pyro Documentation

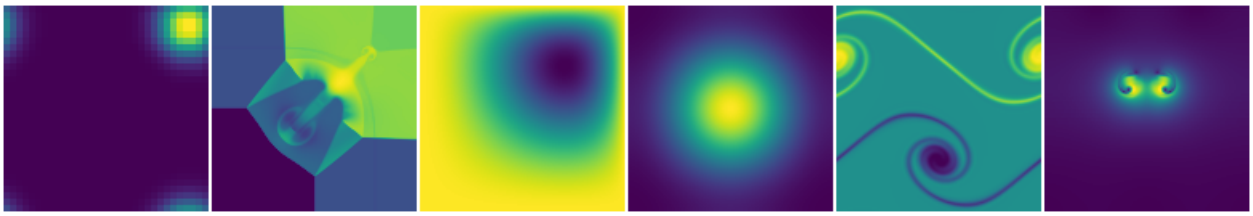
Release 2.2

pyro development team

Oct 19, 2022

PYRO BASICS

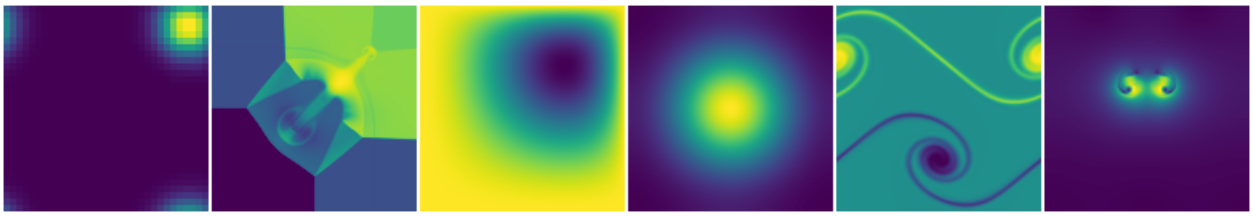
http://github.com/python-hydro/pyro2



CHAPTER

ONE

INTRODUCTION TO PYRO



pyro is a simple framework for implementing and playing with hydrodynamics solvers. It is designed to provide a tutorial for students in computational astrophysics (and hydrodynamics in general) and for easily prototyping new methods. We introduce simple implementations of some popular methods used in the field, with the code written to be easily understandable. All simulations use a single grid (no domain decomposition).

Note: pyro is not meant for demanding scientific simulations—given the choice between performance and clarity, clarity is taken.

pyro builds off of a finite-volume framework for solving PDEs. There are a number of solvers in pyro, allowing for the solution of hyperbolic (wave), parabolic (diffusion), and elliptic (Poisson) equations. In particular, the following solvers are developed:

- linear advection
- compressible hydrodynamics
- shallow water hydrodynamics
- multigrid
- implicit thermal diffusion
- incompressible hydrodynamics
- low Mach number atmospheric hydrodynamics

Runtime visualization shows the evolution as the equations are solved.

CHAPTER

TWO

SETTING UP PYRO

You can clone pyro from github: http://github.com/python-hydro/pyro2

Note: It is strongly recommended that you use python 3.x. While python 2.x might still work, we do not test pyro under python 2, so it may break at any time in the future.

The following python packages are required:

- `numpy`
- `matplotlib`
- `numba`
- `h5py`
- `pytest` (for unit tests)

The following steps are needed before running pyro:

- add `pyro/` to your `PYTHONPATH` environment variable (note this is only needed if you wish to use pyro as a python module - this step is not necessary if you only run pyro via the commandline using the `pyro.py` script). For the bash shell, this is done as:

```
export PYTHONPATH="/path/to/pyro/:${PYTHONPATH}"
```

- define the environment variable `PYRO_HOME` to point to the `pyro2/` directory (only needed for regression testing)

```
export PYRO_HOME="/path/to/pyro/"
```

2.1 Quick test

Run the advection solver to quickly test if things are setup correctly:

```
./pyro.py advection smooth inputs.smooth
```

You should see a plot window pop up with a smooth pulse advecting diagonally through the periodic domain.

CHAPTER

THREE

NOTES ON THE NUMERICAL METHODS

Detailed discussions and derivations of the numerical methods used in pyro are given in the set of notes Introduction to Computational Astrophysical Hydrodynamics, part of the Open Astrophysics Bookshelf.

CHAPTER

FOUR

DESIGN IDEAS

pyro is written entirely in python (by default, we expect python 3), with a few low-level routines compiled *just-in-time* by *numba* for performance. The `numpy` package is used for representing arrays throughout the python code and the `matplotlib` library is used for visualization. Finally, `pytest` is used for unit testing of some components.

All solvers are written for a 2-d grid. This gives a good balance between complexity and speed.

A paper describing the design philosophy of pyro was accepted to Astronomy & Computing [paper link].

4.1 Directory structure

The files for each solver are in their own sub-directory, with additional sub-directories for the mesh and utilities. Each solver has two sub-directories: `problems/` and `tests/`. These store the different problem setups for the solver and reference output for testing.

Your `PYTHONPATH` environment variable should be set to include the top-level `pyro2/` directory.

The overall structure is:

- `pyro2/`: This is the top-level directory. The main driver, `pyro.py`, is here, and all pyro simulations should be run from this directory.
- `advection/`: The linear advection equation solver using the CTU method. All advection-specific routines live here.
  - `problems/`: The problem setups for the advection solver.
  - `tests/`: Reference advection output files for comparison and regression testing.
- `advection_fv4/`: The fourth-order accurate finite-volume advection solver that uses RK4 time integration.
  - `problems/`: The problem setups for the fourth-order advection solver.
  - `tests/`: Reference advection output files for comparison and regression testing.
- `advection_nonuniform/`: The solver for advection with a non-uniform velocity field.
  - `problems/`: The problem setups for the non-uniform advection solver.
  - `tests/`: Reference advection output files for comparison and regression testing.
- `advection_rk/`: The linear advection equation solver using the method-of-lines approach.
  - `problems/`: This is a symbolic link to the advection/problems/ directory.
  - `tests/`: Reference advection output files for comparison and regression testing.
- `advection_weno/`: The method-of-lines WENO solver for linear advection.
  - `problems/`: This is a symbolic link to the advection/problems/ directory.

- analysis/: Various analysis scripts for processing pyro output files.
- compressible/: The fourth-order accurate finite-volume compressible hydro solver that uses RK4 time integration. This is built from the method of McCourquodale and Colella (2011).
  - problems/: The problem setups for the fourth-order compressible hydrodynamics solver.
  - tests/: Reference compressible hydro output for regression testing.
- compressible_fv4/: The compressible hydrodynamics solver using the CTU method. All source files specific to this solver live here.
  - problems/: This is a symbolic link to the compressible/problems/ directory.
  - tests/: Reference compressible hydro output for regression testing.
- compressible_rk/: The compressible hydrodynamics solver using method of lines integration.
  - problems/: This is a symbolic link to the compressible/problems/ directory.
  - tests/: Reference compressible hydro output for regression testing.
- compressible_sdc/: The fourth-order compressible solver, using spectral-deferred correction (SDC) for the time integration.
  - problems/: This is a symbolic link to the compressible/problems/ directory.
  - tests/: Reference compressible hydro output for regression testing.
- diffusion/: The implicit (thermal) diffusion solver. All diffusion-specific routines live here.
  - problems/: The problem setups for the diffusion solver.
  - tests/: Reference diffusion output for regression testing.
- incompressible/: The incompressible hydrodynamics solver. All incompressible-specific routines live here.
  - problems/: The problem setups for the incompressible solver.
  - tests/: Reference incompressible hydro output for regression testing.
- lm_atm/: The low Mach number hydrodynamics solver for atmospherical flows. All low-Mach-specific files live here.
  - problems/: The problem setups for the low Mach number solver.
  - tests/: Reference low Mach hydro output for regression testing.
- mesh/: The main classes that deal with 2-d cell-centered grids and the data that lives on them. All the solvers use these classes to represent their discretized data.
- multigrid/: The multigrid solver for cell-centered data. This solver is used on its own to illustrate how multigrid works, and directly by the diffusion and incompressible solvers.
  - problems/: The problem setups for when the multigrid solver is used in a stand-alone fashion.
  - tests/: Reference multigrid solver solutions (from when the multigrid solver is used stand-alone) for regression testing.
- particles/: The solver for Lagrangian tracer particles.
  - tests/: Particle solver testing.
- swe/: The shallow water solver.
  - problems/: The problem setups for the shallow water solver.
  - tests/: Reference shallow water output for regression testing.

- `util/`: Various service modules used by the pyro routines, including runtime parameters, I/O, profiling, and pretty output modes.

4.2 Numba

`numba` is used to speed up some critical portions of the code. Numba is a *just-in-time compiler* for python. When a call is first made to a function decorated with Numba's `@njit` decorator, it is compiled to machine code 'just-in-time' for it to be executed. Once compiled, it can then run at (near-to) native machine code speed.

We also use Numba's `cache=True` option, which means that once the code is compiled, Numba will write the code into a file-based cache. The next time you run the same bit of code, Numba will use the saved version rather than compiling the code again, saving some compilation time at the start of the simulation.

Note: Because we have chosen to cache the compiled code, Numba will save it in the `__pycache__` directories. If you change the code, a new version will be compiled and saved, but the old version will not be deleted. Over time, you may end up with many unneeded files saved in the `__pycache__` directories. To clean up these files, you can run `./mk.sh clean` in the main `pyro2` directory.

4.3 Main driver

All the solvers use the same driver, the main `pyro.py` script. The flowchart for the driver is:

- parse runtime parameters
- setup the grid (`initialize()` function from the solver)
  - initialize the data for the desired problem (`init_data()` function from the problem)
- do any necessary pre-evolution initialization (`preevolve()` function from the solver)
- evolve while t < tmax and n < max_steps
  - fill boundary conditions (`fill_BC_all()` method of the `CellCenterData2d` class)
  - get the timestep (`compute_timestep()` calls the solver's `method_compute_timestep()` function from the solver)
  - evolve for a single timestep (`evolve()` function from the solver)
  - t = t + dt
  - output (`write()` method of the `CellCenterData2d` class)
  - visualization (`dovis()` function from the solver)
- call the solver's `finalize()` function to output any useful information at the end

This format is flexible enough for the advection, compressible, diffusion, and incompressible evolution solver. Each solver provides a `Simulation` class that provides the following methods (note: inheritance is used, so many of these methods come from the base `NullSimulation` class):

- `compute_timestep`: return the timestep based on the solver's specific needs (through `method_compute_timestep()`) and timestepping parameters in the driver
- `dovis`: performs visualization of the current solution
- `evolve`: advances the system of equations through a single timestep

- `finalize`: any final clean-ups, printing of analysis hints.
- `finished`: return True if we've met the stopping criteria for a simulation
- `initialize`: sets up the grid and solution variables
- `method_compute_timestep`: returns the timestep for evolving the system
- `preevolve`: does any initialization to the fluid state that is necessary before the main evolution. Not every solver will need something here.
- `read_extras`: read in any solver-specific data from a stored output file
- `write`: write the state of the simulation to an HDF5 file
- `write_extras`: any solver-specific writing

Each problem setup needs only provide an `init_data()` function that fills the data in the patch object.

CHAPTER

FIVE

RUNNING

Pyro can be run in two ways: either from the commandline, using the `pyro.py` script and passing in the solver, problem and inputs as arguments, or by using the `Pyro` class.

5.1 Commandline

The `pyro.py` script takes 3 arguments: the solver name, the problem setup to run with that solver (this is defined in the solver's `problems/` sub-directory), and the inputs file (again, usually from the solver's `problems/` directory).

For example, to run the Sedov problem with the compressible solver we would do:

```
./pyro.py compressible sedov inputs.sedov
```

This knows to look for `inputs.sedov` in `compressible/problems/` (alternately, you can specify the full path for the inputs file).

To run the smooth Gaussian advection problem with the advection solver, we would do:

```
./pyro.py advection smooth inputs.smooth
```

Any runtime parameter can also be specified on the command line, after the inputs file. For example, to disable runtime visualization for the above run, we could do:

```
./pyro.py advection smooth inputs.smooth vis.dovis=0
```

Note: Quite often, the slowest part of the runtime is the visualization, so disabling vis as shown above can dramatically speed up the execution. You can always plot the results after the fact using the `plot.py` script, as discussed in *Analysis routines*.

5.2 Pyro class

Alternatively, pyro can be run using the `Pyro` class. This provides an interface that enables simulations to be set up and run in a Jupyter notebook – see `examples/examples.ipynb` for an example notebook. A simulation can be set up and run by carrying out the following steps:

- create a `Pyro` object, initializing it with a specific solver
- initialize the problem, passing in runtime parameters and inputs
- run the simulation

For example, if we wished to use the `compressible` solver to run the Kelvin-Helmholtz problem `kh`, we would do the following:

```
from pyro import Pyro
pyro = Pyro("compressible")
pyro.initialize_problem(problem_name="kh",
                        inputs_file="inputs.kh")
pyro.run_sim()
```

Instead of using an inputs file to define the problem parameters, we can define a dictionary of parameters and pass them into the `initialize_problem` function using the keyword argument `inputs_dict`. If an inputs file is also passed into the function, the parameters in the dictionary will override any parameters in the file. For example, if we wished to turn off visualization for the previous example, we would do:

```
parameters = {"vis.dovis":0}
pyro.initialize_problem(problem_name="kh",
                        inputs_file="inputs.kh",
                        inputs_dict=parameters)
```

It's possible to evolve the simulation forward timestep by timestep manually using the `single_step` function (rather than allowing `run_sim` to do this for us). To evolve our example simulation forward by a single step, we'd run

```
pyro.single_step()
```

This will fill the boundary conditions, compute the timestep `dt`, evolve a single timestep and do output/visualization (if required).

5.3 Runtime options

The behavior of the main driver, the solver, and the problem setup can be controlled by runtime parameters specified in the inputs file (or via the command line or passed into the `initialize_problem` function). Runtime parameters are grouped into sections, with the heading of that section enclosed in `[ .. ]`. The list of parameters are stored in three places:

- the `pyro/_defaults` file
- the solver's `_defaults` file
- problem's `_defaults` file (named `_problem-name.defaults` in the solver's `problem/` sub-directory).

These three files are parsed at runtime to define the list of valid parameters. The inputs file is read next and used to override the default value of any of these previously defined parameters. Additionally, any parameter can be specified at the end of the commandline, and these will be used to override the defaults. The collection of runtime parameters is stored in a `RuntimeParameters` object.

The `runparams.py` module in `util/` controls access to the runtime parameters. You can setup the runtime parameters, parse an inputs file, and access the value of a parameter (`hydro.cfl` in this example) as:

```
rp = RuntimeParameters()
rp.load_params("inputs.test")
...
cfl = rp.get_param("hydro.cfl")
```

When pyro is run, the file `inputs.auto` is output containing the full list of runtime parameters, their value for the simulation, and the comment that was associated with them from the `_defaults` files. This is a useful way to see what parameters are in play for a given simulation.

All solvers use the following parameters:

- section: [driver]

option	value	description
tmax	1.0	maximum simulation time to evolve
max_steps	10000	maximum number of steps to take
fix_dt	-1.0	
init_tstep_factor	0.01	first timestep = init_tstep_factor * CFL timestep
max_dt_change	2.0	max amount the timestep can change between steps
verbose	1.0	verbosity

- section: [io]

option	value	description
basename	pyro_	basename for output files
dt_out	0.1	simulation time between writing output files
n_out	10000	number of timesteps between writing output files
do_io	1	do we output at all?

- section: [mesh]

option	value	description
xmin	0.0	domain minumum x-coordinate
xmax	1.0	domain maximum x-coordinate
ymin	0.0	domain minimum y-coordinate
ymax	1.0	domain maximum y-coordinate
xlboundary	reflect	minimum x BC (‘reflect’, ‘outflow’, or ‘periodic’)
xrboundary	reflect	maximum x BC (‘reflect’, ‘outflow’, or ‘periodic’)
ylboundary	reflect	minimum y BC (‘reflect’, ‘outflow’, or ‘periodic’)
yrboundary	reflect	maximum y BC (‘reflect’, ‘outflow’, or ‘periodic’)
nx	25	number of zones in the x-direction
ny	25	number of zones in the y-direction

- section: [particles]

option	value	description
do_particles	0	include particles? (1=yes, 0=no)
n_particles	100	number of particles
particle_generator	random	how do we generate particles? (random, grid)

- section: [vis]

option	value	description
dovis	1	runtime visualization? (1=yes, 0=no)
store_images	0	store vis images to files (1=yes, 0=no)

CHAPTER

SIX

WORKING WITH OUTPUT

6.1 Utilities

Several simply utilities exist to operate on output files

- `compare.py`: this script takes two plot files and compares them zone-by-zone and reports the differences. This is useful for testing, to see if code changes affect the solution. Many problems have stored benchmarks in their solver's tests directory. For example, to compare the current results for the incompressible shear problem to the stored benchmark, we would do:

  ```
  ./compare.py shear_128_0216.pyro incompressible/tests/shear_128_0216.pyro
  ```

  Differences on the order of machine precision may arise because of optimizations and compiler differences across platforms. Students should familiarize themselves with the details of how computers store numbers (floating point). An excellent read is *What every computer scientist should know about floating-point arithmetic* by D. Goldberg.

- `plot.py`: this script uses the solver's `dovis()` routine to plot an output file. For example, to plot the data in the file `shear_128_0216.pyro` from the incompressible shear problem, you would do:

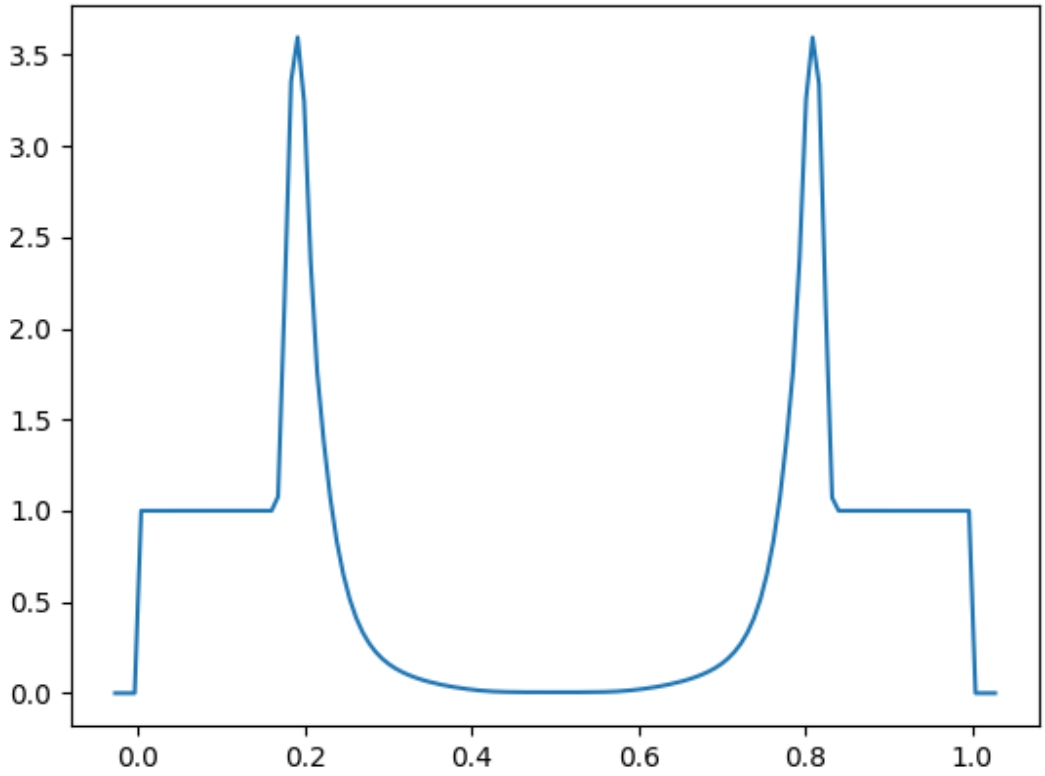
  ```
  ./plot.py -o image.png shear_128_0216.pyro
  ```

  where the `-o` option allows you to specify the output file name.

