pyro Documentation

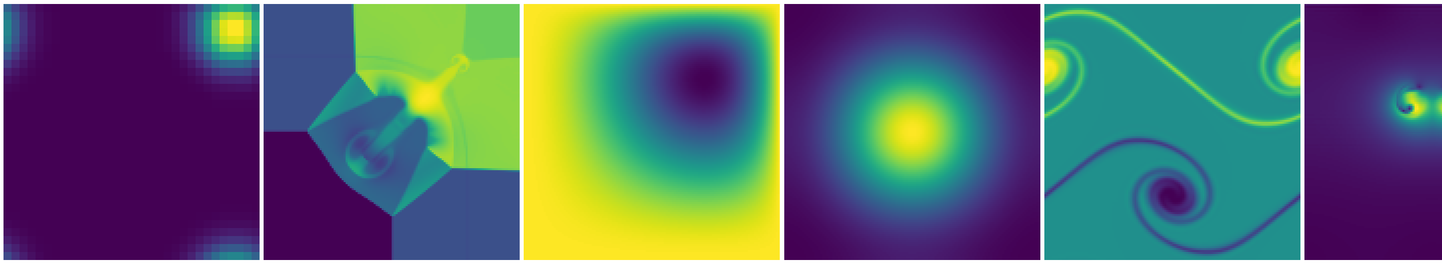
Release 2.2

pyro development team

Feb 22, 2019

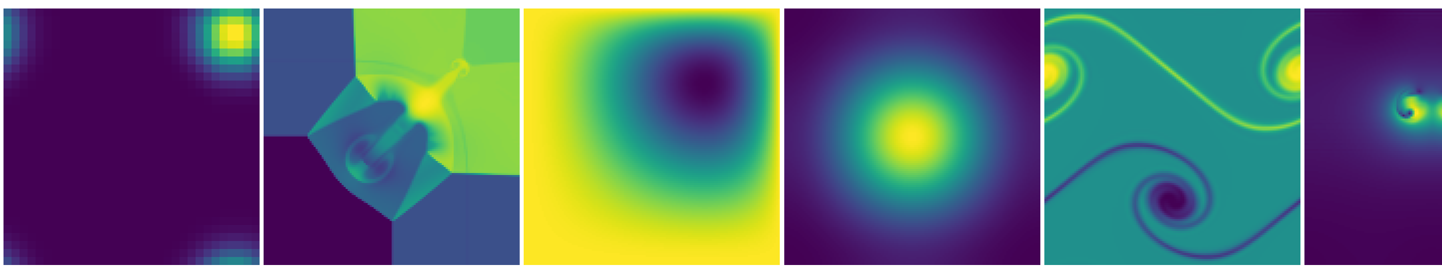
pyro basics

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



CHAPTER 1

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

- shallow water hydrodynamics

Runtime visualization shows the evolution as the equations are solved.

CHAPTER 2

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`
- `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 3

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 4

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 5

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 6

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:

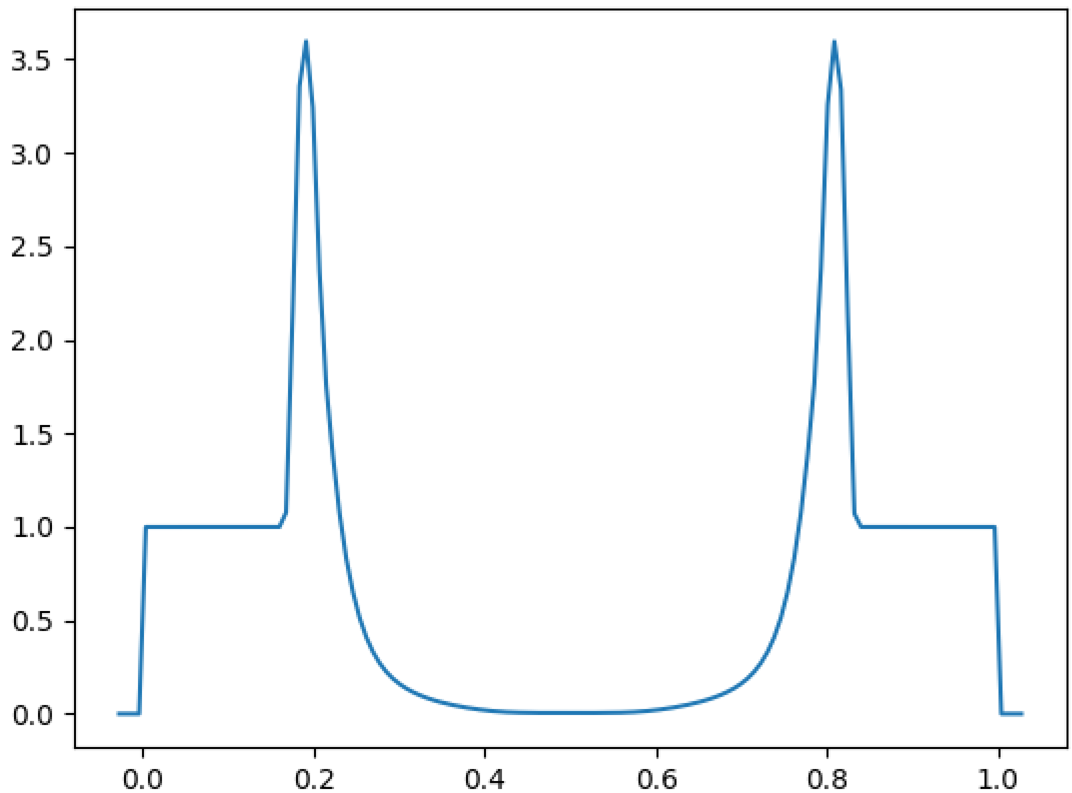
  ```
  ./plot.py -o image.png shear_128_0216.pyro
  ```

