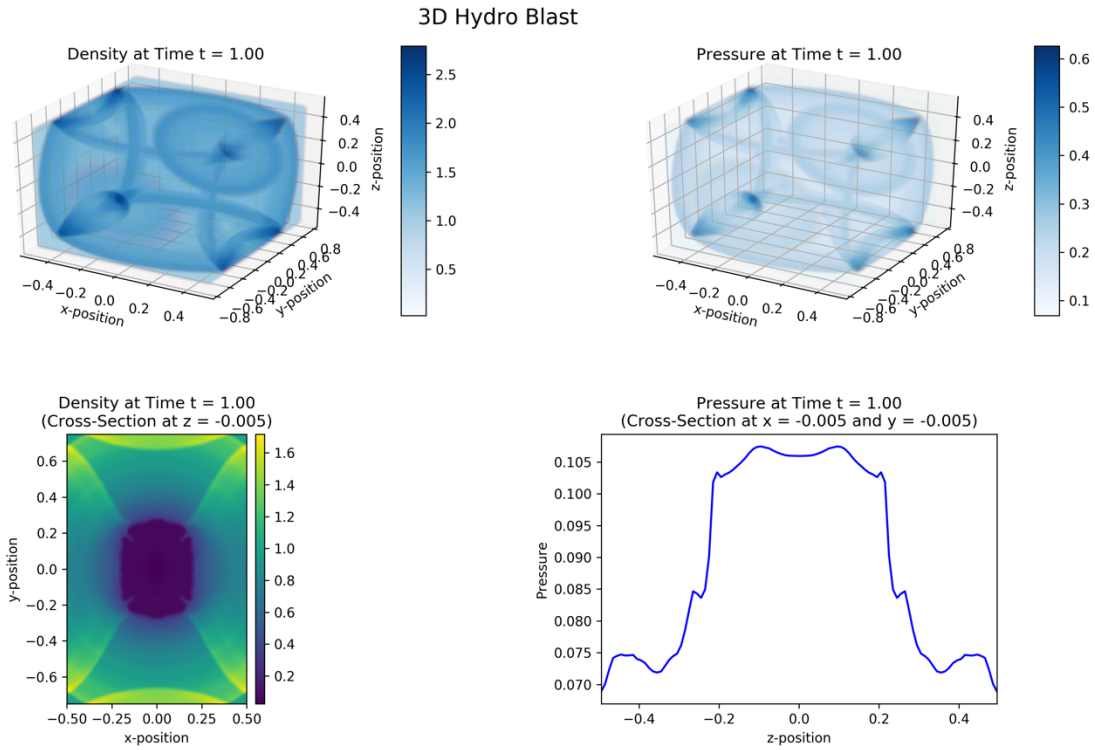

Figure 32: PythonMHD visualization of the 3D hydrodynamic blast at time t = 1.0.

Brio-Wu Shock Tube (1D MHD)

(The script for this test problem is called BrioWuShockTube_1D_InitializationFile.py.)

The Brio-Wu shock tube [7] is the magnetic version of Sod's shock tube. All of the hydrodynamic variables have the same initial values, but we now have a magnetic field with By = +1 in the left chamber, By = -1 in the right chamber, and Bx = 0.75 in both chambers. Figures 33 and 34 show PythonMHD outputs for this standard problem in 1D MHD.