6.2 Reading and plotting manually

pyro output data can be read using the `util.io_pyro.read` method. The following sequence (done in a python session) reads in stored data (from the compressible Sedov problem) and plots data falling on a line in the x direction through the y-center of the domain (note: this will include the ghost cells).

```
import matplotlib.pyplot as plt
import util_pyro.io as io
sim = io.read("sedov_unsplit_0000.h5")
dens = sim.cc_data.get_var("density")
plt.plot(dens.g.x, dens[:,dens.g.ny//2])
plt.show()
```



Note: this includes the ghost cells, by default, seen as the small regions of zeros on the left and right.

CHAPTER

SEVEN

ADDING A PROBLEM

The easiest way to add a problem is to copy an existing problem setup in the solver you wish to use (in its problems/ sub-directory). Three different files will need to be copied (created):

- `problem.py`: this is the main initialization routine. The function `init_data()` is called at runtime by the `Simulation` object's `initialize()` method. Two arguments are passed in, the simulation's `CellCenterData2d` object and the `RuntimeParameters` object. The job of `init_data()` is to fill all of the variables defined in the `CellCenterData2d` object.
- `_problem.defaults`: this contains the runtime parameters and their defaults for your problem. They should be placed in a block with the heading `[problem]` (where `problem` is your problem's name). Anything listed here will be available through the `RuntimeParameters` object at runtime.
- `inputs.problem`: this is the inputs file that is used at runtime to set the parameters for your problem. Any of the general parameters (like the grid size, boundary conditions, etc.) as well as the problem-specific parameters can be set here. Once the problem is defined, you need to add the problem name to the `__all__` list in the `__init__.py` file in the `problems/` sub-directory. This lets python know about the problem.

CHAPTER

EIGHT

MESH OVERVIEW

All solvers are based on a finite-volume/cell-centered discretization. The basic theory of such methods is discussed in *Notes on the numerical methods*.

Note: The core data structure that holds data on the grid is `CellCenterData2d`. This does not distinguish between cell-centered data and cell-averages. This is fine for methods that are second-order accurate, but for higher-order methods, the `FV2d` class has methods for converting between the two data centerings.

8.1 `mesh.patch` implementation and use

We import the basic mesh functionality as:

```
import mesh.patch as patch
import mesh.fv as fv
import mesh.boundary as bnd
import mesh.array_indexer as ai
```

There are several main objects in the patch class that we interact with:

- `patch.Grid2d`: this is the main grid object. It is basically a container that holds the number of zones in each coordinate direction, the domain extrema, and the coordinates of the zones themselves (both at the edges and center).
- `patch.CellCenterData2d`: this is the main data object—it holds cell-centered data on a grid. To build a `patch.CellCenterData2d` object you need to pass in the `patch.Grid2d` object that defines the mesh. The `patch.CellCenterData2d` object then allocates storage for the unknowns that live on the grid. This class also provides methods to fill boundary conditions, retrieve the data in different fashions, and read and write the object from/to disk.
- `fv.FV2d`: this is a special class derived from `patch.CellCenterData2d` that implements some extra functions needed to convert between cell-center data and averages with fourth-order accuracy.
- `bnd.BC`: This is simply a container that holds the names of the boundary conditions on each edge of the domain.
- `ai.ArrayIndexer`: This is a class that subclasses the NumPy ndarray and makes the data in the array know about the details of the grid it is defined on. In particular, it knows which cells are valid and which are the ghost cells, and it has methods to do the $a_{i+1,j}$ operations that are common in difference methods.
- `integration.RKIntegrator`: This class implements Runge-Kutta integration in time by managing a hierarchy of grids at different time-levels. A Butcher tableau provides the weights and evaluation points for the different stages that make up the integration.

The procedure for setting up a grid and the data that lives on it is as follows:

```
myg = patch.Grid2d(16, 32, xmax=1.0, ymax=2.0)
```

This creates the 2-d grid object `myg` with 16 zones in the x-direction and 32 zones in the y-direction. It also specifies the physical coordinate of the rightmost edge in x and y.

```
mydata = patch.CellCenterData2d(myg)

bc = bnd.BC(xlb="periodic", xrb="periodic", ylb="reflect-even", yrb="outflow")

mydata.register_var("a", bc)
mydata.create()
```

This creates the cell-centered data object, `mydata`, that lives on the grid we just built above. Next we create a boundary condition object, specifying the type of boundary conditions for each edge of the domain, and finally use this to register a variable, `a` that lives on the grid. Once we call the `create()` method, the storage for the variables is allocated and we can no longer add variables to the grid. Note that each variable needs to specify a BC—this allows us to do different actions for each variable (for example, some may do even reflection while others may do odd reflection).

8.2 Jupyter notebook

A Jupyter notebook that illustrates some of the basics of working with the grid is provided as mesh-examples.ipynb. This will demonstrate, for example, how to use the `ArrayIndexer` methods to construct differences.

8.3 Tests

The actual filling of the boundary conditions is done by the `fill_BC` method. The script `bc_demo.py` tests the various types of boundary conditions by initializing a small grid with sequential data, filling the BCs, and printing out the results.

CHAPTER

NINE

ADVECTION SOLVERS

The linear advection equation:

$$a_t + ua_x + va_y = 0$$

provides a good basis for understanding the methods used for compressible hydrodynamics. Chapter 4 of the notes summarizes the numerical methods for advection that we implement in pyro.

pyro has several solvers for linear advection, which solve the equation with different spatial and temporal intergration schemes.

9.1 `advection` solver

`advection` implements the directionally unsplit corner transport upwind algorithm [Colella90] with piecewise linear reconstruction. This is an overall second-order accurate method, with timesteps restricted by

$$\Delta t < \min \left\{ \frac{\Delta x}{|u|}, \frac{\Delta y}{|v|} \right\}$$

The parameters for this solver are:

- section: [advection]

option	value	description
u	`1.0`	advective velocity in x-direction
v	`1.0`	advective velocity in y-direction
limiter	`2`	limiter (0 = none, 1 = 2nd order, 2 = 4th order)

- section: [driver]

option	value	description
cfl	`0.8`	advective CFL number

- section: [particles]

option	value	description
do_particles	`0`	
particle_generator	`grid`	

9.2 advection_fv4 solver

advection_fv4 uses a fourth-order accurate finite-volume method with RK4 time integration, following the ideas in [McCorquodaleColella11]. It can be thought of as a method-of-lines integration, and as such has a slightly more restrictive timestep:

$$\Delta t \lesssim \left[\frac{|u|}{\Delta x} + \frac{|v|}{\Delta y} \right]^{-1}$$

The main complexity comes from needing to average the flux over the faces of the zones to achieve 4th order accuracy spatially.

The parameters for this solver are:

- section: [advection]

option	value	description
u	1.0	advective velocity in x-direction
v	1.0	advective velocity in y-direction
limiter	1	limiter (0 = none, 1 = ppm)
temporal_method	RK4	integration method (see mesh/integrators.py)

- section: [driver]

option	value	description
cfl	0.8	advective CFL number

9.3 advection_nonuniform solver

advection_nonuniform models advection with a non-uniform velocity field. This is used to implement the slotted disk problem from [Zal79]. The basic method is similar to the algorithm used by the main advection solver.

The paramters for this solver are:

- section: [advection]

option	value	description
u	1.0	advective velocity in x-direction
v	1.0	advective velocity in y-direction
limiter	2	limiter (0 = none, 1 = 2nd order, 2 = 4th order)

- section: [driver]

option	value	description
cfl	0.8	advective CFL number

- section: [particles]

option	value	description
do_particles	0	
particle_generator	grid	

9.4 advection_rk solver

advection_rk uses a method of lines time-integration approach with piecewise linear spatial reconstruction for linear advection. This is overall second-order accurate, so it represents a simpler algorithm than the advection_fv4 method (in particular, we can treat cell-centers and cell-averages as the same, to second order).

The parameter for this solver are:

- section: [advection]

option	value	description
u	1.0	advective velocity in x-direction
v	1.0	advective velocity in y-direction
limiter	2	limiter (0 = none, 1 = 2nd order, 2 = 4th order)
temporal_method	RK4	integration method (see mesh/integrators/.py)

- section: [driver]

option	value	description
cfl	0.8	advective CFL number

9.5 advection_weno solver

advection_weno uses a WENO reconstruction and method of lines time-integration

The main parameters that affect this solver are:

- section: [advection]

option	value	description
u	1.0	advective velocity in x-direction
v	1.0	advective velocity in y-direction
limiter	0	Unused here, but needed to inherit from advection base class
weno_order	3	k in WENO scheme
temporal_method	RK4	integration method (see mesh/integrators/.py)

- section: [driver]

option	value	description
cfl	0.5	advective CFL number

9.6 General ideas

The main use for the advection solver is to understand how Godunov techniques work for hyperbolic problems. These same ideas will be used in the compressible and incompressible solvers. This video shows graphically how the basic advection algorithm works, consisting of reconstruction, evolution, and averaging steps:

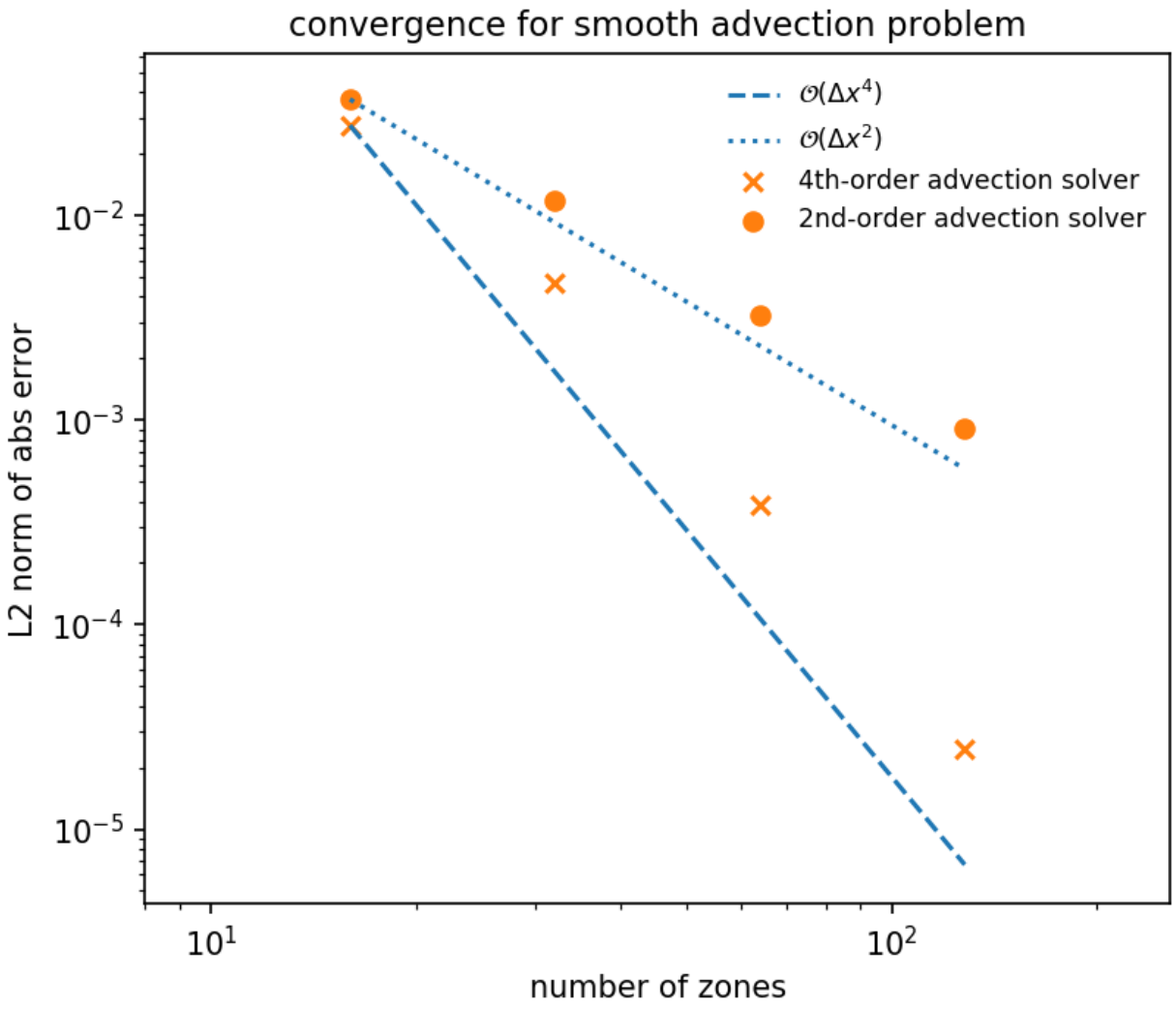
9.7 Examples

9.7.1 smooth

The smooth problem initializes a Gaussian profile and advects it with $u = v = 1$ through periodic boundaries for a period. The result is that the final state should be identical to the initial state—any disagreement is our numerical error. This is run as:

```
./pyro.py advection smooth inputs.smooth
```

By varying the resolution and comparing to the analytic solution, we can measure the convergence rate of the method. The `smooth_error.py` script in `analysis/` will compare an output file to the analytic solution for this problem.



The points above are the L2-norm of the absolute error for the smooth advection problem after 1 period with `CFL=0.8`, for both the `advection` and `advection_fv4` solvers. The dashed and dotted lines show ideal scaling. We see that we achieve nearly 2nd order convergence for the `advection` solver and 4th order convergence with the `advection_fv4` solver. Departures from perfect scaling are likely due to the use of limiters.

9.7.2 tophat

The tophat problem initializes a circle in the center of the domain with value 1, and 0 outside. This has very steep jumps, and the limiters will kick in strongly here.

9.8 Exercises

The best way to learn these methods is to play with them yourself. The exercises below are suggestions for explorations and features to add to the advection solver.

9.8.1 Explorations

- Test the convergence of the solver for a variety of initial conditions (tophat hat will differ from the smooth case because of limiting). Test with limiting on and off, and also test with the slopes set to 0 (this will reduce it down to a piecewise constant reconstruction method).
- Run without any limiting and look for oscillations and under and overshoots (does the advected quantity go negative in the tophat problem?)

9.8.2 Extensions

- Implement a dimensionally split version of the advection algorithm. How does the solution compare between the unsplit and split versions? Look at the amount of overshoot and undershoot, for example.
- Research the inviscid Burger's equation—this looks like the advection equation, but now the quantity being advected is the velocity itself, so this is a non-linear equation. It is very straightforward to modify this solver to solve Burger's equation (the main things that need to change are the Riemann solver and the fluxes, and the computation of the timestep).

  The neat thing about Burger's equation is that it admits shocks and rarefactions, so some very interesting flow problems can be setup.

CHAPTER

TEN

COMPRESSIBLE HYDRODYNAMICS SOLVERS

The Euler equations of compressible hydrodynamics take the form:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho U) = 0$$

$$\frac{\partial (\rho U)}{\partial t} + \nabla \cdot (\rho U U) + \nabla p = \rho g$$

$$\frac{\partial (\rho E)}{\partial t} + \nabla \cdot [(\rho E + p) U] = \rho U \cdot g$$

with $\rho E = \rho e + \frac{1}{2}\rho |U|^2$ and $p = p(\rho, e)$. Note these do not include any dissipation terms, since they are usually negligible in astrophysics.

pyro has several compressible solvers to solve this equation set. The implementations here have flattening at shocks, artificial viscosity, a simple gamma-law equation of state, and (in some cases) a choice of Riemann solvers. Optional constant gravity in the vertical direction is allowed.

Note: All the compressible solvers share the same `problems/` directory, which lives in `compressible/problems/`. For the other compressible solvers, we simply use a symbolic-link to this directory in the solver's directory.

10.1 `compressible` solver

`compressible` is based on a directionally unsplit (the corner transport upwind algorithm) piecewise linear method for the Euler equations, following [Colella90]. This is overall second-order accurate.

The parameters for this solver are:

- section: [compressible]

option	value	description
use_flattening	`1`	apply flattening at shocks (1)
z0	`0.75`	flattening z0 parameter
z1	`0.85`	flattening z1 parameter
delta	`0.33`	flattening delta parameter
cvisc	`0.1`	artifical viscosity coefficient
limiter	`2`	limiter (0 = none, 1 = 2nd order, 2 = 4th order)
grav	`0.0`	gravitational acceleration (in y-direction)
riemann	`HLLC`	HLLC or CGF

- section: [driver]

option	value	description
cfl	0.8	

- section: [eos]

option	value	description
gamma	1.4	pres = rho ener (gamma - 1)

- section: [particles]

option	value	description
do_particles	0	
particle_generator	grid	

10.2 compressible_rk solver

compressible_rk uses a method of lines time-integration approach with piecewise linear spatial reconstruction for the Euler equations. This is overall second-order accurate.

The parameters for this solver are:

- section: [compressible]

option	value	description
use_flattening	1	apply flattening at shocks (1)
z0	0.75	flattening z0 parameter
z1	0.85	flattening z1 parameter
delta	0.33	flattening delta parameter
cvisc	0.1	artifical viscosity coefficient
limiter	2	limiter (0 = none, 1 = 2nd order, 2 = 4th order)
temporal_method	RK4	integration method (see mesh/integration.py)
grav	0.0	gravitational acceleration (in y-direction)
riemann	HLLC	HLLC or CGF
well_balanced	0	use a well-balanced scheme to keep the model in hydrostatic equilibrium

- section: [driver]

option	value	description
cfl	0.8	

- section: [eos]

option	value	description
gamma	1.4	pres = rho ener (gamma - 1)

10.3 compressible_fv4 solver

compressible_fv4 uses a 4th order accurate method with RK4 time integration, following [McCorquodaleColella11].

The parameter for this solver are:

- section: [compressible]

option	value	description
use_flattening	1	apply flattening at shocks (1)
z0	0.75	flattening z0 parameter
z1	0.85	flattening z1 parameter
delta	0.33	flattening delta parameter
cvisc	0.1	artifical viscosity coefficient
limiter	2	limiter (0 = none, 1 = 2nd order, 2 = 4th order)
temporal_method	RK4	integration method (see mesh/integration.py)
grav	0.0	gravitational acceleration (in y-direction)

- section: [driver]

option	value	description
cfl	0.8	

- section: [eos]

option	value	description
gamma	1.4	pres = rho ener (gamma - 1)

10.4 compressible_sdc solver

compressible_sdc uses a 4th order accurate method with spectral-deferred correction (SDC) for the time integration. This shares much in common with the compressible_fv4 solver, aside from how the time-integration is handled.