  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.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.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 7

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 8

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 9

Mesh examples

this notebook illustrates the basic ways of interacting with the pyro2 mesh module. We create some data that lives on a grid and show how to fill the ghost cells. The pretty_print() function shows us that they work as expected.

[1]:
```
from __future__ import print_function
import numpy as np
import mesh.boundary as bnd
import mesh.patch as patch
import matplotlib.pyplot as plt
%matplotlib inline

# for unit testing, we want to ensure the same random numbers
np.random.seed(100)
```

9.1 Setup a Grid with Variables

There are a few core classes that we deal with when creating a grid with associated variables:

- `Grid2d` : this holds the size of the grid (in zones) and the physical coordinate information, including coordinates of cell edges and centers
- `BC` : this is a container class that simply holds the type of boundary condition on each domain edge.
- `ArrayIndexer` : this is an array of data along with methods that know how to access it with different offsets into the data that usually arise in stencils (like {i+1, j})
- `CellCenterData2d` : this holds the data that lives on a grid. Each variable that is part of this class has its own boundary condition type.

We start by creating a `Grid2d` object with 4 x 6 cells and 2 ghost cells

[2]:
```
g = patch.Grid2d(4, 6, ng=2)
print(g)
```

```
2-d grid: nx = 4, ny = 6, ng = 2
```

[3]:
```
help(g)
```

```
Help on Grid2d in module mesh.patch object:

class Grid2d(builtins.object)
 |  the 2-d grid class.  The grid object will contain the coordinate
 |  information (at various centerings).
 |
 |  A basic (1-d) representation of the layout is::
 |
 |     |     |      |     X     |     |      |     |     X     |      |     |
 |     +-*-+- // -+-*-X-*-+-*-+- // -+-*-+-*-X-*-+- // -+-*-+
 |        0          ng-1    ng   ng+1         ... ng+nx-1 ng+nx      2ng+nx-1
 |
 |                          ilo                      ihi
 |
 |     |<- ng guardcells->|<--- nx interior zones --->|<- ng guardcells->|
 |
 |  The '*' marks the data locations.
 |
 |  Methods defined here:
 |
 |  __eq__(self, other)
 |      are two grids equivalent?
 |
 |  __init__(self, nx, ny, ng=1, xmin=0.0, xmax=1.0, ymin=0.0, ymax=1.0)
 |      Create a Grid2d object.
 |
 |      The only data that we require is the number of points that
 |      make up the mesh in each direction.  Optionally we take the
 |      extrema of the domain (default is [0,1]x[0,1]) and number of
 |      ghost cells (default is 1).
 |
 |      Note that the Grid2d object only defines the discretization,
 |      it does not know about the boundary conditions, as these can
 |      vary depending on the variable.
 |
 |      Parameters
 |      -------
 |      nx : int
 |          Number of zones in the x-direction
 |      ny : int
 |          Number of zones in the y-direction
 |      ng : int, optional
 |          Number of ghost cells
 |      xmin : float, optional
 |          Physical coordinate at the lower x boundary
 |      xmax : float, optional
 |          Physical coordinate at the upper x boundary
 |      ymin : float, optional
 |          Physical coordinate at the lower y boundary
 |      ymax : float, optional
 |          Physical coordinate at the upper y boundary
 |
 |  __str__(self)
```