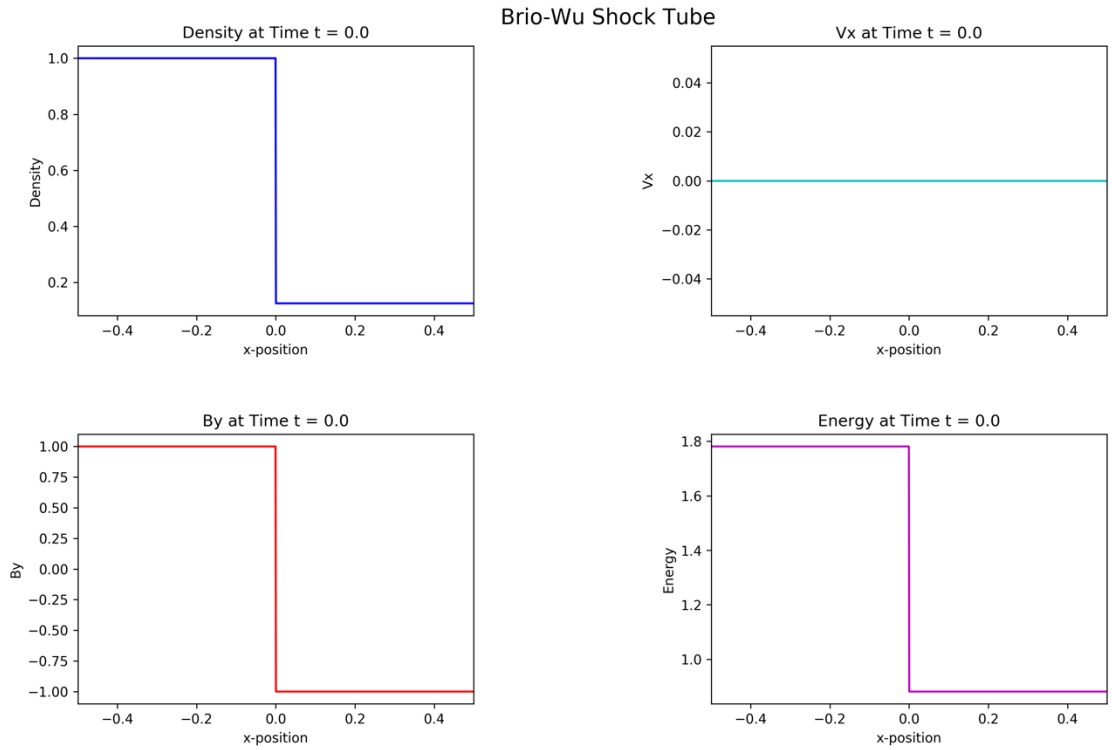

Figure 33: PythonMHD visualization of the Brio-Wu shock tube at time t = 0.

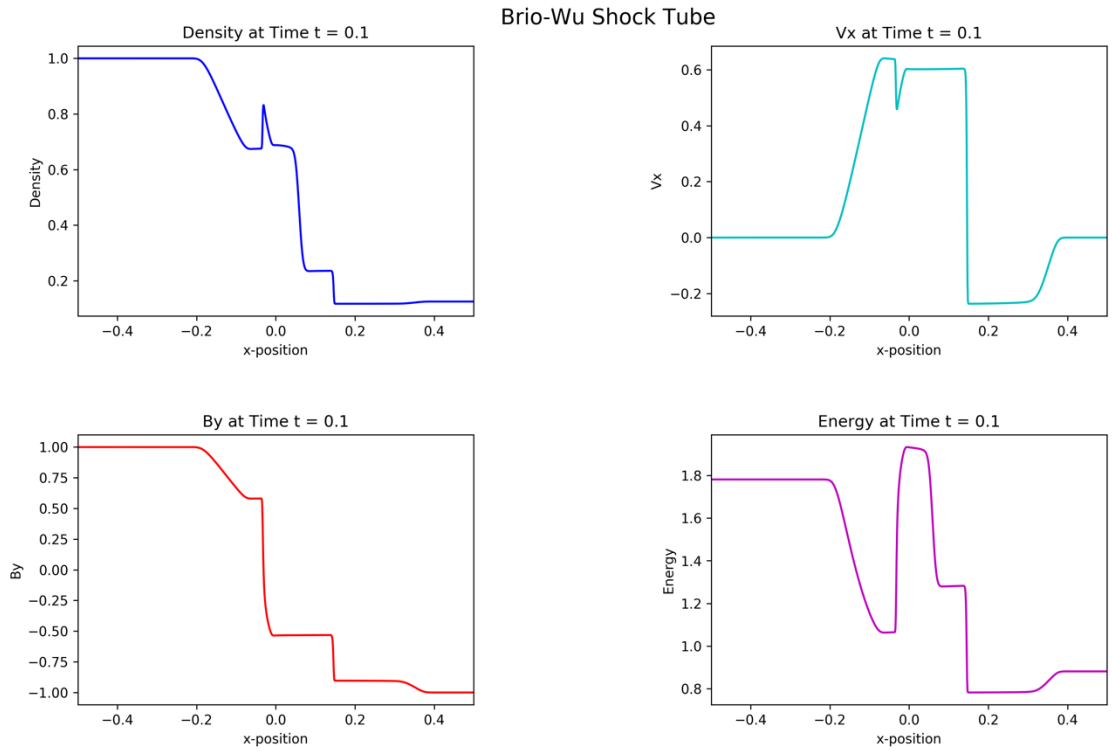

Figure 34: PythonMHD visualization of the Brio-Wu shock tube at time t = 0.1.

Orszag-Tang Vortex (2D MHD)

(The script for this test problem is called OrszagTang_InitializationFile.py.)

The Orszag-Tang vortex [8] is a standard test problem for 2D MHD. At time t = 0, the gas initially has a uniform density and pressure. We create the vortex by having velocities and magnetic field components vary sinusoidally across the simulation grid. Please see the initialization file for details on how these values are initialized. Figures 35-38 show PythonMHD outputs for this test problem.