The parameters for this solver are:

- section: [compressible]

option	value	description
use_flattening	1	apply flattening at shocks (1)
z0	0.75	flattening z0 parameter
z1	0.85	flattening z1 parameter
delta	0.33	flattening delta parameter
cvisc	0.1	artifical viscosity coefficient
limiter	2	limiter (0 = none, 1 = 2nd order, 2 = 4th order)
temporal_method	RK4	integration method (see mesh/integration.py)
grav	0.0	gravitational acceleration (in y-direction)

- section: [driver]

option	value	description
cfl	0.8	

- section: [eos]

option	value	description
gamma	`1.4`	pres = rho ener (gamma - 1)

10.5 Example problems

Note: The 4th-order accurate solver (`compressible_fv4`) requires that the initialization create cell-averages accurate to 4th-order. To allow for all the solvers to use the same problem setups, we assume that the initialization routines initialize cell-centers (which is fine for 2nd-order accuracy), and the `preevolve()` method will convert these to cell-averages automatically after initialization.

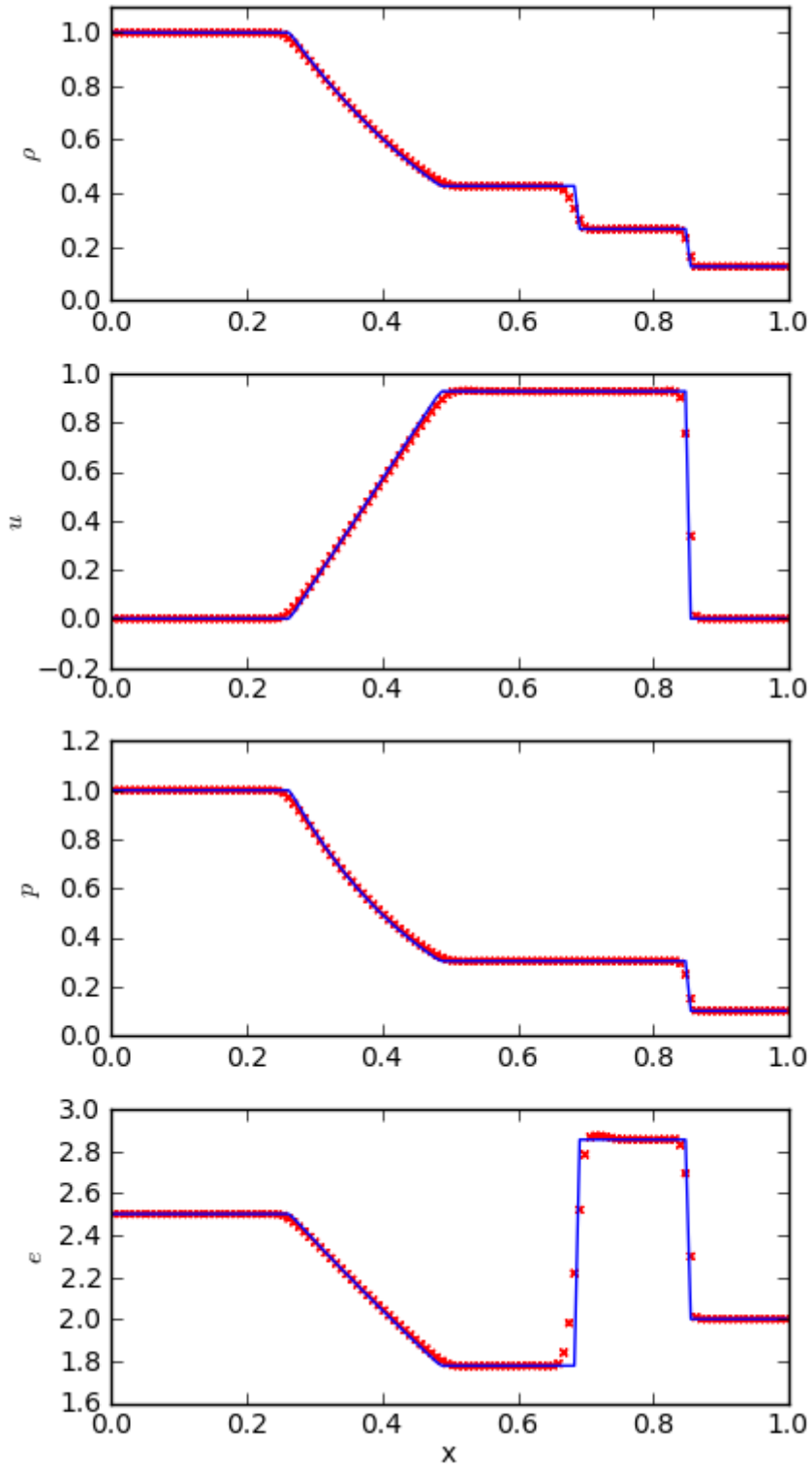
10.5.1 Sod

The Sod problem is a standard hydrodynamics problem. It is a one-dimensional shock tube (two states separated by an interface), that exhibits all three hydrodynamic waves: a shock, contact, and rarefaction. Furthermore, there are exact solutions for a gamma-law equation of state, so we can check our solution against these exact solutions. See Toro's book for details on this problem and the exact Riemann solver.

Because it is one-dimensional, we run it in narrow domains in the x- or y-directions. It can be run as:

```
./pyro.py compressible sod inputs.sod.x
./pyro.py compressible sod inputs.sod.y
```

A simple script, `sod_compare.py` in `analysis/` will read a pyro output file and plot the solution over the exact Sod solution. Below we see the result for a Sod run with 128 points in the x-direction, gamma = 1.4, and run until t = 0.2 s.



We see excellent agreement for all quantities. The shock wave is very steep, as expected. The contact wave is smeared out over ~5 zones—this is discussed in the notes above, and can be improved in the PPM method with contact steepening.

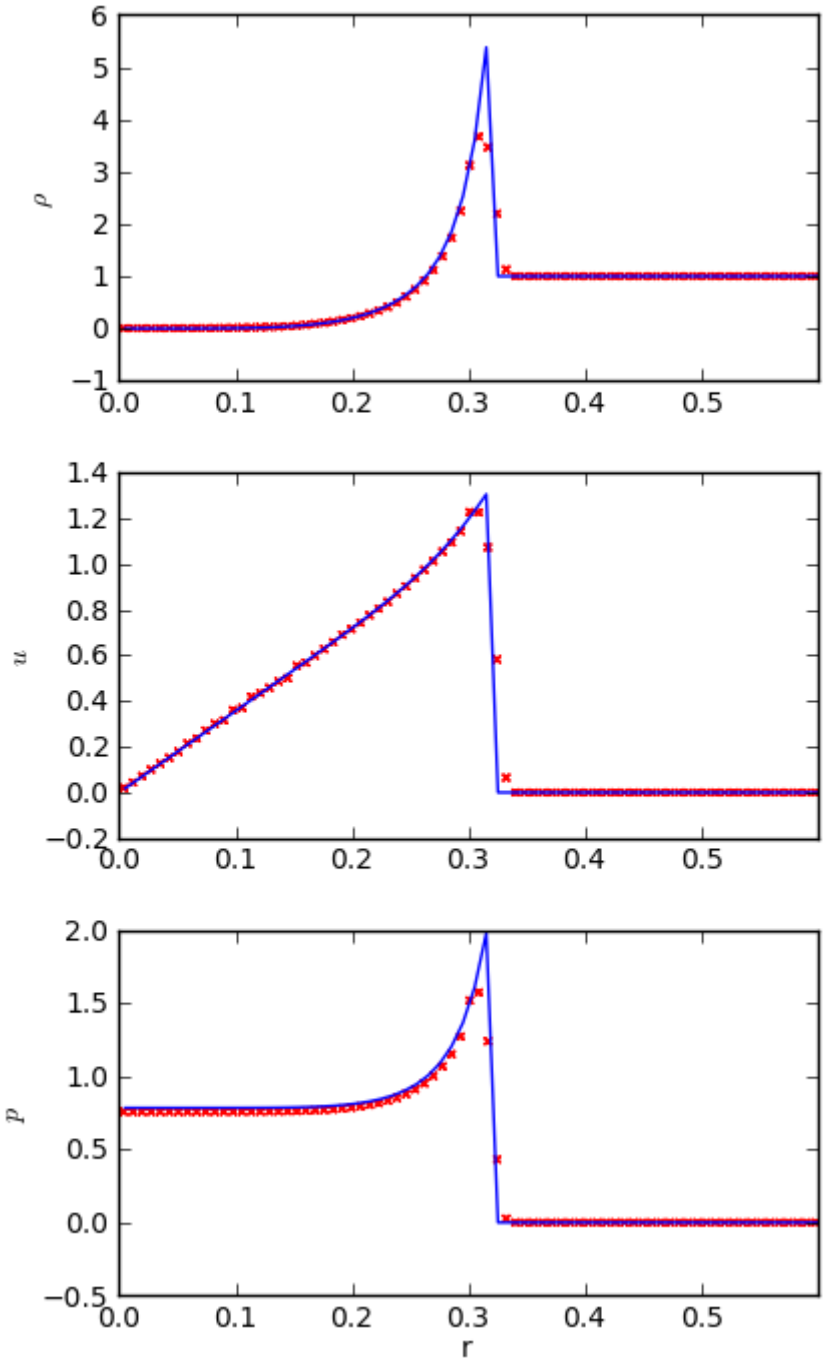
10.5.2 Sedov

The Sedov blast wave problem is another standard test with an analytic solution (Sedov 1959). A lot of energy is point into a point in a uniform medium and a blast wave propagates outward. The Sedov problem is run as:

```
./pyro.py compressible sedov inputs.sedov
```

The video below shows the output from a 128 x 128 grid with the energy put in a radius of 0.0125 surrounding the center of the domain. A gamma-law EOS with gamma = 1.4 is used, and we run until 0.1

We see some grid effects because it is hard to initialize a small circular explosion on a rectangular grid. To compare to the analytic solution, we need to radially bin the data. Since this is a 2-d explosion, the physical geometry it represents is a cylindrical blast wave, so we compare to Sedov's cylindrical solution. The radial binning is done with the `sedov_compare.py` script in `analysis/`

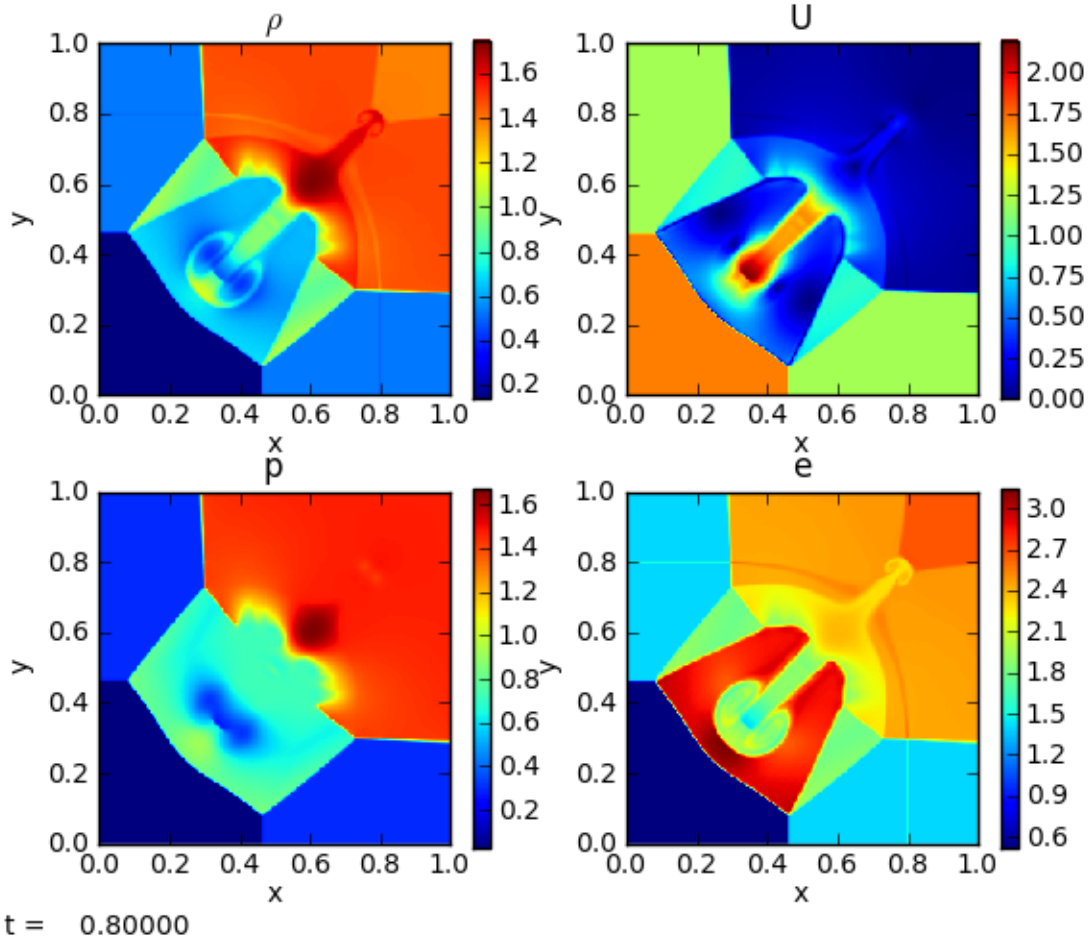


This shows good agreement with the analytic solution.

10.5.3 quad

The quad problem sets up different states in four regions of the domain and watches the complex interfaces that develop as shocks interact. This problem has appeared in several places (and a detailed investigation is online by Pawel Artymowicz). It is run as:

```
./pyro.py compressible quad inputs.quad
```



10.5.4 rt

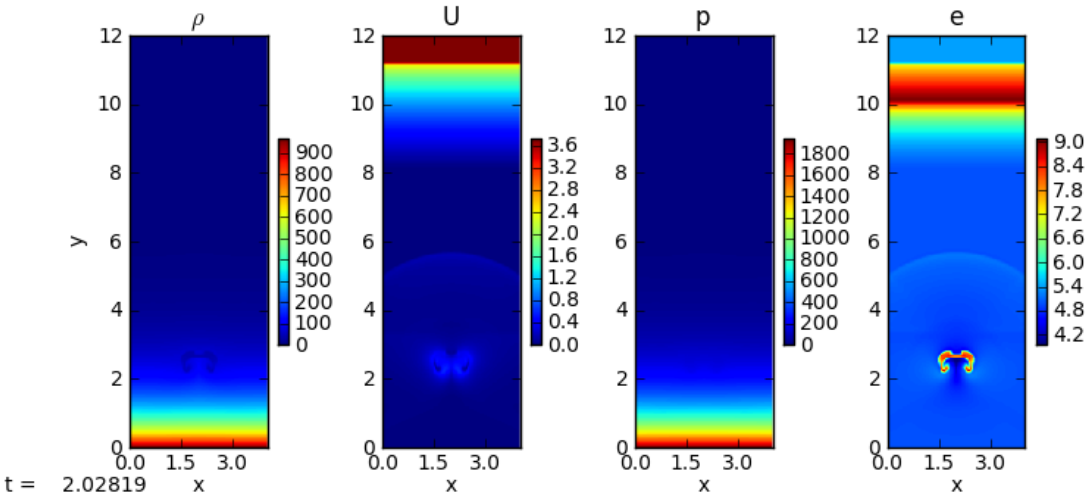
The Rayleigh-Taylor problem puts a dense fluid over a lighter one and perturbs the interface with a sinusoidal velocity. Hydrostatic boundary conditions are used to ensure any initial pressure waves can escape the domain. It is run as:

```
./pyro.py compressible rt inputs.rt
```

10.5.5 bubble

The bubble problem initializes a hot spot in a stratified domain and watches it buoyantly rise and roll up. This is run as:

```
./pyro.py compressible bubble inputs.bubble
```



The shock at the top of the domain is because we cut off the stratified atmosphere at some low density and the resulting material above that rains down on our atmosphere. Also note the acoustic signal propagating outward from the bubble (visible in the U and e panels).

10.6 Exercises

10.6.1 Explorations

- Measure the growth rate of the Rayleigh-Taylor instability for different wavenumbers.
- There are multiple Riemann solvers in the compressible algorithm. Run the same problem with the different Riemann solvers and look at the differences. Toro's text is a good book to help understand what is happening.
- Run the problems with and without limiting—do you notice any overshoots?

10.6.2 Extensions

- Limit on the characteristic variables instead of the primitive variables. What changes do you see? (the notes show how to implement this change.)
- Add passively advected species to the solver.
- Add an external heating term to the equations.
- Add 2-d axisymmetric coordinates (r-z) to the solver. This is discussed in the notes. Run the Sedov problem with the explosion on the symmetric axis—now the solution will behave like the spherical sedov explosion instead of the cylindrical explosion.
- Swap the piecewise linear reconstruction for piecewise parabolic (PPM). The notes and the Miller and Colella paper provide a good basis for this. Research the Roe Riemann solver and implement it in pyro.

10.7 Going further

The compressible algorithm presented here is essentially the single-grid hydrodynamics algorithm used in the Castro code—an adaptive mesh radiation hydrodynamics code developed at CCSE/LBNL. Castro is freely available for download.

A simple, pure Fortran, 1-d compressible hydrodynamics code that does piecewise constant, linear, or parabolic (PPM) reconstruction is also available. See the hydro1d page.