(continues on next page)

(continued from previous page)

```
 |      print out some basic information about the grid object
 |
 |  coarse_like(self, N)
 |      return a new grid object coarsened by a factor n, but with
 |      all the other properties the same
 |
 |  fine_like(self, N)
 |      return a new grid object finer by a factor n, but with
 |      all the other properties the same
 |
 |  scratch_array(self, nvar=1)
 |      return a standard numpy array dimensioned to have the size
 |      and number of ghostcells as the parent grid
 |
 |  ----------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  ----------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

Then create a dataset that lives on this grid and add a variable name. For each variable that lives on the grid, we need to define the boundary conditions – this is done through the BC object.

[4]:
```
bc = bnd.BC(xlb="periodic", xrb="periodic", ylb="reflect", yrb="outflow")
print(bc)
```

```
BCs: -x: periodic  +x: periodic  -y: reflect-even  +y: outflow
```

[5]:
```
d = patch.CellCenterData2d(g)
d.register_var("a", bc)
d.create()
print(d)
```

```
cc data: nx = 4, ny = 6, ng = 2
         nvars = 1
         variables:
               a: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: periodic     +x: periodic     -y: reflect-even +y: outflow
```

9.2 Working with the data

Now we fill the grid with random data. `get_var()` returns an `ArrayIndexer` object that has methods for accessing views into the data. Here we use `a.v()` to get the "valid" region, i.e. excluding ghost cells.

[6]:
```
a = d.get_var("a")
a.v()[:,:] = np.random.rand(g.nx, g.ny)
```

when we pretty_print() the variable, we see the ghost cells colored red. Note that we just filled the interior above.

[7]:
```
a.pretty_print()
```

```
         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0
         0         0   0.12157    0.2092   0.17194   0.33611         0         0
         0         0 0.0047189   0.89132   0.81168   0.81765         0         0
         0         0   0.84478   0.57509   0.97862   0.94003         0         0
         0         0   0.42452   0.13671    0.2197    0.4317         0         0
         0         0   0.27837   0.82585   0.10838   0.27407         0         0
         0         0    0.5434   0.67075   0.18533   0.81622         0         0
         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0

         ^ y
         |
         +---> x
```

`pretty_print()` can also take an argumet, specifying the format string to be used for the output.

[8]:
```
a.pretty_print(fmt="%7.3g")
```

```
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0
      0      0  0.122  0.209  0.172  0.336      0      0
      0      00.00472  0.891  0.812  0.818      0      0
      0      0  0.845  0.575  0.979   0.94      0      0
      0      0  0.425  0.137   0.22  0.432      0      0
      0      0  0.278  0.826  0.108  0.274      0      0
      0      0  0.543  0.671  0.185  0.816      0      0
      0      0      0      0      0      0      0      0
      0      0      0      0      0      0      0      0

         ^ y
         |
         +---> x
```

now fill the ghost cells – notice that the left and right are periodic, the upper is outflow, and the lower is reflect, as specified when we registered the data above.

[9]:
```
d.fill_BC("a")
a.pretty_print()
```

```
   0.17194   0.33611   0.12157    0.2092   0.17194   0.33611   0.12157    0.2092
   0.17194   0.33611   0.12157    0.2092   0.17194   0.33611   0.12157    0.2092
   0.17194   0.33611   0.12157    0.2092   0.17194   0.33611   0.12157    0.2092
   0.81168   0.81765 0.0047189   0.89132   0.81168   0.81765 0.0047189   0.89132
   0.97862   0.94003   0.84478   0.57509   0.97862   0.94003   0.84478   0.57509
    0.2197    0.4317   0.42452   0.13671    0.2197    0.4317   0.42452   0.13671
   0.10838   0.27407   0.27837   0.82585   0.10838   0.27407   0.27837   0.82585
   0.18533   0.81622    0.5434   0.67075   0.18533   0.81622    0.5434   0.67075
   0.18533   0.81622    0.5434   0.67075   0.18533   0.81622    0.5434   0.67075
   0.10838   0.27407   0.27837   0.82585   0.10838   0.27407   0.27837   0.82585
```

(continues on next page)

(continued from previous page)

```
          ^ y
          |
          +--> x
```

We can find the L2 norm of the data easily

[10]:
```
a.norm()
```

[10]:
```
0.5749769043407793
```

and the min and max

[11]:
```
print(a.min(), a.max())
```