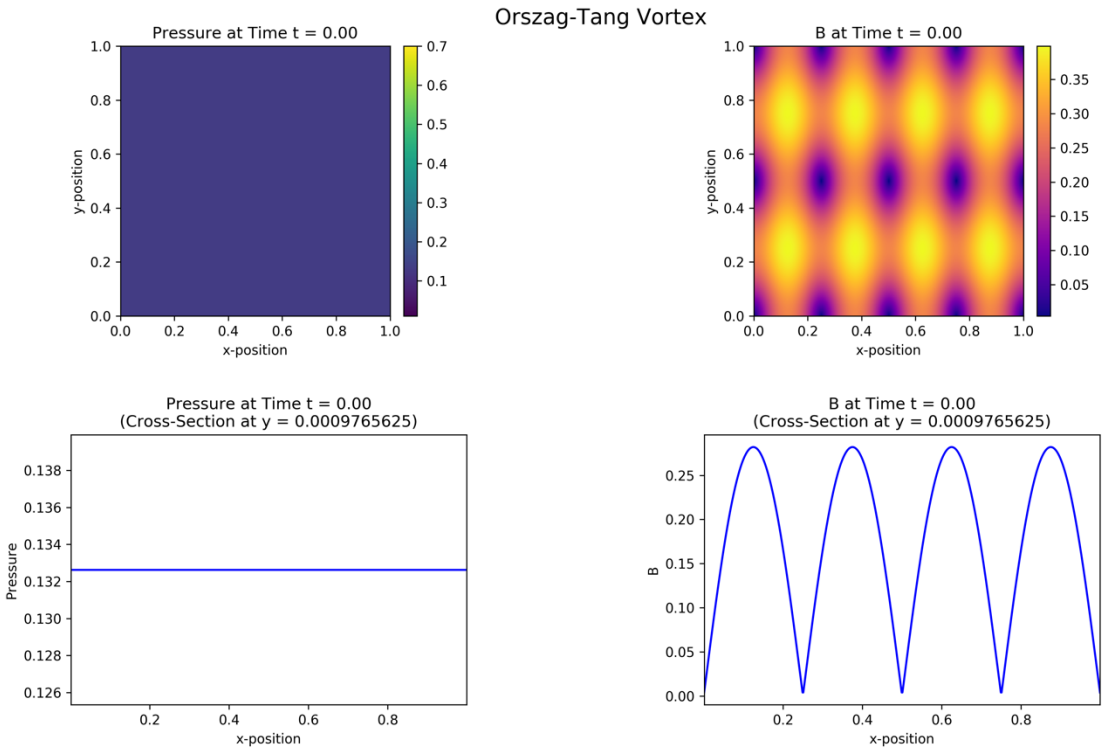

Figure 35: PythonMHD visualization of the Orszag-Tang vortex at time t = 0.0.

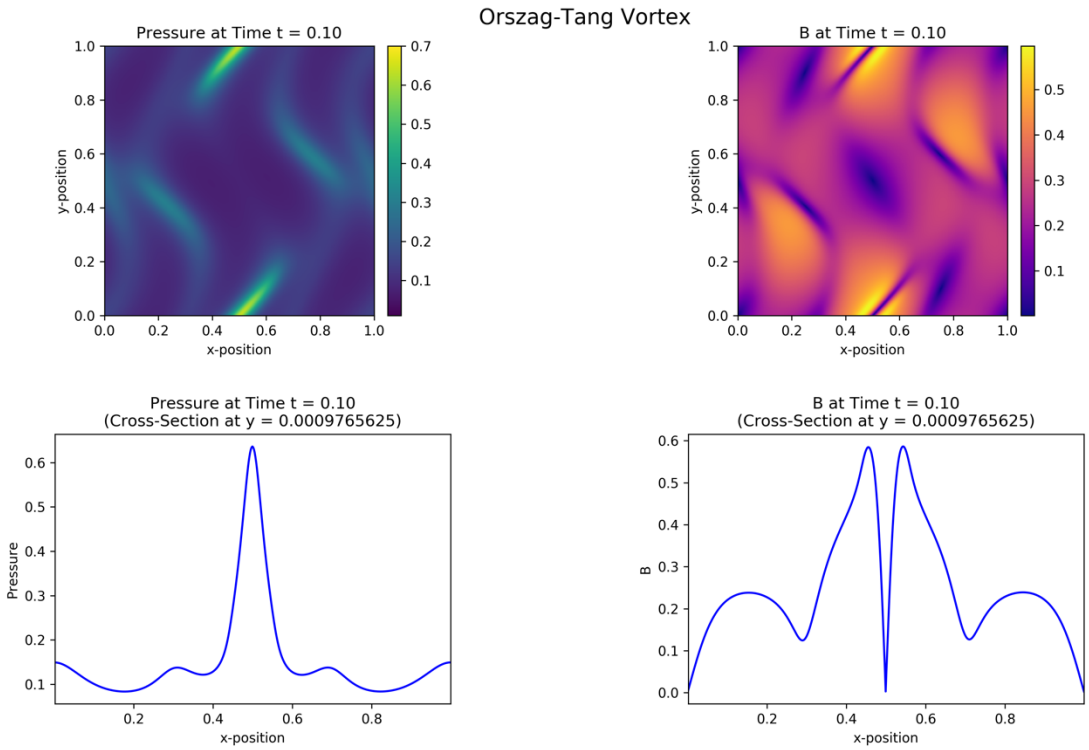

Figure 36: PythonMHD visualization of the Orszag-Tang vortex at time t = 0.1.