CHAPTER

ELEVEN

COMPRESSIBLE SOLVER COMPARISONS

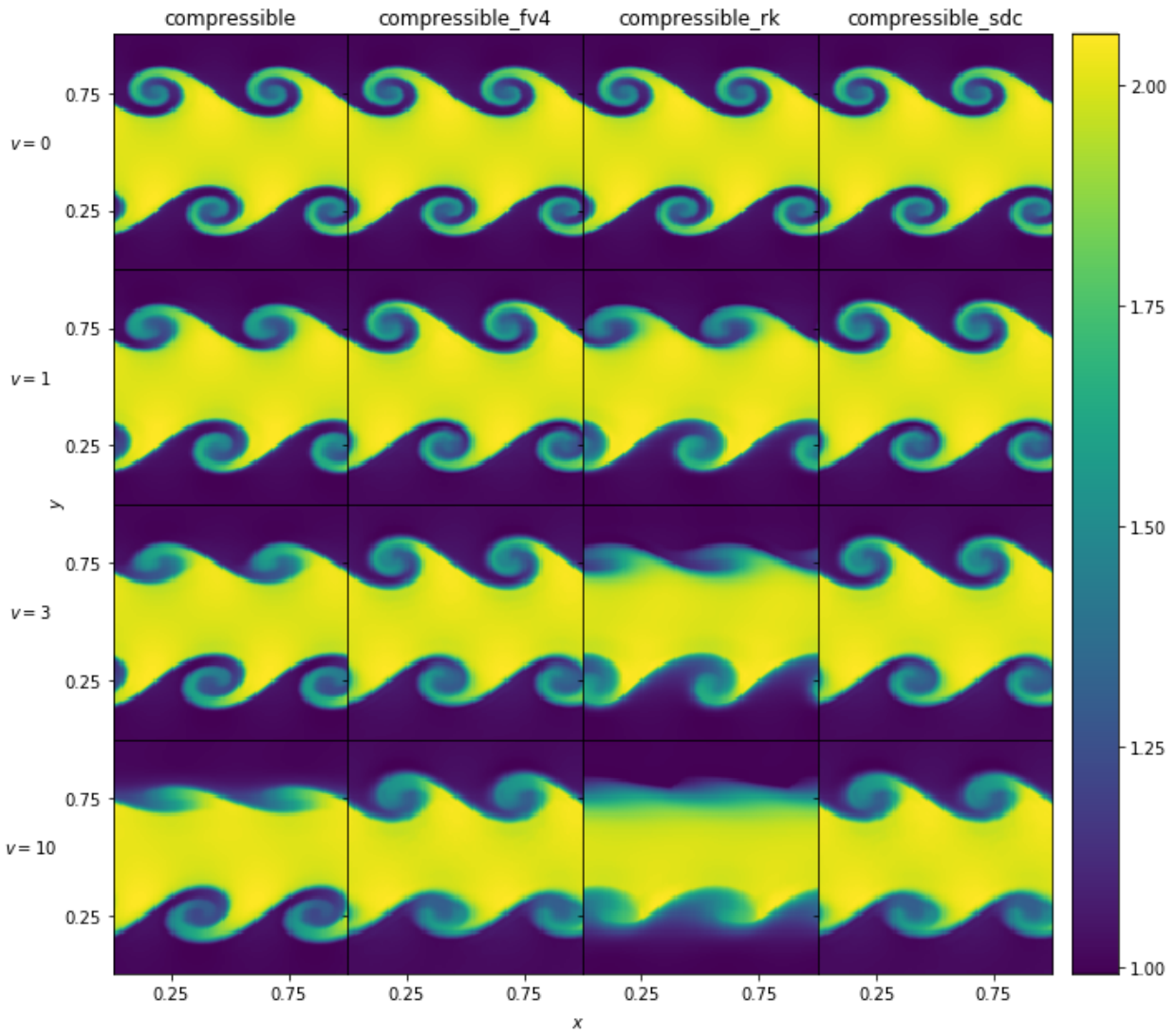
We run various problems run with the different compressible solvers in pyro (standard Riemann, Runge-Kutta, fourth order).

11.1 Kelvin-Helmholtz

The McNally Kelvin-Helmholtz problem sets up a heavier fluid moving in the negative x-direction sandwiched between regions of lighter fluid moving in the positive x-direction.

The image below shows the KH problem initialized with McNally's test. It ran on a 128 x 128 grid, with gamma = 1.7, and ran until t = 2.0. This is run with:

```
./pyro.py compressible kh inputs.kh kh.vbulk=0
./pyro.py compressible_rk kh inputs.kh kh.vbulk=0
./pyro.py compressible_fv4 kh inputs.kh kh.vbulk=0
./pyro.py compressible_sdc kh inputs.kh kh.vbulk=0
```

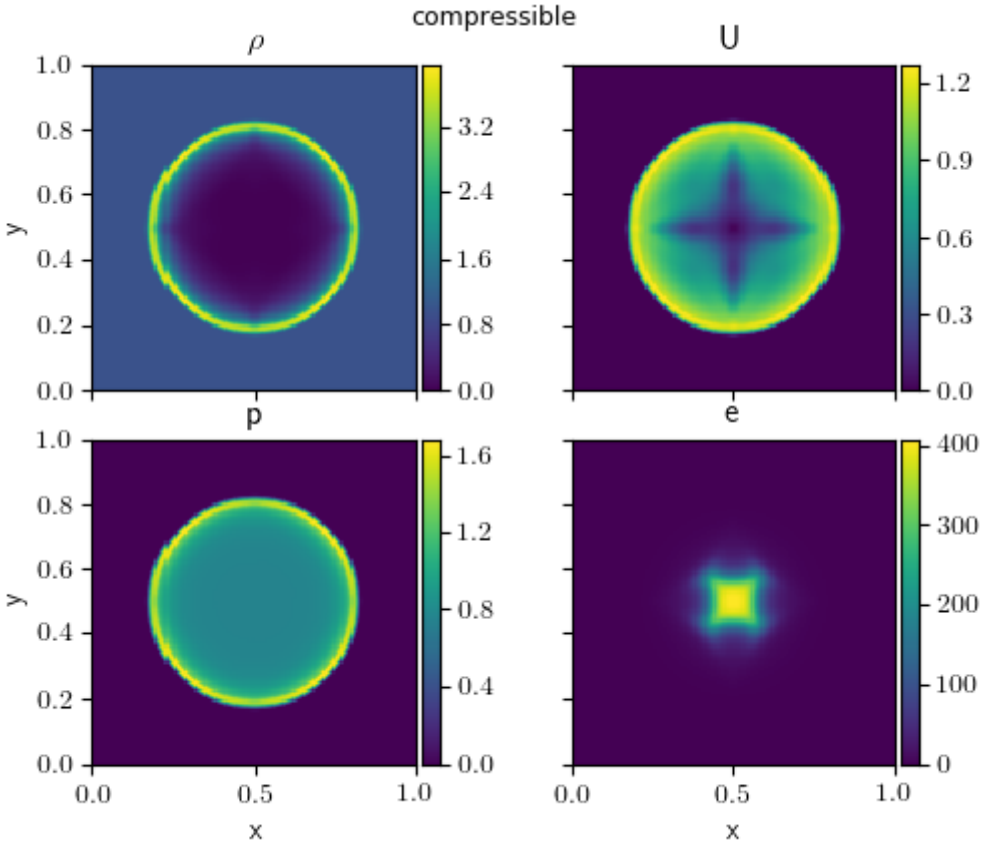


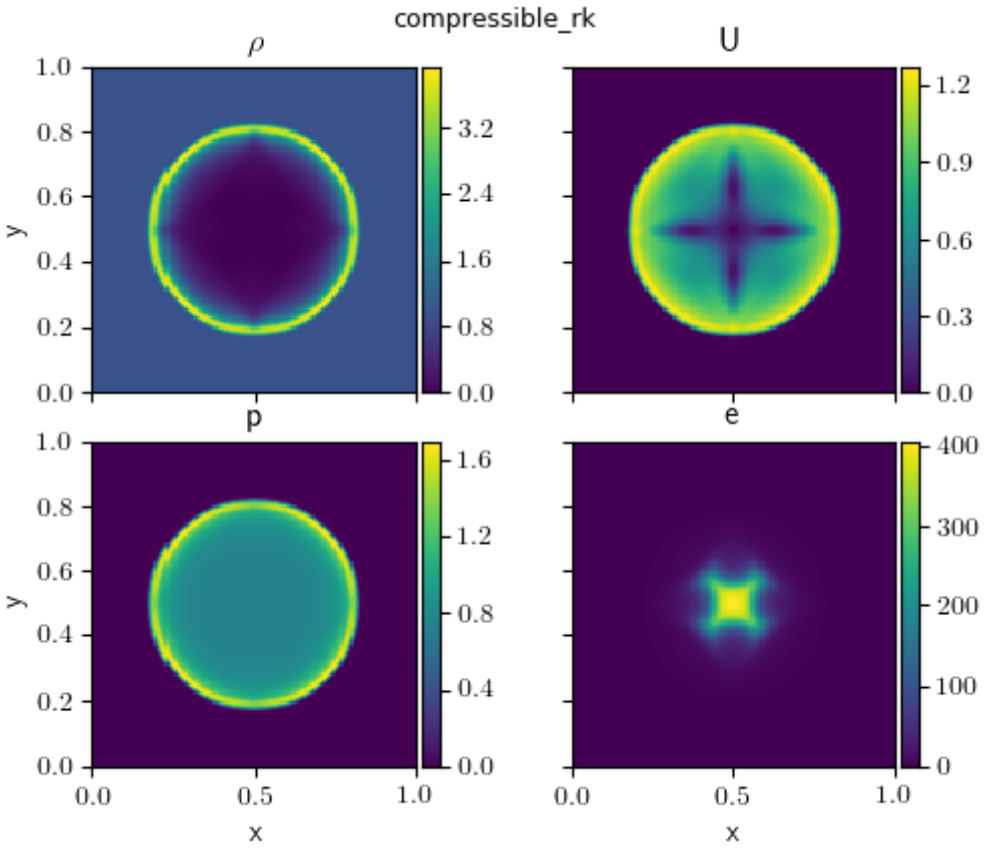
We vary the velocity in the positive y-direction (vbulk) to see how effective the solvers are at preserving the initial shape.

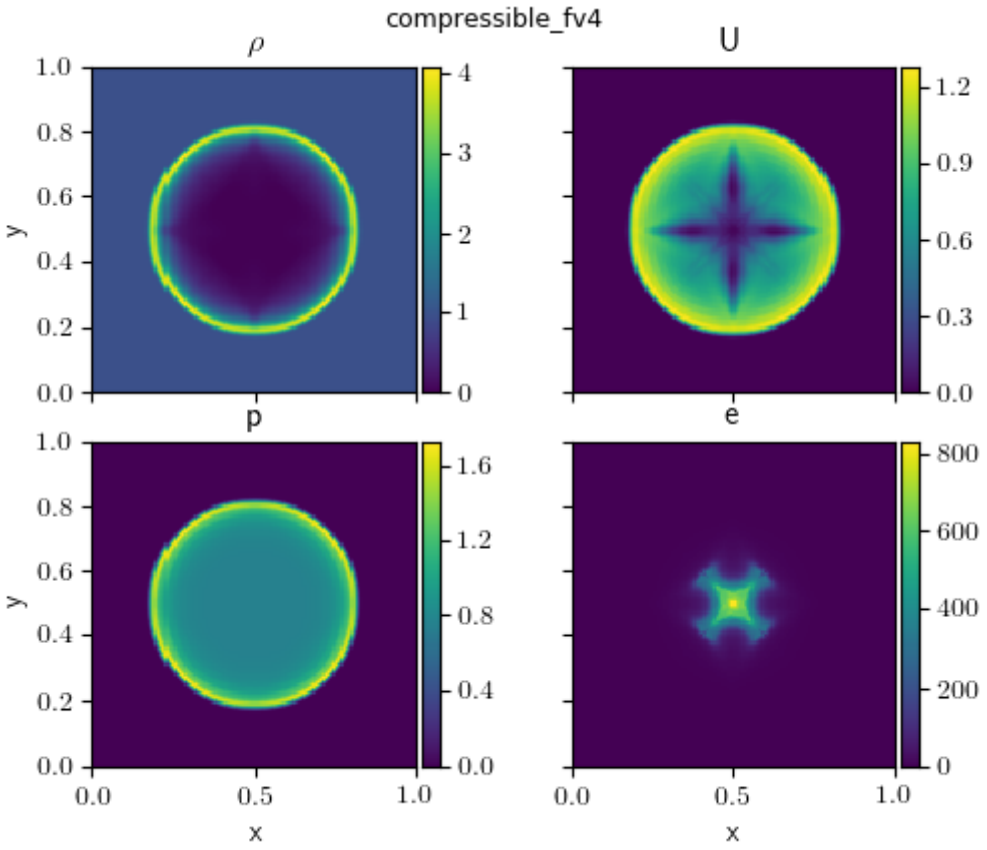
11.2 Sedov

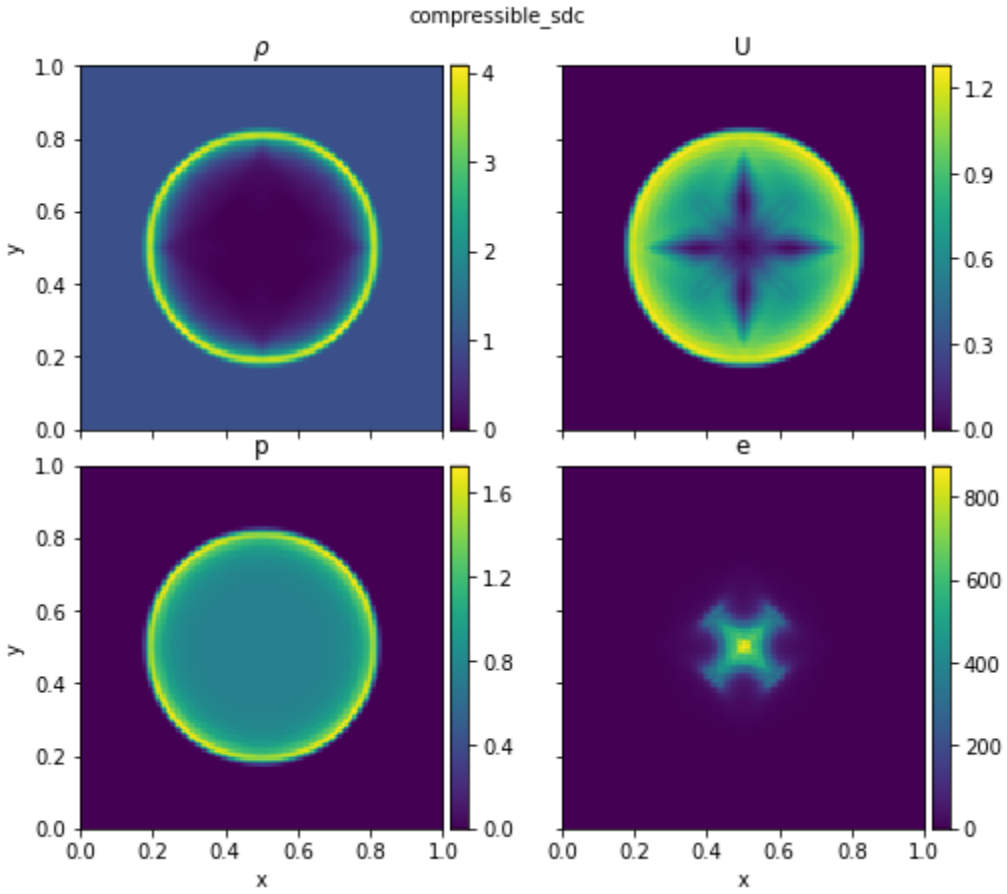
The Sedov problem ran on a 128 x 128 grid, with gamma = 1.4, and until t = 0.1, which can be run as:

```
./pyro.py compressible sedov inputs.sedov
./pyro.py compressible_rk sedov inputs.sedov
./pyro.py compressible_fv4 sedov inputs.sedov
./pyro.py compressible_sdc sedov inputs.sedov
```


compressible
ρ
U
p
e
y
x
1.0
0.8
0.6
0.4
0.2
0.0
0.5
3.2
2.4
1.6
1.2
0.9
0.3
400
300
200
100
0


compressible_rk
ρ
U
p
e
y
x
1.0
0.8
0.6
0.4
0.2
0.0
0.5
3.2
2.4
1.6
1.2
0.9
0.3
400
300
200
100
0

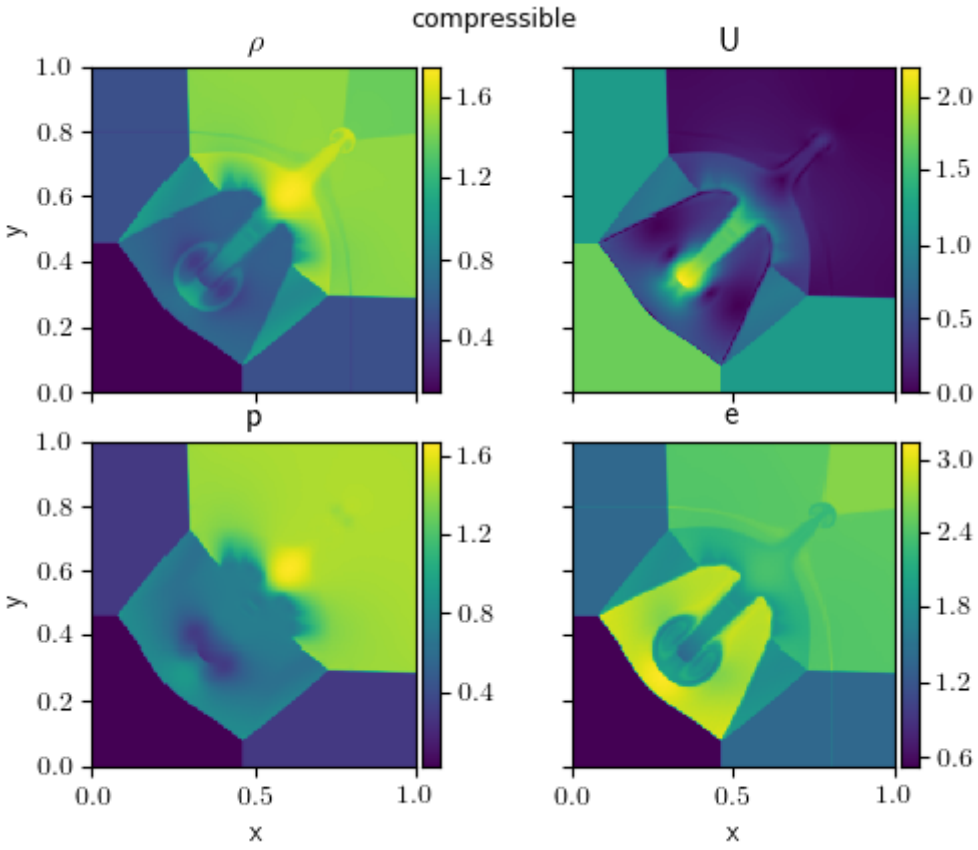

compressible_fv4
ρ
U
p
e
y
x
1.0
0.8
0.6
0.4
0.2
0.0
0.5
4
3
2
1
0
1.2
0.9
0.6
0.3
1.6
0.8
0.4
800
600
400
200