```
0.004718856190972565 0.9786237847073697
```

9.3 ArrayIndexer

We we access the data, an `ArrayIndexer` object is returned. The `ArrayIndexer` sub-classes the NumPy `ndarray`, so it can do all of the methods that a NumPy array can, but in addition, we can use the `ip()`, `jp()`, or `ipjp()` methods to the `ArrayIndexer` object shift our view in the x, y, or x & y directions.

To make this clearer, we'll change our data set to be nicely ordered numbers. We index the `ArrayIndex` the same way we would a NumPy array. The index space includes ghost cells, so the `ilo` and `ihi` attributes from the grid object are useful to index just the valid region. The `.v()` method is a shortcut that also gives a view into just the valid data.

Note: when we use one of the `ip()`, `jp()`, `ipjp()`, or `v()` methods, the result is a regular NumPy `ndarray`, not an `ArrayIndexer` object. This is because it only spans part of the domain (e.g., no ghost cells), and therefore cannot be associated with the `Grid2d` object that the `ArrayIndexer` is built from.

[12]:
```
type(a)
```

[12]:
```
mesh.array_indexer.ArrayIndexer
```

[13]:
```
type(a.v())
```

[13]:
```
numpy.ndarray
```

[14]:
```
a[:,:] = np.arange(g.qx*g.qy).reshape(g.qx, g.qy)
```

[15]:
```
a.pretty_print()
```

```
         9        19        29        39        49        59        69        79
         8        18        28        38        48        58        68        78
         7        17        27        37        47        57        67        77
         6        16        26        36        46        56        66        76
         5        15        25        35        45        55        65        75
         4        14        24        34        44        54        64        74
         3        13        23        33        43        53        63        73
         2        12        22        32        42        52        62        72
         1        11        21        31        41        51        61        71
         0        10        20        30        40        50        60        70
```

(continues on next page)

(continued from previous page)

```
    ^ y
    |
    +--> x
```

We index our arrays as {i,j}, so x (indexed by i) is the row and y (indexed by j) is the column in the NumPy array. Note that python arrays are stored in row-major order, which means that all of the entries in the same row are adjacent in memory. This means that when we simply print out the `ndarray`, we see constant-x horizontally, which is the transpose of what we are used to.

[16]:
```
a.v()
```

[16]:
```
array([[22., 23., 24., 25., 26., 27.],
       [32., 33., 34., 35., 36., 37.],
       [42., 43., 44., 45., 46., 47.],
       [52., 53., 54., 55., 56., 57.]])
```

We can offset our view into the array by one in x – this would be like {i+1, j} when we loop over data. The `ip()` method is used here, and takes an argument which is the (positive) shift in the x (i) direction. So here's a shift by 1

[17]:
```
a.ip(-1, buf=1)
```

[17]:
```
array([[ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.],
       [11., 12., 13., 14., 15., 16., 17., 18.],
       [21., 22., 23., 24., 25., 26., 27., 28.],
       [31., 32., 33., 34., 35., 36., 37., 38.],
       [41., 42., 43., 44., 45., 46., 47., 48.],
       [51., 52., 53., 54., 55., 56., 57., 58.]])
```

A shifted view is necessarily smaller than the original array, and relies on ghost cells to bring new data into view. Because of this, the underlying data is no longer the same size as the original data, so we return it as an `ndarray` (which is actually just a view into the data in the `ArrayIndexer` object, so no copy is made.

To see that it is simply a view, lets shift and edit the data

[18]:
```
d = a.ip(1)
d[1,1] = 0.0
a.pretty_print()
```

```
         9        19        29        39        49        59        69        79
         8        18        28        38        48        58        68        78
         7        17        27        37        47        57        67        77
         6        16        26        36        46        56        66        76
         5        15        25        35        45        55        65        75
         4        14        24        34        44        54        64        74
         3        13        23        33         0        53        63        73
         2        12        22        32        42        52        62        72
         1        11        21        31        41        51        61        71
         0        10        20        30        40        50        60        70