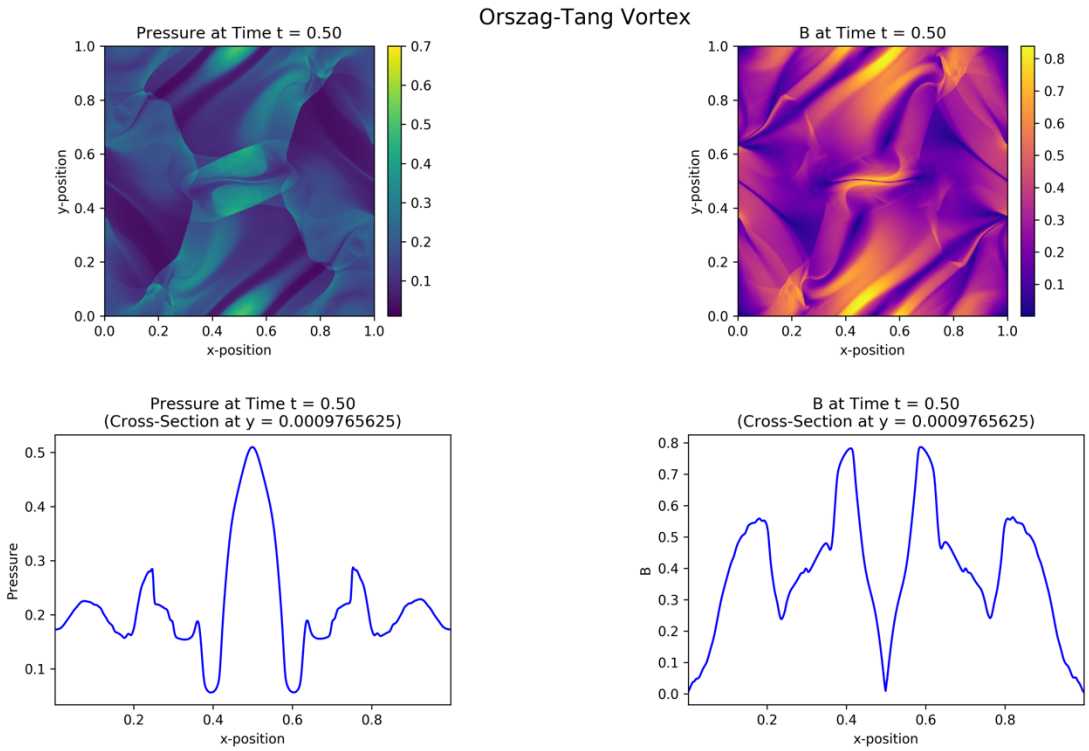

Figure 37: PythonMHD visualization of the Orszag-Tang vortex at time t = 0.5.