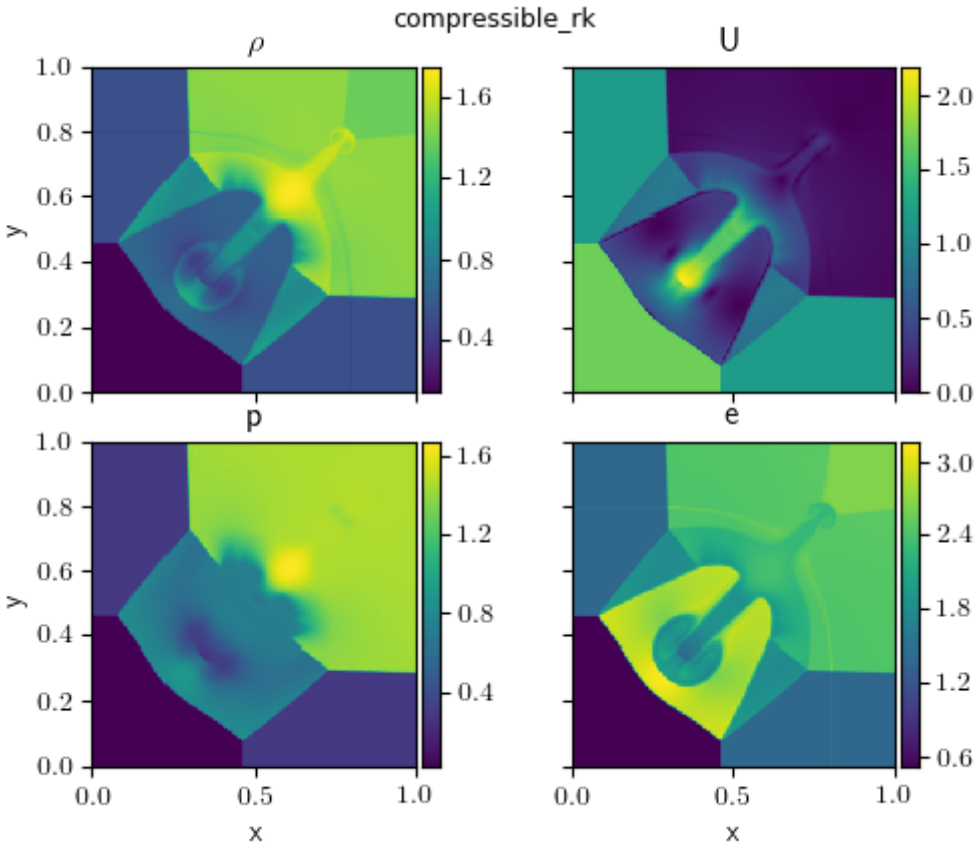


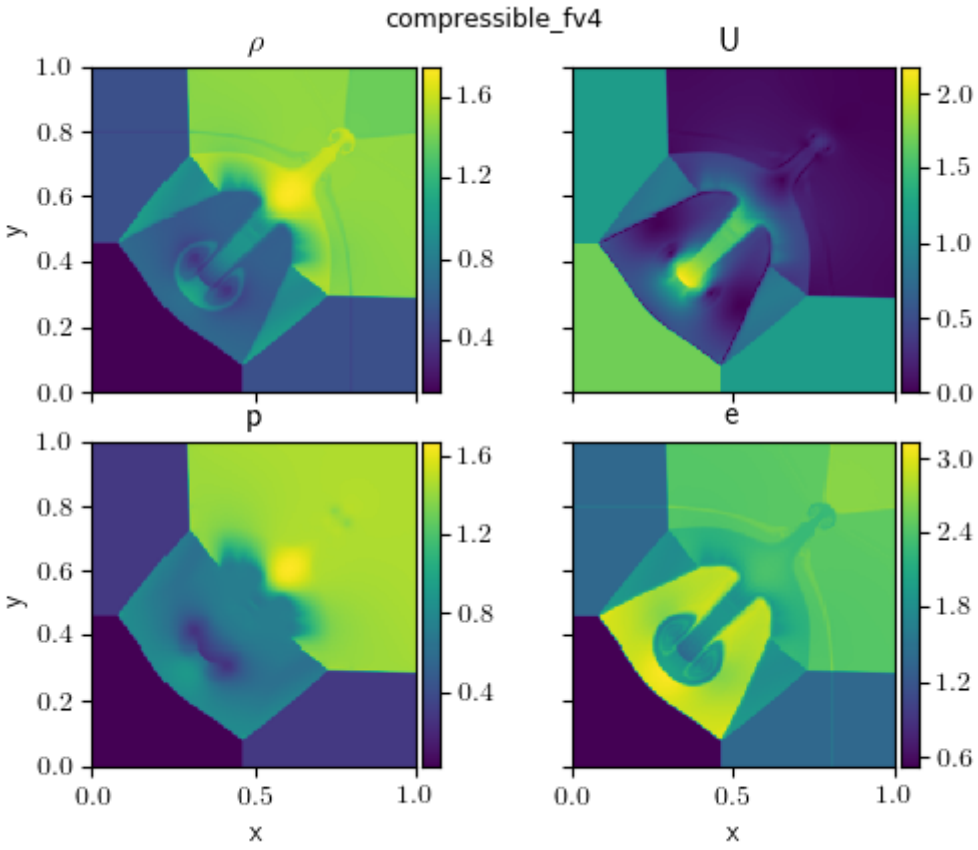
11.3 Quad

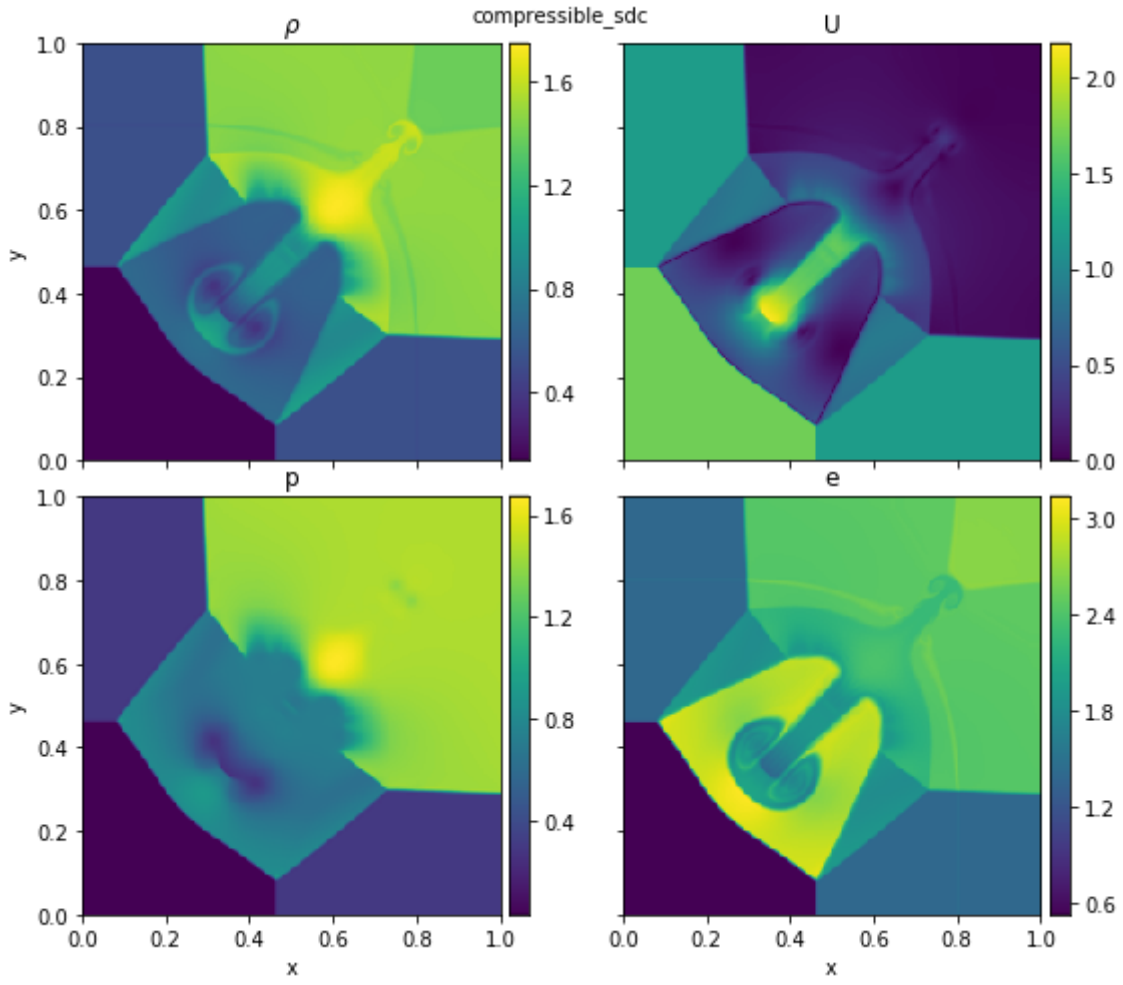
The quad problem ran on a 256 x 256 grid until t = 0.8, which can be run as:

```
./pyro.py compressible quad inputs.quad
./pyro.py compressible_rk quad inputs.quad
./pyro.py compressible_fv4 quad inputs.quad
./pyro.py compressible_sdc quad inputs.quad
```


compressible
ρ
U
p
e
y
x
1.0
0.8
0.6
0.4
0.2
0.0
0.5
1.6
1.2
0.8
0.4
2.0
1.5
1.0
0.5
0.0
3.0
2.4
1.8
1.2
0.6


compressible_rk
ρ
U
p
e
y
x
0.0
0.2
0.4
0.6
0.8
1.0
0.5
0.4
0.8
1.2
1.6
0.5
1.0
1.5
2.0
0.6
1.2
1.8
2.4
3.0

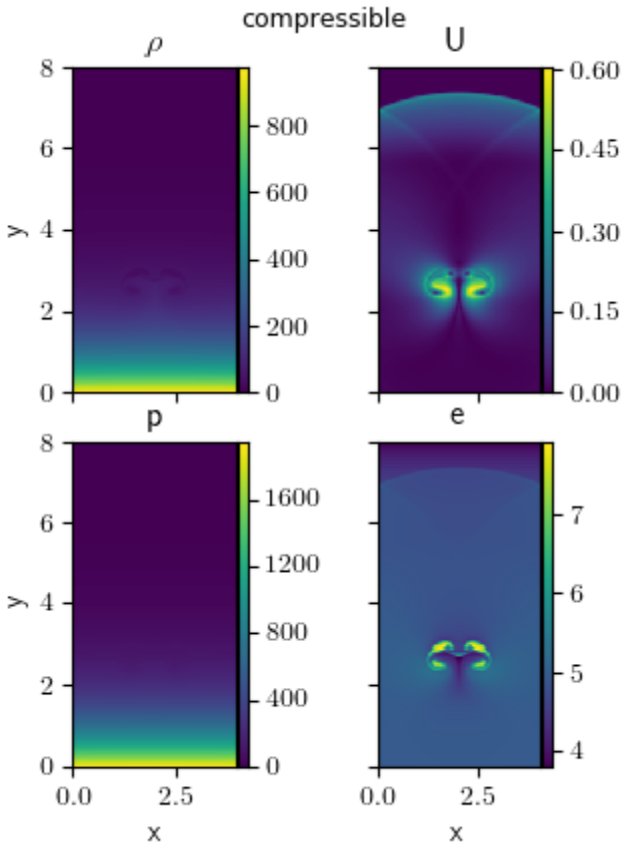

compressible_fv4
ρ
U
p
e
y
x
1.0
0.8
0.6
0.4
0.2
0.0
0.5
1.6
1.2
0.8
0.4
2.0
1.5
1.0
0.5
0.0
3.0
2.4
1.8
1.2
0.6

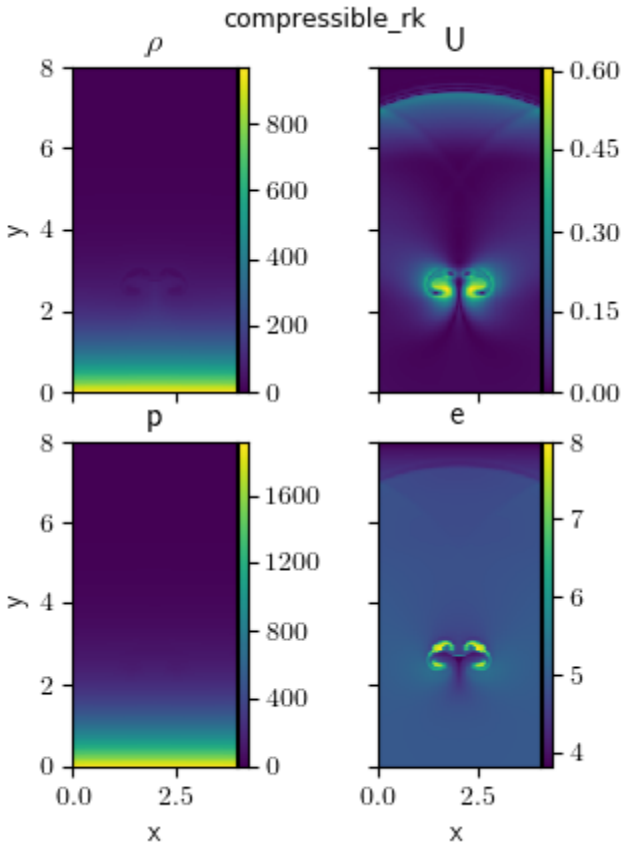


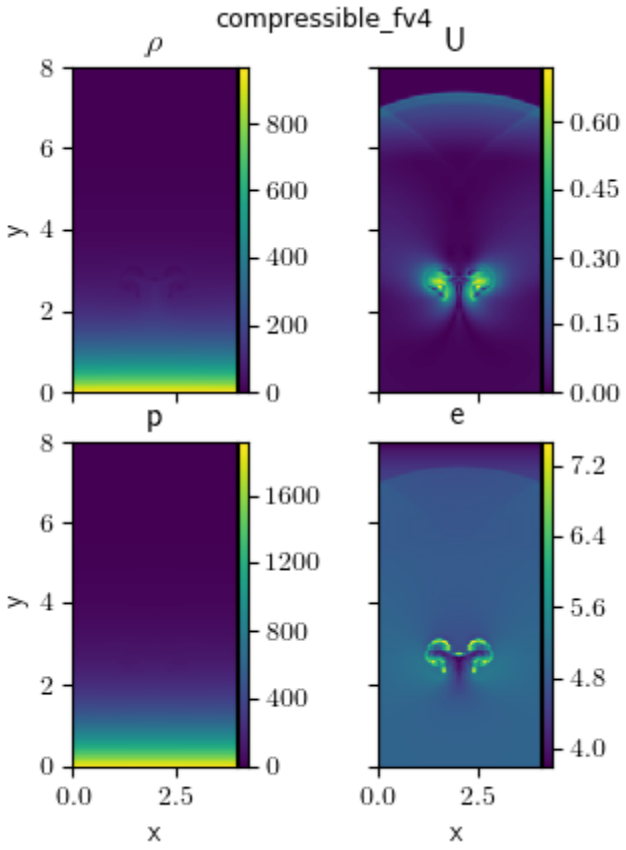
11.4 Bubble

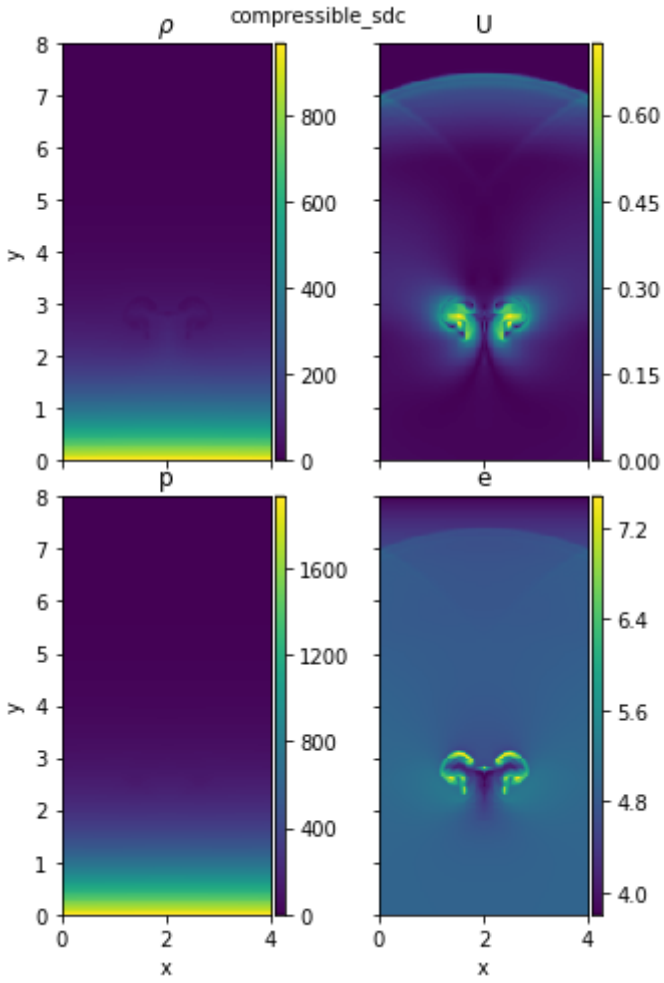
The bubble problem ran on a 128 x 256 grid until t = 3.0, which can be run as:

```
./pyro.py compressible bubble inputs.bubble
./pyro.py compressible_rk bubble inputs.bubble
./pyro.py compressible_fv4 bubble inputs.bubble
./pyro.py compressible_sdc bubble inputs.bubble
```


compressible
ρ
U
p
e
y
x
0
2
4
6
8
0.0
2.5
0
200
400
600
800
0.00
0.15
0.30
0.45
0.60
0
400
800
1200
1600
4
5
6
7


compressible_rk
ρ
U
p
e
y
x
0
2
4
6
8
0.0
2.5
0
200
400
600
800
0.00
0.15
0.30
0.45
0.60
0
400
800
1200
1600
4
5
6
7
8

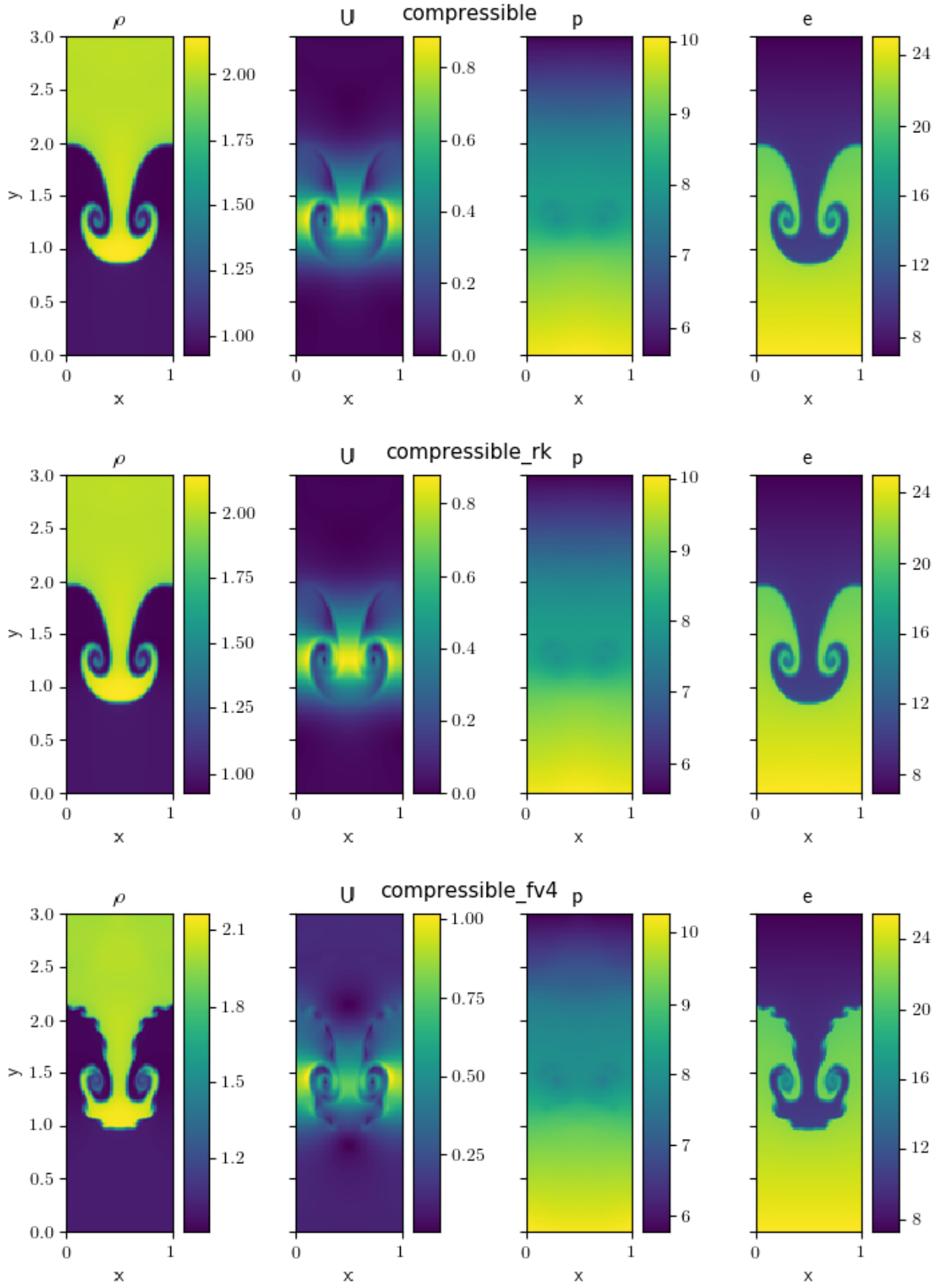

compressible_fv4
ρ
U
p
e
y
x
0
2
4
6
8
0.0
2.5
800
600
400
200
0
0.60
0.45
0.30
0.15
0.00
1600
1200
800
400
0
7.2
6.4
5.6
4.8
4.0