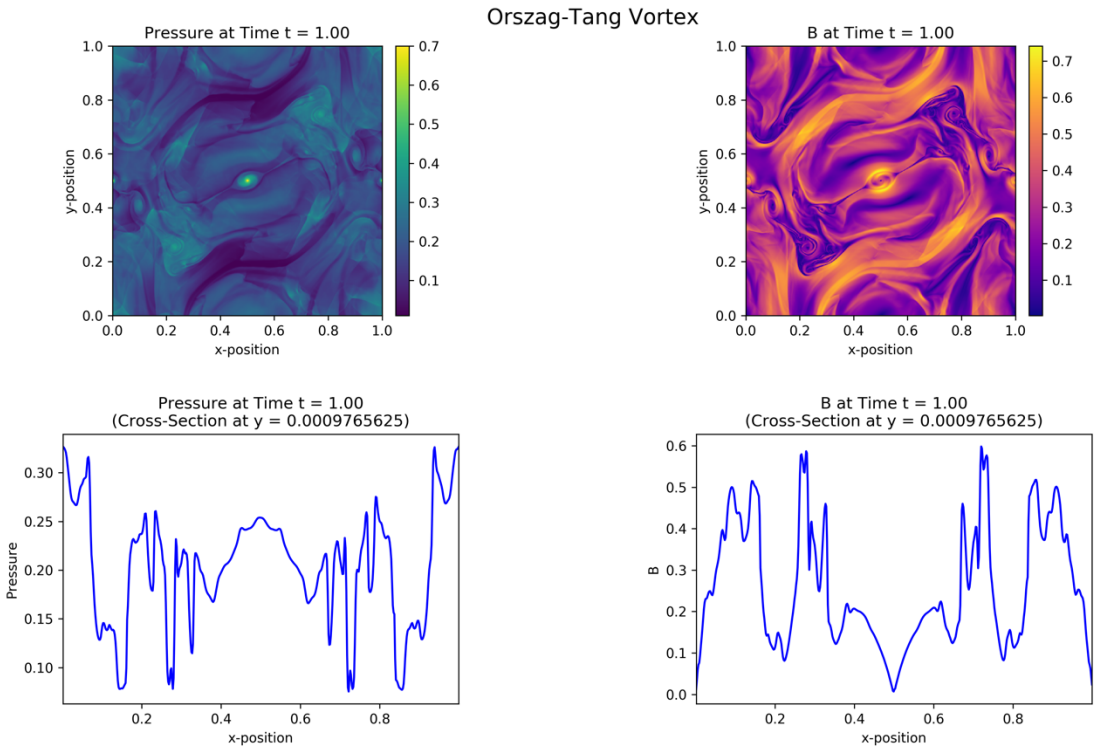

Figure 38: PythonMHD visualization of the Orszag-Tang vortex at time t = 1.0.

3D MHD Blast (3D MHD)

(The script for this test problem is called MHDBlast_3D_InitializationFile.py.)

This problem is the magnetic version of the 3D hydrodynamic blast [6]. With the same hydrodynamic variables that we used in the hydrodynamic problem, we introduce a uniform magnetic field with a magnitude of 1/4pi. The direction of the magnetic field is also uniform, with all of the field vectors forming a 45-degree angle with the x-axis. PythonMHD results for this problem are shown in Figures 39-42.

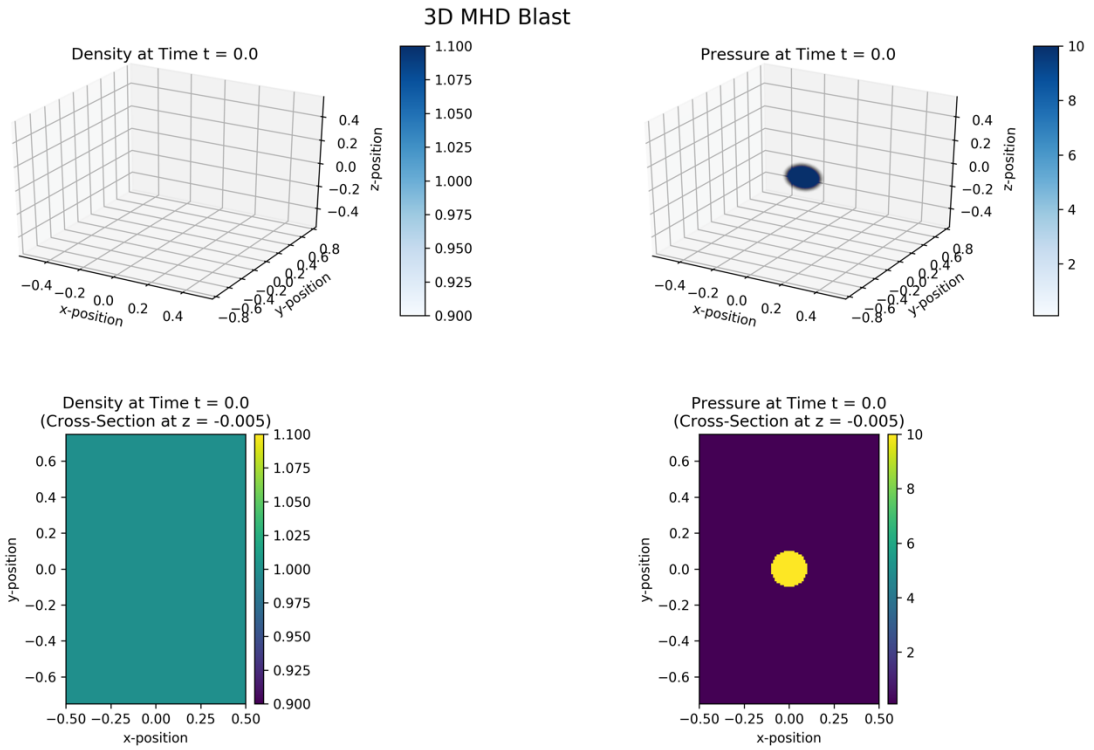

Figure 39: PythonMHD visualization of the 3D MHD blast at time t = 0.0.