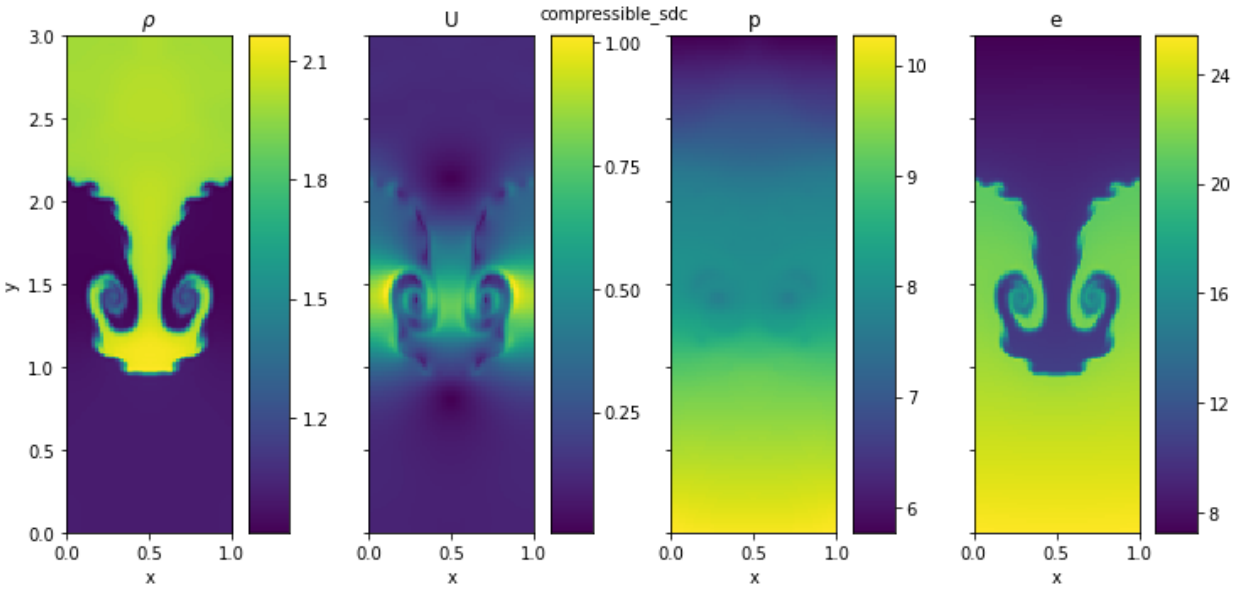


11.5 Rayleigh-Taylor

The Rayleigh-Taylor problem ran on a 64 x 192 grid until t = 3.0, which can be run as:

```
./pyro.py compressible rt inputs.rt
./pyro.py compressible_rk rt inputs.rt
./pyro.py compressible_fv4 rt inputs.rt
./pyro.py compressible_sdc rt inputs.rt
```


compressible
ρ
U
p
e
compressible_rk
ρ
U
p
e
compressible_fv4
ρ
U
p
e
x
y


ρ
U
compressible_sdc
p
e
y
x
x
x
x
x
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
2.1
1.8
1.5
1.2
1.00
0.75
0.50
0.25
10
9
8
7
6
24
20
16
12
8

CHAPTER

TWELVE

MULTIGRID SOLVERS

pyro solves elliptic problems (like Laplace's equation or Poisson's equation) through multigrid. This accelerates the convergence of simple relaxation by moving the solution down and up through a series of grids. Chapter 9 of the pdf notes gives an introduction to solving elliptic equations, including multigrid.

There are three solvers:

- The core solver, provided in the class `MG.CellCenterMG2d` solves constant-coefficient Helmholtz problems of the form $(\alpha - \beta\nabla^2)\phi = f$
- The class `variable_coeff_MG.VarCoeffCCMG2d` solves variable coefficient Poisson problems of the form $\nabla \cdot (\eta\nabla\phi) = f$. This class inherits the core functionality from `MG.CellCenterMG2d`.
- The class `general_MG.GeneralMG2d` solves a general elliptic equation of the form $\alpha\phi + \nabla\cdot(\beta\nabla\phi) + \gamma\cdot\nabla\phi = f$. This class inherits the core functionality from `MG.CellCenterMG2d`.

  This solver is the only one to support inhomogeneous boundary conditions.

We simply use V-cycles in our implementation, and restrict ourselves to square grids with zoning a power of 2.

The multigrid solver is not controlled through pyro.py since there is no time-dependence in pure elliptic problems. Instead, there are a few scripts in the multigrid/ subdirectory that demonstrate its use.

12.1 Examples

12.1.1 multigrid test

A basic multigrid test is run as (using a path relative to the root of the `pyro2` repository):

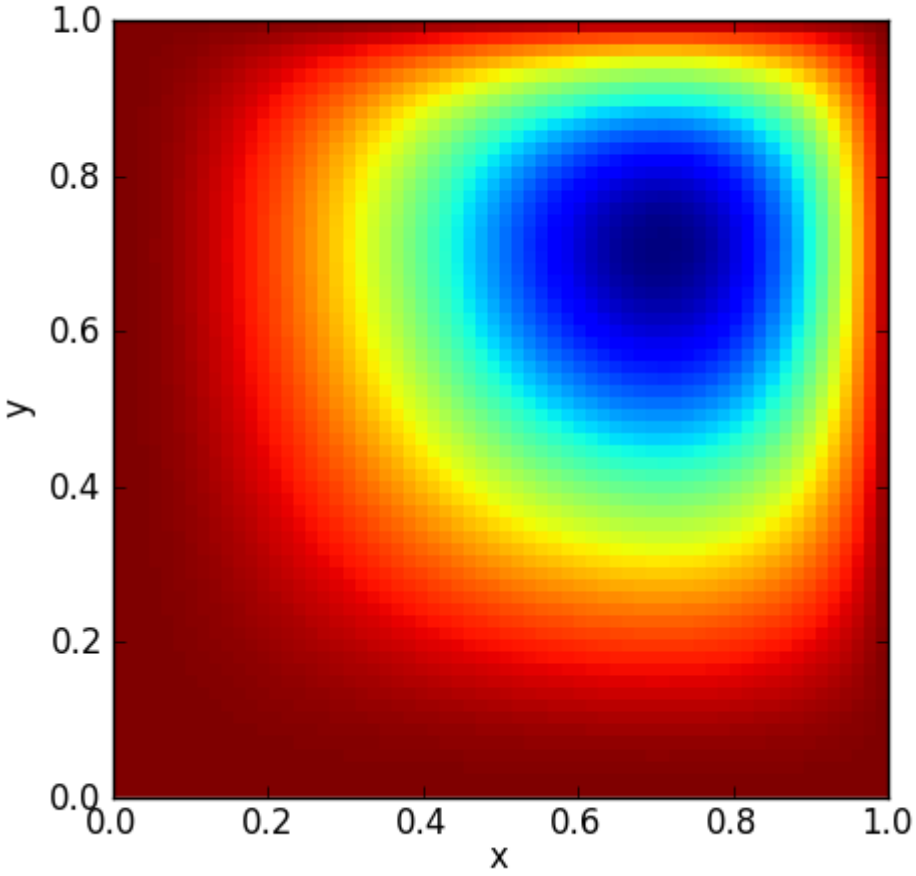
```
./examples/multigrid/mg_test_simple.py
```

The `mg_test_simple.py` script solves a Poisson equation with a known analytic solution. This particular example comes from the text *A Multigrid Tutorial, 2nd Ed.*, by Briggs. The example is:

$$u_{xx} + u_{yy} = -2\left[(1-6x^2)y^2(1-y^2) + (1-6y^2)x^2(1-x^2)\right]$$

on $[0,1] \times [0,1]$ with $u = 0$ on the boundary.

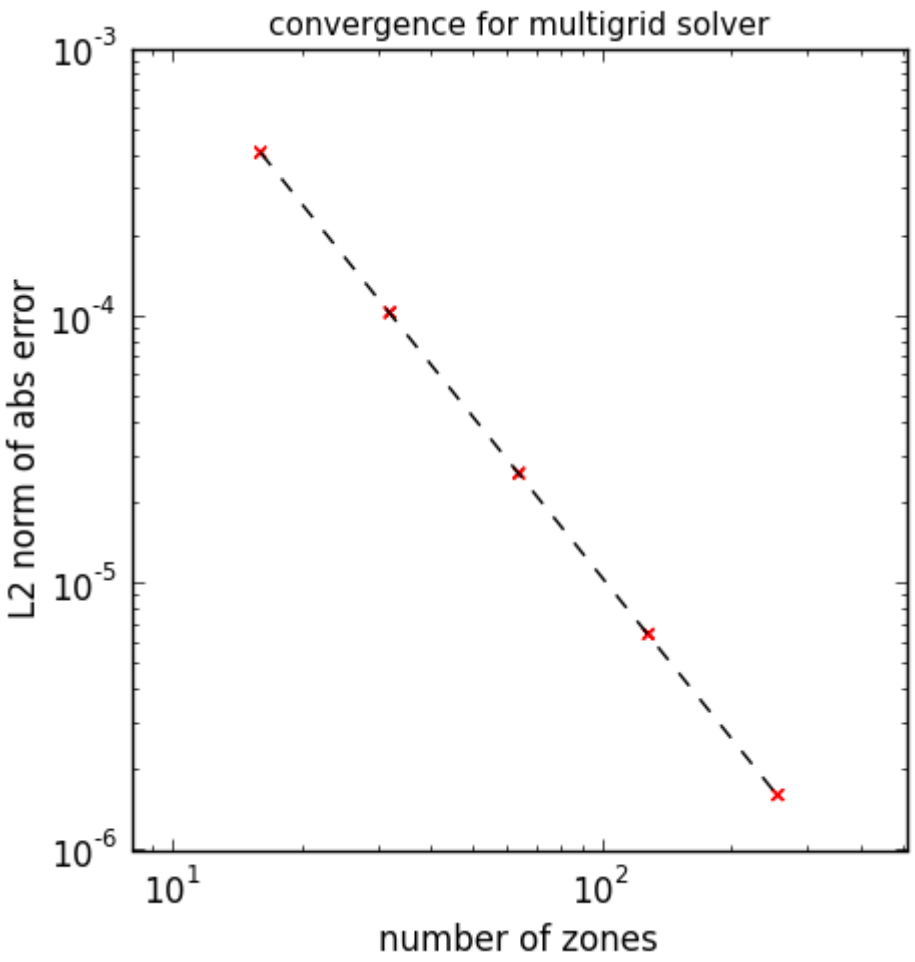
The solution to this is shown below.



Since this has a known analytic solution:

$$u(x, y) = (x^2 - x^4)(y^4 - y^2)$$

We can assess the convergence of our solver by running at a variety of resolutions and computing the norm of the error with respect to the analytic solution. This is shown below:



The dotted line is 2nd order convergence, which we match perfectly.

The movie below shows the smoothing at each level to realize this solution:

You can run this example locally by running the `mg_vis.py` script:

```
./examples/multigrid/mg_vis.py
```

12.1.2 projection

Another example uses multigrid to extract the divergence free part of a velocity field. This is run as:

```
./examples/multigrid/project_periodic.py
```

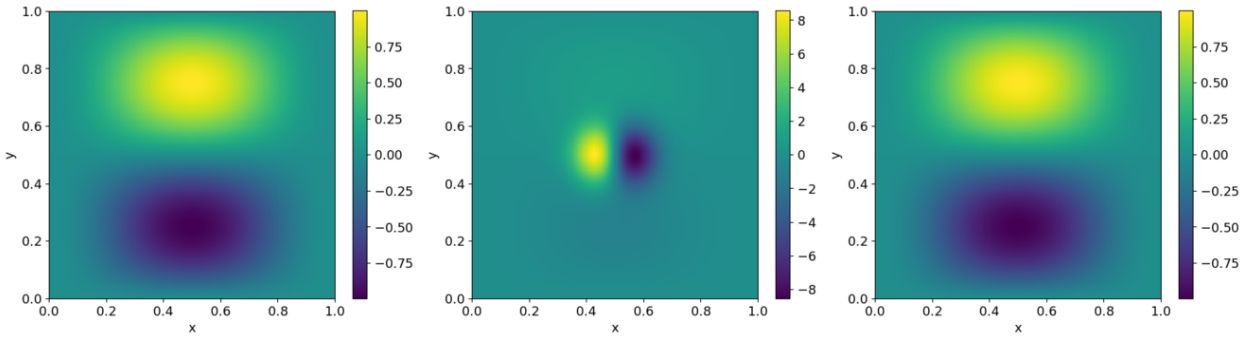
Given a vector field, U, we can decompose it into a divergence free part, U_d, and the gradient of a scalar, ϕ:

$$U = U_d + \nabla\phi$$

We can project out the divergence free part by taking the divergence, leading to an elliptic equation:

$$\nabla^2\phi = \nabla \cdot U$$

The `project-periodic.py` script starts with a divergence free velocity field, adds to it the gradient of a scalar, and then projects it to recover the divergence free part. The error can found by comparing the original velocity field to the recovered field. The results are shown below:



Left is the original u velocity, middle is the modified field after adding the gradient of the scalar, and right is the recovered field.

12.2 Exercises

12.2.1 Explorations

- Try doing just smoothing, no multigrid. Show that it still converges second order if you use enough iterations, but that the amount of time needed to get a solution is much greater.

12.2.2 Extensions

- Implement inhomogeneous dirichlet boundary conditions
- Add a different bottom solver to the multigrid algorithm
- Make the multigrid solver work for non-square domains

CHAPTER

THIRTEEN

DIFFUSION

pyro solves the constant-conductivity diffusion equation:

$$\frac{\partial \phi}{\partial t} = k \nabla^2 \phi$$

This is done implicitly using multigrid, using the solver `diffusion`.

The diffusion equation is discretized using Crank-Nicolson differencing (this makes the diffusion operator time-centered) and the implicit discretization forms a Helmholtz equation solved by the pyro multigrid class. The main parameters that affect this solver are:

- section: [diffusion]

option	value	description
k	`1.0`	conductivity

- section: [driver]

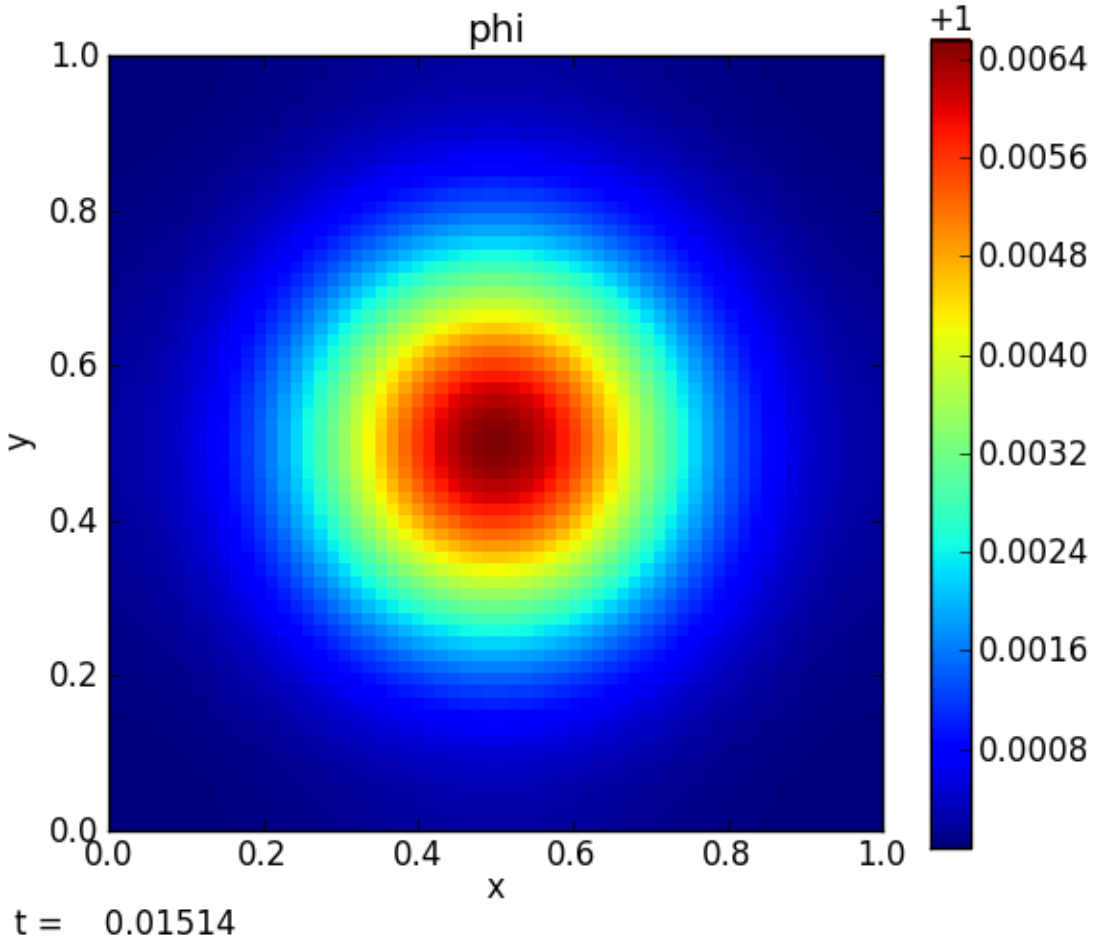
option	value	description
cfl	`0.8`	diffusion CFL number

13.1 Examples

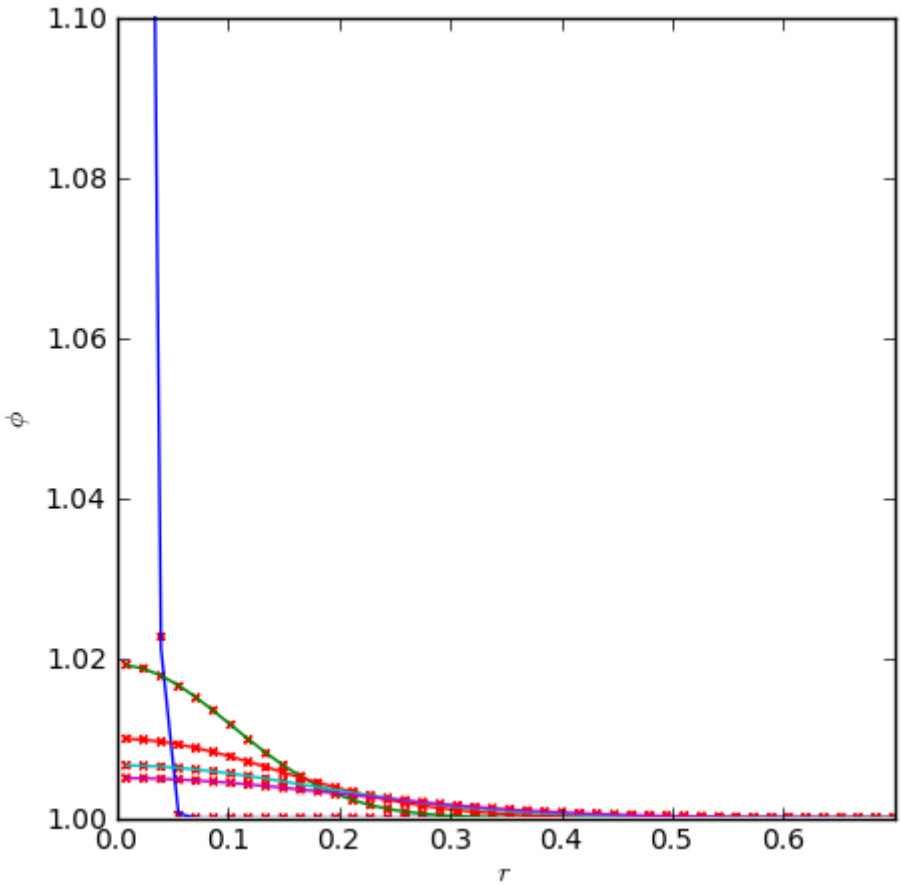
13.1.1 gaussian

The gaussian problem initializes a strongly peaked Gaussian centered in the domain. The analytic solution for this shows that the profile remains a Gaussian, with a changing width and peak. This allows us to compare our solver to the analytic solution. This is run as:

```
./pyro.py diffusion gaussian inputs.gaussian
```



The above figure shows the scalar field after diffusing significantly from its initial strongly peaked state. We can compare to the analytic solution by making radial profiles of the scalar. The plot below shows the numerical solution (red points) overplotted on the analytic solution (solid curves) for several different times. The y-axis is restricted in range to bring out the detail at later times.



13.2 Exercises

The best way to learn these methods is to play with them yourself. The exercises below are suggestions for explorations and features to add to the advection solver.

13.2.1 Explorations

- Test the convergence of the solver by varying the resolution and comparing to the analytic solution.
- How does the solution error change as the CFL number is increased well above 1?
- Setup some other profiles and experiment with different boundary conditions.

13.2.2 Extensions

- Switch from Crank-Nicolson (2nd order in time) to backward's Euler (1st order in time) and compare the solution and convergence. This should only require changing the source term and coefficents used in setting up the multigrid solve. It does not require changes to the multigrid solver itself.
- Implement a non-constant coefficient diffusion solver—note: this will require improving the multigrid solver.

CHAPTER

FOURTEEN

INCOMPRESSIBLE HYDRODYNAMICS SOLVER

pyro's incompressible solver solves:

$$\frac{\partial U}{\partial t} + U \cdot \nabla U + \nabla p = 0$$
$$\nabla \cdot U = 0$$

The algorithm combines the Godunov/advection features used in the advection and compressible solver together with multigrid to enforce the divergence constraint on the velocities.

Here we implement a cell-centered approximate projection method for solving the incompressible equations. At the moment, only periodic BCs are supported.

The main parameters that affect this solver are:

- section: [driver]

option	value	description
cfl	`0.8`	

- section: [incompressible]

option	value	description
limiter	`2`	limiter (0 = none, 1 = 2nd order, 2 = 4th order)
proj_type	`2`	what are we projecting? 1 includes -Gp term in U*

- section: [particles]

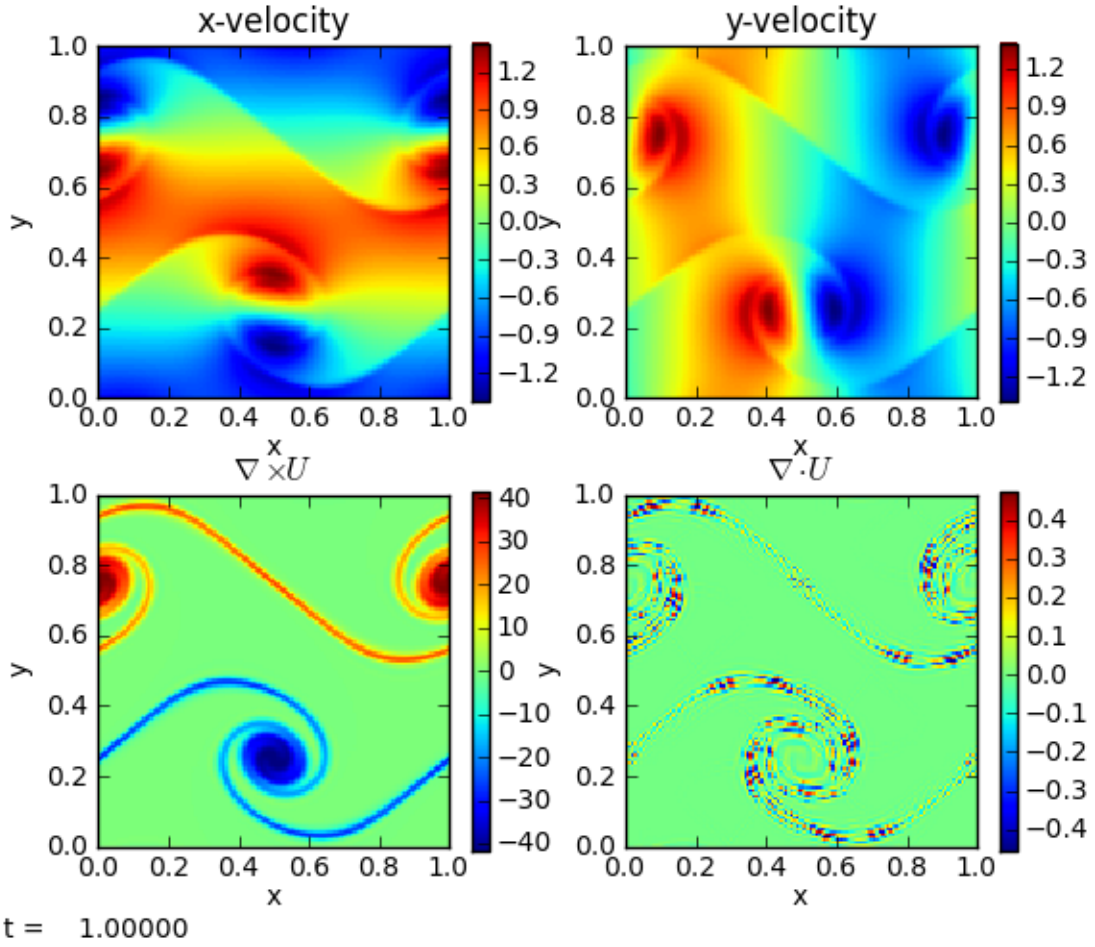
option	value	description
do_particles	`0`	
particle_generator	`grid`	

14.1 Examples

14.1.1 shear

The shear problem initializes a shear layer in a domain with doubly-periodic boundaries and looks at the development of two vortices as the shear layer rolls up. This problem was explored in a number of papers, for example, Bell, Colella, & Glaz (1989) and Martin & Colella (2000). This is run as:

```
./pyro.py incompressible shear inputs.shear
```



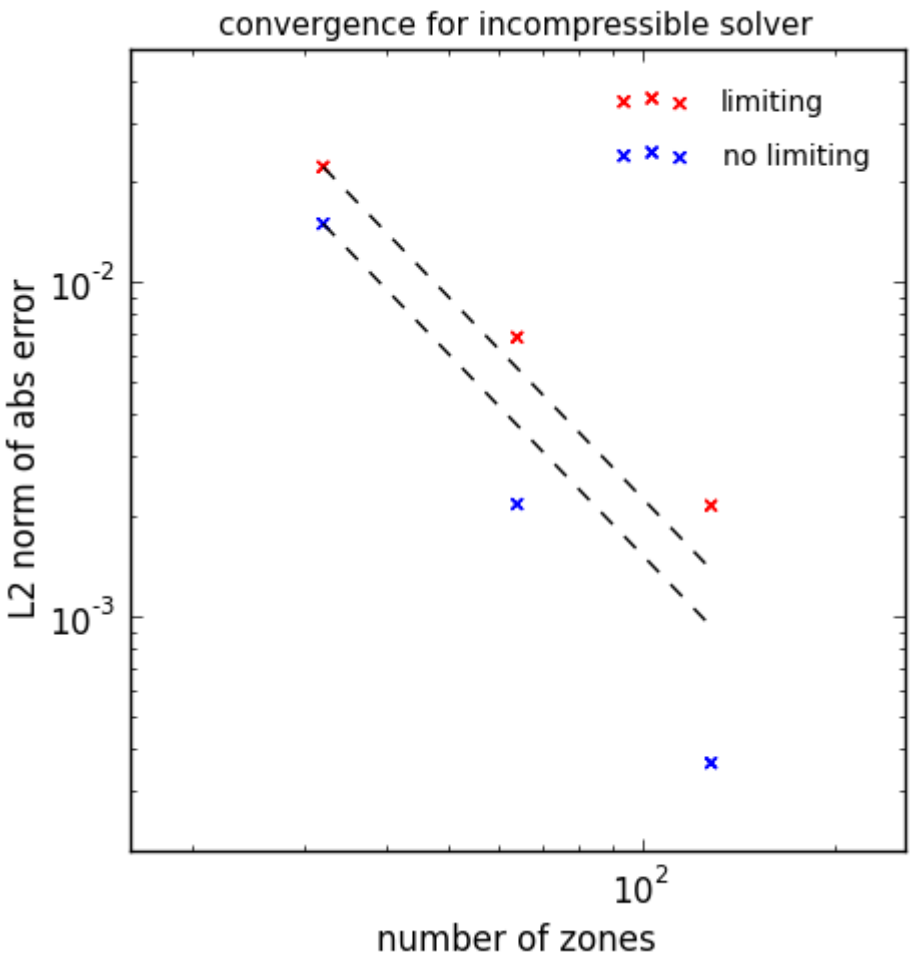
The vorticity panel (lower left) is what is usually shown in papers. Note that the velocity divergence is not zero—this is because we are using an approximate projection.

14.1.2 convergence

The convergence test initializes a simple velocity field on a periodic unit square with known analytic solution. By evolving at a variety of resolutions and comparing to the analytic solution, we can measure the convergence rate of the algorithm. The particular set of initial conditions is from Minion (1996). Limiting can be disabled by adding `incompressible.limiter=0` to the run command. The basic set of tests shown below are run as:

```
./pyro.py incompressible converge inputs.converge.32 vis.dovis=0
./pyro.py incompressible converge inputs.converge.64 vis.dovis=0
./pyro.py incompressible converge inputs.converge.128 vis.dovis=0
```

The error is measured by comparing with the analytic solution using the routine `incomp_converge_error.py` in `analysis/`.



The dashed line is second order convergence. We see almost second order behavior with the limiters enabled and slightly better than second order with no limiting.

14.2 Exercises

14.2.1 Explorations

- Disable the MAC projection and run the converge problem—is the method still 2nd order?
- Disable all projections—does the solution still even try to preserve $\nabla \cdot U = 0$?
- Experiment with what is projected. Try projecting U_t to see if that makes a difference.

14.2.2 Extensions

- Switch the final projection from a cell-centered approximate projection to a nodal projection. This will require writing a new multigrid solver that operates on nodal data.
- Add viscosity to the system. This will require doing 2 parabolic solves (one for each velocity component). These solves will look like the diffusion operation, and will update the provisional velocity field.
- Switch to a variable density system. This will require adding a mass continuity equation that is advected and switching the projections to a variable-coeffient form (since now enters).

14.3 Going further

The incompressible algorithm presented here is a simplified version of the projection methods used in the Maestro low Mach number hydrodynamics code. Maestro can do variable-density incompressible, anelastic, and low Mach number stratified flows in stellar (and terrestrial) environments in close hydrostatic equilibrium.

CHAPTER

FIFTEEN

LOW MACH NUMBER HYDRODYNAMICS SOLVER

pyro's low Mach hydrodynamics solver is designed for atmospheric flows. It captures the effects of stratification on a fluid element by enforcing a divergence constraint on the velocity field. The governing equations are:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho U) = 0$$

$$\frac{\partial U}{\partial t} + U \cdot \nabla U + \frac{\beta_0}{\rho} \nabla \left(\frac{p'}{\beta_0} \right) = \frac{\rho'}{\rho} g$$

$$\nabla \cdot (\beta_0 U) = 0$$

with $\nabla p_0 = \rho_0 g$ and $\beta_0 = p_0^{1/\gamma}$.

As with the incompressible solver, we implement a cell-centered approximate projection method.

The main parameters that affect this solver are:

- section: [driver]

option	value	description
cfl	`0.8`	

- section: [eos]

option	value	description
gamma	`1.4`	pres = rho ener (gamma - 1)

- section: [lm-atmosphere]

option	value	description
limiter	`2`	limiter (0 = none, 1 = 2nd order, 2 = 4th order)
proj_type	`2`	what are we projecting? 1 includes -Gp term in U*
grav	`-2.0`	

15.1 Examples

15.1.1 bubble

The bubble problem places a buoyant bubble in a stratified atmosphere and watches the development of the roll-up due to shear as it rises. This is run as:

```
./pyro.py lm_atm bubble inputs.bubble
```

CHAPTER

SIXTEEN

SHALLOW WATER SOLVER

The (augmented) shallow water equations take the form:

$$\frac{\partial h}{\partial t} + \nabla \cdot (hU) = 0$$

$$\frac{\partial (hU)}{\partial t} + \nabla \cdot (hUU) + \frac{1}{2} g \nabla h^2 = 0$$

$$\frac{\partial (h\psi)}{\partial t} + \nabla \cdot (hU\psi) = 0$$

with h is the fluid height, U the fluid velocity, g the gravitational acceleration and $\psi = \psi(x, t)$ represents some passive scalar.

The implementation here has flattening at shocks and a choice of Riemann solvers.

The main parameters that affect this solver are:

- section: [driver]

option	value	description
cfl	`0.8`	

- section: [particles]

option	value	description
do_particles	`0`	
particle_generator	`grid`	

- section: [swe]

option	value	description
use_flattening	`0`	apply flattening at shocks (1)
cvisc	`0.1`	artifical viscosity coefficient
limiter	`2`	limiter (0 = none, 1 = 2nd order, 2 = 4th order)
grav	`1.0`	gravitational acceleration (in y-direction)
riemann	`Roe`	HLLC or Roe

16.1 Example problems

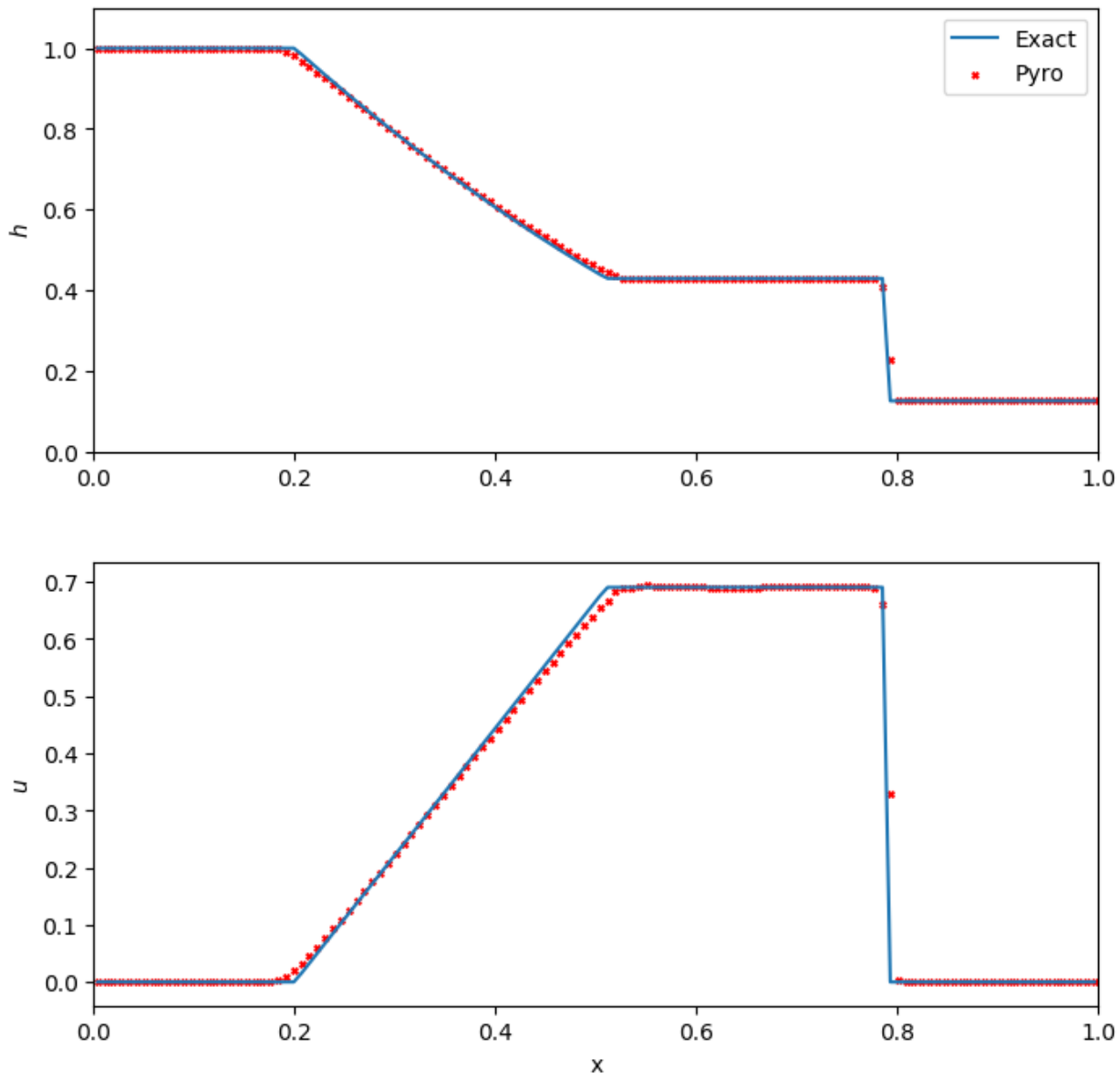
16.1.1 dam

The dam break problem is a standard hydrodynamics problem, analagous to the Sod shock tube problem in compressible hydrodynamics. It considers a one-multidimensional problem of two regions of fluid at different heights, initially separated by a dam. The problem then models the evolution of the system when this dam is removed. As for the Sod problem, there exists an exact solution for the dam break probem, so we can check our solution against the exact solutions. See Toro's shallow water equations book for details on this problem and the exact Riemann solver.