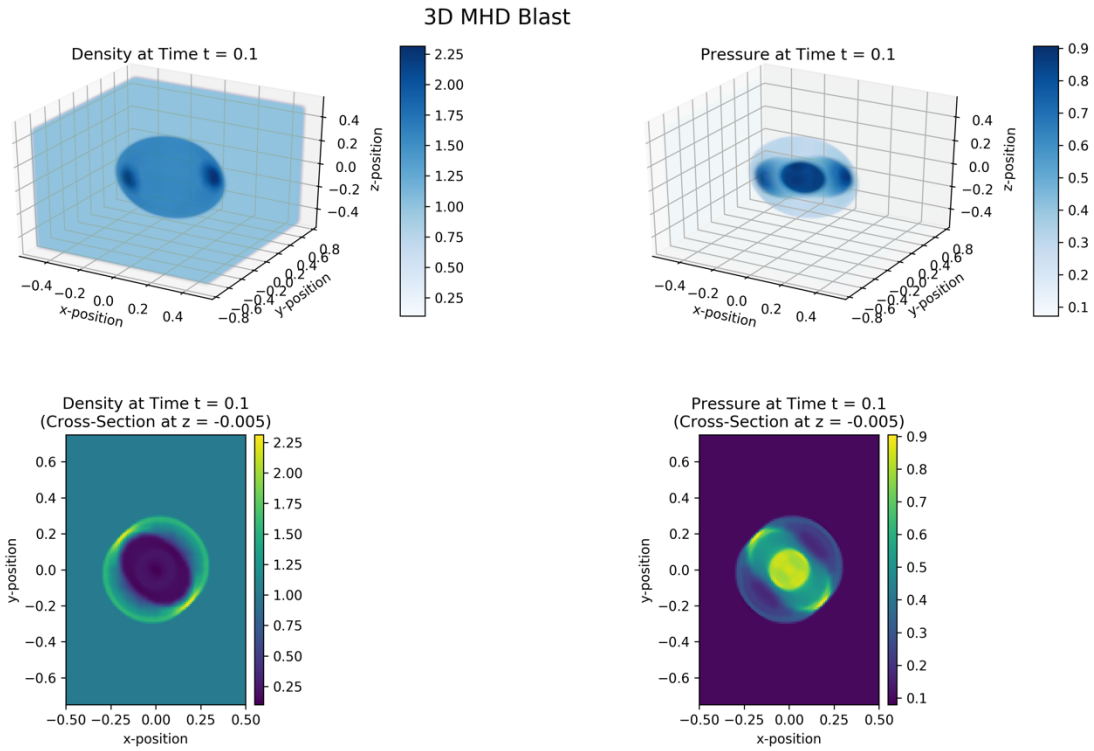

Figure 40: PythonMHD visualization of the 3D MHD blast at time t = 0.1.

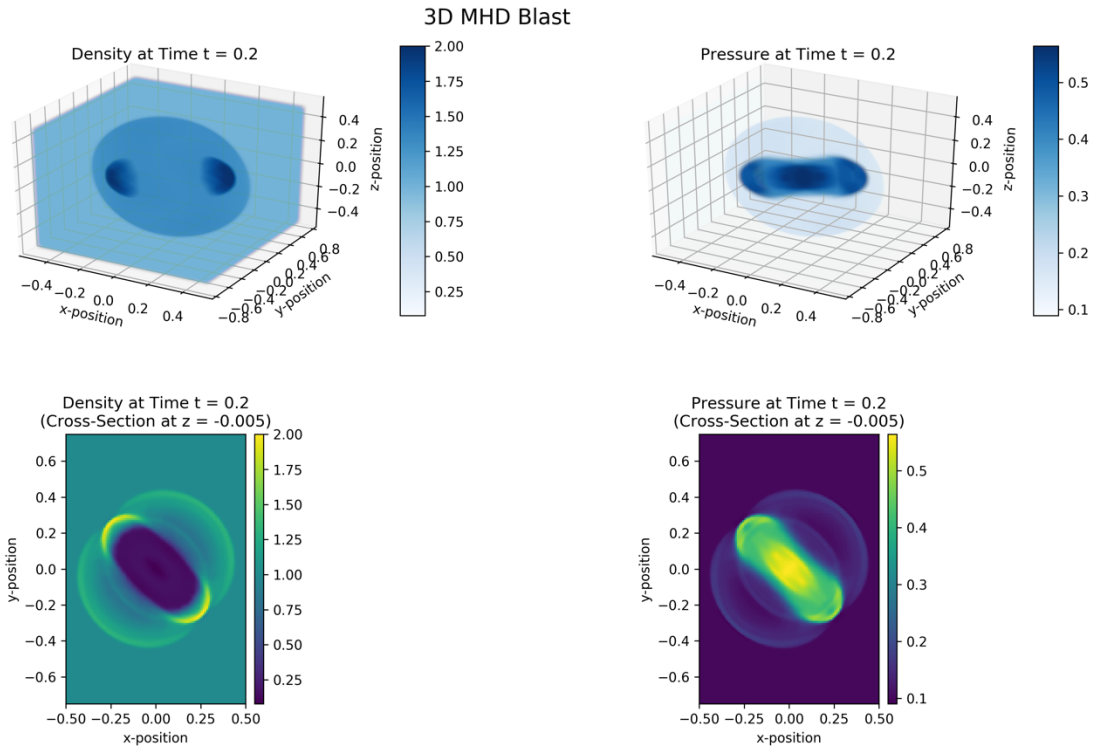

Figure 41: PythonMHD visualization of the 3D MHD blast at time t = 0.2.