Because it is one-dimensional, we run it in narrow domains in the x- or y-directions. It can be run as:

```
./pyro.py swe dam inputs.dam.x
./pyro.py swe dam inputs.dam.y
```

A simple script, `dam_compare.py` in `analysis/` will read a pyro output file and plot the solution over the exact dam break solution (as given by Stoker (1958) and Wu, Huang & Zheng (1999)). Below we see the result for a dam break run with 128 points in the x-direction, and run until t = 0.3 s.



We see excellent agreement for all quantities. The shock wave is very steep, as expected. For this problem, the Roe-fix solver performs slightly better than the HLLC solver, with less smearing at the shock and head/tail of the rarefaction.

16.1.2 quad

The quad problem sets up different states in four regions of the domain and watches the complex interfaces that develop as shocks interact. This problem has appeared in several places (and a detailed investigation is online by Pawel Artymowicz). It is run as:

```
./pyro.py swe quad inputs.quad
```

16.1.3 kh

The Kelvin-Helmholtz problem models three layers of fluid: two at the top and bottom of the domain travelling in one direction, one in the central part of the domain travelling in the opposite direction. At the interface of the layers, shearing produces the characteristic Kelvin-Helmholtz instabilities, just as is seen in the standard compressible problem. It is run as:

```
./pyro.py swe kh inputs.kh
```

16.2 Exercises

16.2.1 Explorations

- There are multiple Riemann solvers in the swe algorithm. Run the same problem with the different Riemann solvers and look at the differences. Toro's shallow water text is a good book to help understand what is happening.
- Run the problems with and without limiting—do you notice any overshoots?

16.2.2 Extensions

- Limit on the characteristic variables instead of the primitive variables. What changes do you see? (the notes show how to implement this change.)
- Add a source term to model a non-flat sea floor (bathymetry).

CHAPTER

SEVENTEEN

PARTICLES

A solver for modelling particles.

17.1 particles.particles implementation and use

We import the basic particles module functionality as:

```
import particles.particles as particles
```

The particles solver is made up of two classes:

- `Particle`, which holds the data about a single particle (its position and velocity);
- `Particles`, which holds the data about a collection of particles.

The particles are stored as a dictionary, and their positions are updated based on the velocity on the grid. The keys are tuples of the particles' initial positions (however the values of the keys themselves are never used in the module, so this could be altered using e.g. a custom `particle_generator` function without otherwise affecting the behaviour of the module).

The particles can be initialized in a number of ways:

- `randomly_generate_particles`, which randomly generates `n_particles` within the domain.
- `grid_generate_particles`, which will generate approximately `n_particles` equally spaced in the x-direction and y-direction (note that it uses the same number of particles in each direction, so the spacing will be different in each direction if the domain is not square). The number of particles will be increased/decreased in order to fill the whole domain.
- `array_generate_particles`, which generates particles based on array of particle positions passed to the constructor.
- The user can define their own `particle_generator` function and pass this into the `Particles` constructor. This function takes the number of particles to be generated and returns a dictionary of `Particle` objects.

We can turn on/off the particles solver using the following runtime paramters:

[particles]	
do_particles	Do we want to model particles? (0=no, 1=yes)
n_particles	The number of particles to be modelled
particle_generator	How do we initialize the particles? "random" randomly generates particles throughout the domain, "grid" generates equally spaced particles, "array" generates particles at positions given in an array passed to the constructor. This option can be overridden by passing a custom generator function to the `Particles` constructor.

Using these runtime parameters, we can initialize particles in a problem using the following code in the solver's `Simulation.initialize` function:

```
if self.rp.get_param("particles.do_particles") == 1:
      n_particles = self.rp.get_param("particles.n_particles")
      particle_generator = self.rp.get_param("particles.particle_generator")
      self.particles = particles.Particles(self.cc_data, bc, n_particles, particle_
↪generator)
```

The particles can then be advanced by inserting the following code after the update of the other variables in the solver's `Simulation.evolve` function:

```
if self.particles is not None:
     self.particles.update_particles(self.dt)
```

This will both update the positions of the particles and enforce the boundary conditions.

For some problems (e.g. advection), the x- and y- velocities must also be passed in as arguments to this function as they cannot be accessed using the standard `get_var("velocity")` command. In this case, we would instead use

```
if self.particles is not None:
     self.particles.update_particles(self.dt, u, v)
```

17.2 Plotting particles

Given the `Particles` object `particles`, we can plot the particles by getting their positions using

```
particle_positions = particles.get_positions()
```

In order to track the movement of particles over time, it's useful to 'dye' the particles based on their initial positions. Assuming that the keys of the particles dictionary were set as the particles' initial positions, this can be done by calling

```
colors = particles.get_init_positions()
```

For example, if we color the particles from white to black based on their initial x-position, we can plot them on the figure axis `ax` using the following code:

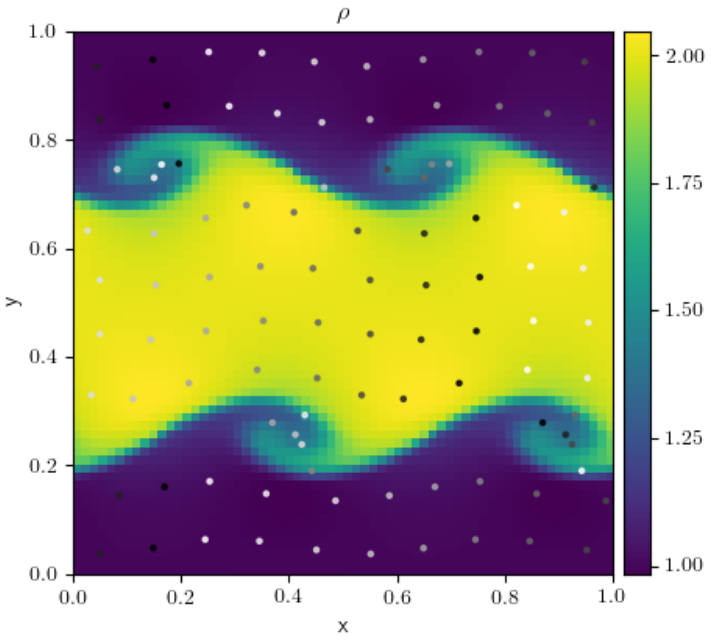
```
particle_positions = particles.get_positions()

# dye particles based on initial x-position
colors = particles.get_init_positions()[:, 0]

# plot particles
ax.scatter(particle_positions[:, 0],
    particle_positions[:, 1], s=5, c=colors, alpha=0.8, cmap="Greys")

ax.set_xlim([myg.xmin, myg.xmax])
ax.set_ylim([myg.ymin, myg.ymax])
```

Applying this to the Kelvin-Helmholtz problem with the `compressible` solver, we can produce a plot such as the one below, where the particles have been plotted on top of the fluid density.



t = 1.6032

CHAPTER

EIGHTEEN

ANALYSIS ROUTINES

In addition to the main pyro program, there are many analysis tools that we describe here. Note: some problems write a report at the end of the simulation specifying the analysis routines that can be used with their data.

- `compare.py`: this takes two simulation output files as input and compares zone-by-zone for exact agreement. This is used as part of the regression testing.

  usage: `./compare.py file1 file2`

- `plot.py`: this takes an output file as input and plots the data using the solver's dovis method. It deduces the solver from the attributes stored in the HDF5 file.

  usage: `./plot.py file`

- `analysis/`

  - `convergence.py`: this compares two files with different resolutions (one a factor of 2 finer than the other). It coarsens the finer data and then computes the norm of the difference. This is used to test the convergence of solvers.

  - `dam_compare.py`: this takes an output file from the shallow water dam break problem and plots a slice through the domain together with the analytic solution (calculated in the script).

    usage: `./dam_compare.py file`

  - `gauss_diffusion_compare.py`: this is for the diffusion solver's Gaussian diffusion problem. It takes a sequence of output files as arguments, computes the angle-average, and the plots the resulting points over the analytic solution for comparison with the exact result.

    usage: `./gauss_diffusion_compare.py file*`

  - `incomp_converge_error.py`: this is for the incompressible solver's converge problem. This takes a single output file as input and compares the velocity field to the analytic solution, reporting the L2 norm of the error.

    usage: `./incomp_converge_error.py file`

  - `plotvar.py`: this takes a single output file and a variable name and plots the data for that variable.

    usage: `./plotvar.py file variable`

  - `sedov_compare.py`: this takes an output file from the compressible Sedov problem, computes the angle-average profile of the solution and plots it together with the analytic data (read in from `cylindrical-sedov.out`).

    usage: `./sedov_compare.py file`

  - `smooth_error.py`: this takes an output file from the advection solver's smooth problem and compares to the analytic solution, outputing the L2 norm of the error.

    usage: `./smooth_error.py file`

- `sod_compare.py`: this takes an output file from the compressible Sod problem and plots a slice through the domain over the analytic solution (read in from `sod-exact.out`).

  usage: `./sod_compare.py file`

CHAPTER

NINETEEN

TESTING

There are two types of testing implemented in pyro: unit tests and regression tests. Both of these are driven by the `test.py` script in the root directory.

19.1 Unit tests

pyro implements unit tests using `py.test`. These can be run via:

```
./test.py -u
```

19.2 Regression tests

The main driver, `pyro.py` has the ability to create benchmarks and compare output to stored benchmarks at the end of a simulation. Benchmark output is stored in each solver's `tests/` directory. When testing, we compare zone-by-zone for each variable to see if we agree exactly. If there is any disagreement, this means that we've made a change to the code that we need to understand (if may be a bug or may be a fix or optimization).

We can compare to the stored benchmarks simply by running:

```
./test.py
```

Note: When running on a new machine, it is possible that roundoff-level differences may mean that we do not pass the regression tests. In this case, one would need to create a new set of benchmarks for that machine and use those for future tests.

CHAPTER

TWENTY

CONTRIBUTING AND GETTING HELP

20.1 Contributing

Contributions are welcomed from anyone, including posting issues or submitting pull requests to the pyro github.

Users who make significant contributions will be listed as developers in the pyro acknowledgements and be included in any future code papers.

20.2 Issues

Creating an issue on github is a good way to request new features, file a bug report, or notify us of any difficulties that arise using pyro.

To request support using pyro, please create an issue on the pyro github and the developers will be happy to assist you.

If you are reporting a bug, please indicate any information necessary to reproduce the bug including your version of python.

20.3 Pull Requests

Any contributions that have the potential to change answers should be done via pull requests. A pull request should be generated from your fork of pyro and target the *main* branch.

The unit and regression tests will run automatically once the PR is submitted, and then one of the pyro developers will review the PR and if needed, suggest modifications prior to merging the PR.

If there are a number of small commits making up the PR, we may wish to squash commits upon merge to have a clean history. *Please ensure that your PR title and first post are descriptive, since these will be used for a squashed commit message.*

20.4 Discussions

We us github discussions: https://github.com/python-hydro/pyro2/discussions for support. You are encouraged to post in the discussions to ask questions.

CHAPTER

TWENTYONE

ACKNOWLEDGMENTS

Pyro developed by (in alphabetical order):

- Alice Harpole
- Ian Hawke
- Michael Zingale

You are free to use this code and the accompanying notes in your classes. Please credit "pyro development team" for the code, and *please send a note to the pyro-help e-mail list describing how you use it, so we can keep track of it (and help justify the development effort).*

If you use pyro in a publication, please cite it using this bibtex citation:

```
@article{pyro,
  doi = {10.21105/joss.01265},
  url = {https://doi.org/10.21105/joss.01265},
  year = {2019},
  publisher = {The Open Journal},
  volume = {4},
  number = {34},
  pages = {1265},
  author = {Alice Harpole and Michael Zingale and Ian Hawke and Taher Chegini},
  title = {pyro: a framework for hydrodynamics explorations and prototyping},
  journal = {Journal of Open Source Software}
}
```

pyro benefited from numerous useful discussions with Ann Almgren, John Bell, and Andy Nonaka.

CHAPTER

TWENTYTWO

HISTORY

The original pyro code was written in 2003-4 to help developmer Zingale understand these methods for himself. It was originally written using the Numeric array package and handwritten C extensions for the compute-intensive kernels. It was ported to numarray when that replaced Numeric, and continued to use C extensions. This version "pyro2" was resurrected beginning in 2012 and rewritten for numpy using f2py, and brought up to date. Most recently we've dropped f2py and are using numba for the compute-intensive kernels.

CHAPTER

TWENTYTHREE

PYRO2

23.1 advection package

23.1.1 Subpackages

advection.problems package

Submodules

advection.problems.smooth module

advection.problems.test module

advection.problems.tophat module

23.1.2 Submodules

23.1.3 advection.advective_fluxes module

23.1.4 advection.simulation module

23.2 advection_fv4 package

23.2.1 Subpackages

advection_fv4.problems package

Submodules

advection_fv4.problems.smooth module

23.2.2 Submodules

23.2.3 advection_fv4.fluxes module

23.2.4 advection_fv4.interface module

23.2.5 advection_fv4.simulation module

23.3 advection_nonuniform package

23.3.1 Subpackages

advection_nonuniform.problems package

Submodules

advection_nonuniform.problems.slotted module

advection_nonuniform.problems.test module

23.3.2 Submodules

23.3.3 advection_nonuniform.advective_fluxes module

23.3.4 advection_nonuniform.simulation module

23.4 advection_rk package

23.4.1 Submodules

23.4.2 advection_rk.fluxes module

23.4.3 advection_rk.simulation module

23.5 advection_weno package

23.5.1 Submodules

23.5.2 advection_weno.fluxes module

23.5.3 advection_weno.simulation module

23.6 compare module

23.7 compressible package

23.7.1 Subpackages

compressible.problems package

Submodules

compressible.problems.acoustic_pulse module

compressible.problems.advect module

compressible.problems.bubble module

compressible.problems.gresho module

compressible.problems.hse module

23.7.2 Submodules

23.7.3 compressible.BC module

23.7.4 compressible.derives module

23.7.5 compressible.eos module

23.7.6 compressible.interface module

23.7.7 compressible.simulation module

23.7.8 compressible.unsplit_fluxes module

23.8 compressible_fv4 package

23.8.1 Subpackages

compressible_fv4.problems.acoustic_pulse module

23.8.2 Submodules

23.8.3 compressible_fv4.fluxes module

23.8.4 compressible_fv4.simulation module

23.9 compressible_react package

23.9.1 Subpackages

compressible_react.problems package

Submodules

compressible_react.problems.flame module

compressible_react.problems.rt module

23.9.2 Submodules

23.9.3 compressible_react.simulation module

23.10 compressible_rk package

23.10.1 Submodules

23.10.2 compressible_rk.fluxes module

23.10.3 compressible_rk.simulation module

23.11 compressible_sdc package

23.11.1 Submodules

23.11.2 compressible_sdc.simulation module

23.12 diffusion package

23.12.1 Subpackages

diffusion.problems package

Submodules

diffusion.problems.gaussian module

diffusion.problems.test module

23.12.2 Submodules

23.12.3 diffusion.simulation module

23.13 examples package

23.13.1 Subpackages

examples.multigrid package

Submodules

examples.multigrid.mg_test_general_alphabeta_only module

examples.multigrid.mg_test_general_beta_only module

examples.multigrid.mg_test_general_constant module

examples.multigrid.mg_test_general_dirichlet module

examples.multigrid.mg_test_general_inhomogeneous module

examples.multigrid.mg_test_simple module

examples.multigrid.mg_test_vc_constant module

examples.multigrid.mg_test_vc_dirichlet module

examples.multigrid.mg_test_vc_periodic module

examples.multigrid.mg_vis module

examples.multigrid.project_periodic module

examples.multigrid.prolong_restrict_demo module

23.14 incompressible package

23.14.1 Subpackages

incompressible.problems package

Submodules

incompressible.problems.converge module

incompressible.problems.shear module

23.14.2 Submodules

23.14.3 incompressible.incomp_interface module

23.14.4 incompressible.simulation module

23.15 lm_atm package

23.15.1 Subpackages

lm_atm.problems package

Submodules

lm_atm.problems.bubble module

lm_atm.problems.gresho module

23.15.2 Submodules

23.15.3 lm_atm.LM_atm_interface module

23.15.4 lm_atm.simulation module

23.16 mesh package

23.16.1 Submodules

23.16.2 mesh.array_indexer module

23.16.3 mesh.boundary module

23.16.4 mesh.fv module

23.16.5 mesh.integration module

23.16.6 mesh.patch module

23.16.7 mesh.reconstruction module

23.17 multigrid package

23.17.1 Submodules

23.17.2 multigrid.MG module

23.17.3 multigrid.edge_coeffs module

23.17.4 multigrid.general_MG module

23.17.5 multigrid.variable_coeff_MG module

23.18 particles package

23.18.1 Submodules

23.18.2 particles.particles module

23.19 plot module

23.20 pyro module

23.21 simulation_null module

23.22 swe package

23.22.2 Submodules

23.22.3 swe.derives module

23.22.4 swe.interface module

23.22.5 swe.simulation module

23.22.6 swe.unsplit_fluxes module

23.23 util package

23.23.1 Submodules

23.23.2 util.io_pyro module

23.23.3 util.msg module

23.23.4 util.plot_tools module

23.23.5 util.profile_pyro module

23.23.6 util.runparams module

CHAPTER

TWENTYFOUR

REFERENCES

CHAPTER

TWENTYFIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Zal79] Steven T Zalesak. Fully multidimensional flux-corrected transport algorithms for fluids. *Journal of Computational Physics*, 31(3):335 – 362, 1979. URL: http://www.sciencedirect.com/science/article/pii/0021999179900512, doi:https://doi.org/10.1016/0021-9991(79)90051-2.

[Colella90] P. Colella. Multidimensional upwind methods for hyperbolic conservation laws. *Journal of Computational Physics*, 87:171–200, March 1990. doi:10.1016/0021-9991(90)90233-Q.

[McCorquodaleColella11] P. McCorquodale and P. Colella. A high-order finite-volume method for conservation laws on locally refined grids. *Communication in Applied Mathematics and Computational Science*, 6(1):1–25, 2011.

PYTHON MODULE INDEX

e

INDEX

E

M