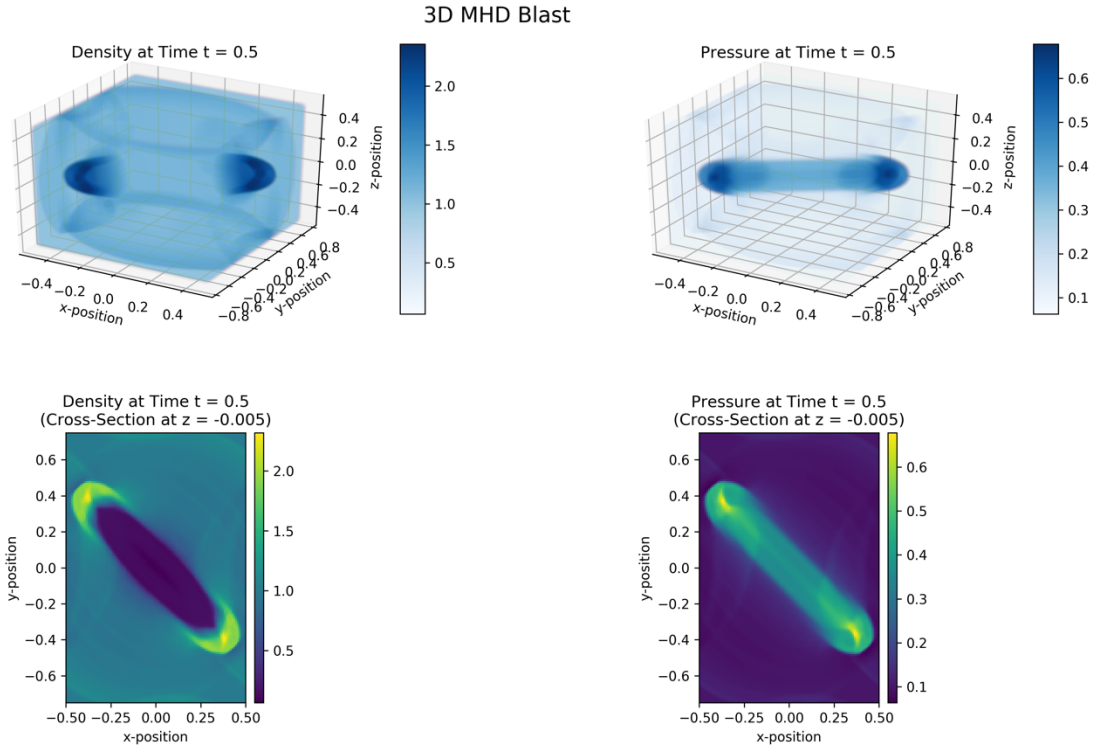

Figure 42: PythonMHD visualization of the 3D MHD blast at time t = 0.5.

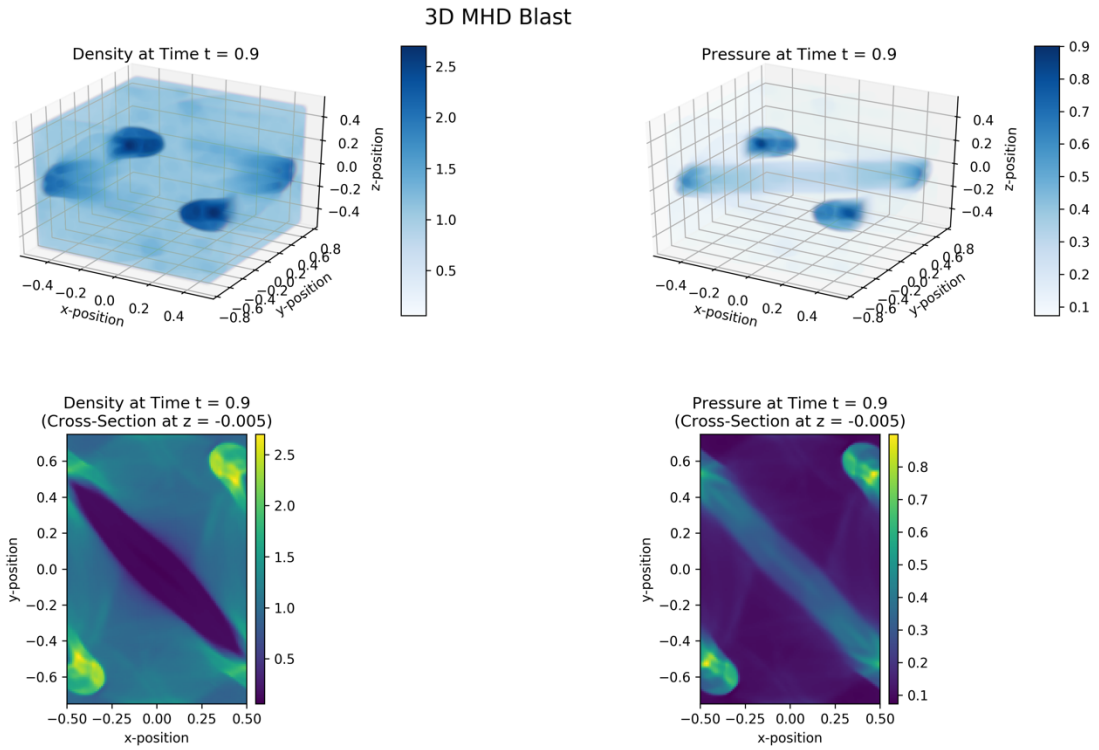

Figure 43: PythonMHD visualization of the 3D MHD blast at time t = 0.9.

PythonMHD Methods

The methods that PythonMHD uses to evolve hydrodynamic and MHD gas systems are documented thoroughly (with references to the original inventors of all algorithms) in the comments throughout PythonMHD's source files. My goal was for the code itself to basically serve as a textbook on computational fluid dynamics for anyone who wants to learn more about how MHD simulation codes model astrophysical gas systems.

As mentioned previously, Athena [4,5] was the main inspiration for all of the numerical methods in PythonMHD. When implementing the simulation algorithms, my goal was to generate simulation results that are numerically identical to Athena (within round-off error). Details on the numerical differences between Athena and PythonMHD will be posted in the near future. For hydrodynamics, PythonMHD implements Athena's version of Colella's Corner

Transport Upwind (CTU) method [9]. For MHD simulations, PythonMHD implements Athena's Corner Transport Upwind + Constrained Transport (CTU+CT) method [4].

Coming Soon: Future Code Releases

Many new features and test problems will be added over the next few weeks and months. Below is a list of new features that will be released as soon as possible, in the order in which they are most likely to become available:

- Line Integral Convolution/LIC Visualizations for Magnetic Fields and Velocities
- Physical/Simulation Unit Convertor
- First-Order/PLM Spatial Reconstruction
    - PythonMHD currently only supports no spatial reconstruction or second-order/PPM reconstruction.
- MPI Parallelization for Faster Runtimes
- Pre-Compiled C Functions for Faster Runtimes
- Non-Cartesian Coordinates
- More Riemann Solvers
- Self-Gravity

References

[1] Toro, E. F. (2009). Riemann solvers and numerical methods for fluid dynamics: A practical introduction. Springer, Berlin, Heidelberg. https://doi-org.uml.idm.oclc.org/10.1007/b79761.

[2] Leboe-McGowan, D. S. (2022). PythonMHD: A new simulation code for astrophysical magnetohydrodynamics. [Master's thesis, University of Manitoba]. https://mspace.lib.umanitoba.ca/items/7b23c9bc-a038-4f2b-a616-ce6393afed8b

[3] Sod, G. (1978). A survey of several finite difference methods for systems of nonlinear

hyperbolic conservation laws. Journal of Computational Physics, 27(1), 1-31. https://doi.org/10.1016/0021-9991(78)90023-2.

[4] Stone, J. M., Gardiner, T. A., Teuben, P., Hawley, J. F., & Simon, J. B. (2008). Athena: A new code for astrophysical MHD. The Astrophysical Journal Supplemental Series, 178(1), 137-177. https://iopscience.iop.org/article/10.1086/588755/pdf.

[5] https://github.com/PrincetonUniversity/Athena-Cversion

[6] Gardiner, T. A., & Stone, J. M. (2005). An unsplit Godunov method for ideal MHD via Constrained Transport. Journal of Computational Physics, 205(2), 509-539. https://arxiv.org/pdf/astro-ph/0501557.pdf.

[7] Brio, M., and Wu, C. C. (1988). An upwind differencing scheme for the equations of ideal magnetohydrodynamics. Journal of Computational Physics, 75(2), 400-422.

[8] Orszag, S. A., & Tang, C. M. (1979). Small-scale structure of two-dimensional magnetohydrodynamic turbulence. Journal of Fluid Mechanics, 90(1). 129-143. https://doi.org/10.1017/S002211207900210X.

[9] Colella, P. (1990). Multidimensional upwind methods for hyperbolic conservation laws. Journal of Computational Physics, 87(1), 171-200. https://doi.org/10.1016/0021-9991(90)90233-Q.