    ^ y
    |
    +--> x
```

Here, since d was really a view into $a_{i+1,j}$, and we accessed element (1,1) into that view (with 0,0 as the origin), we were really accessing the element (2,1) in the valid region

9.4 Differencing

`ArrayIndexer` objects are easy to use to construct differences, like those that appear in a stencil for a finite-difference, without having to explicitly loop over the elements of the array.

Here's we'll create a new dataset that is initialized with a sine function

[19]:
```
g = patch.Grid2d(8, 8, ng=2)
d = patch.CellCenterData2d(g)
bc = bnd.BC(xlb="periodic", xrb="periodic", ylb="periodic", yrb="periodic")
d.register_var("a", bc)
d.create()

a = d.get_var("a")
a[:,:] = np.sin(2.0*np.pi*a.g.x2d)
d.fill_BC("a")
```

Our grid object can provide us with a scratch array (an `ArrayIndexer` object) define on the same grid

[20]:
```
b = g.scratch_array()
type(b)
```

[20]:
```
mesh.array_indexer.ArrayIndexer
```

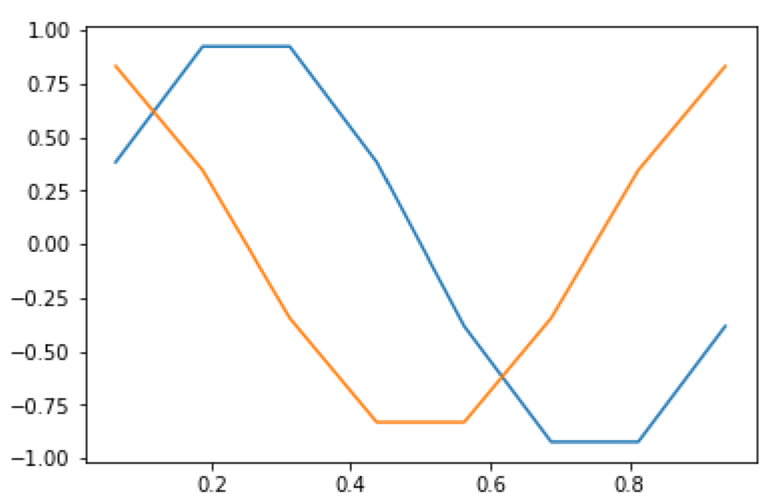
We can then fill the data in this array with differenced data from our original array – since `b` has a separate data region in memory, its elements are independent of `a`. We do need to make sure that we have the same number of elements on the left and right of the =. Since by default, `ip()` will return a view with the same size as the valid region, we can use `.v()` on the left to accept the differences.

Here we compute a centered-difference approximation to the first derivative

[21]:
```
b.v()[:,:] = (a.ip(1) - a.ip(-1))/(2.0*a.g.dx)
# normalization was 2.0*pi
b[:,:] /= 2.0*np.pi
```

[22]:
```
plt.plot(g.x[g.ilo:g.ihi+1], a[g.ilo:g.ihi+1,a.g.jc])
plt.plot(g.x[g.ilo:g.ihi+1], b[g.ilo:g.ihi+1,b.g.jc])
print (a.g.dx)
```

```
0.125
```



9.5 Coarsening and prolonging

we can get a new `ArrayIndexer` object on a coarser grid for one of our variables

```
[23]: c = d.restrict("a")
```

```
[24]: c.pretty_print()
```

```
         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0
         0         0   0.65328   0.65328  -0.65328  -0.65328         0         0
         0         0   0.65328   0.65328  -0.65328  -0.65328         0         0
         0         0   0.65328   0.65328  -0.65328  -0.65328         0         0
         0         0   0.65328   0.65328  -0.65328  -0.65328         0         0
         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0

         ^ y
         |
         +--> x
```

or a finer grid

```
[25]: f = d.prolong("a")
```

```
[26]: f.pretty_print(fmt="%6.2g")
```

```
     0     0     0     0     0     0     0     0     0     0     0     0     0     0 ␣
↪    0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0     0     0 ␣
↪    0     0     0     0     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
     0     0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22     0     0
```

(continues on next page)

(continued from previous page)

```
     0      0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22      0      0
     0      0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22      0      0
     0      0  0.22  0.55  0.86  0.99  0.99  0.86  0.55  0.22 -0.22 -0.55 -0.86 -0.99␣
↪-0.99 -0.86 -0.55 -0.22      0      0
     0      0      0      0      0      0      0      0      0      0      0      0      0      0 ␣
↪    0      0      0      0      0      0
     0      0      0      0      0      0      0      0      0      0      0      0      0      0 ␣
↪    0      0      0      0      0      0

         ^ y
         |
         +--> x
```

CHAPTER 10

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.

10.1 `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`	

10.2 `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

10.3 `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	

- section: [slotted]

option	value	description
omega	0.5	angular velocity
offset	0.25	offset of the slot's center from domain's center

10.4 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

10.5 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]
- section: [driver]

option	value	description
cfl	0.5	advective CFL number

10.6 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:

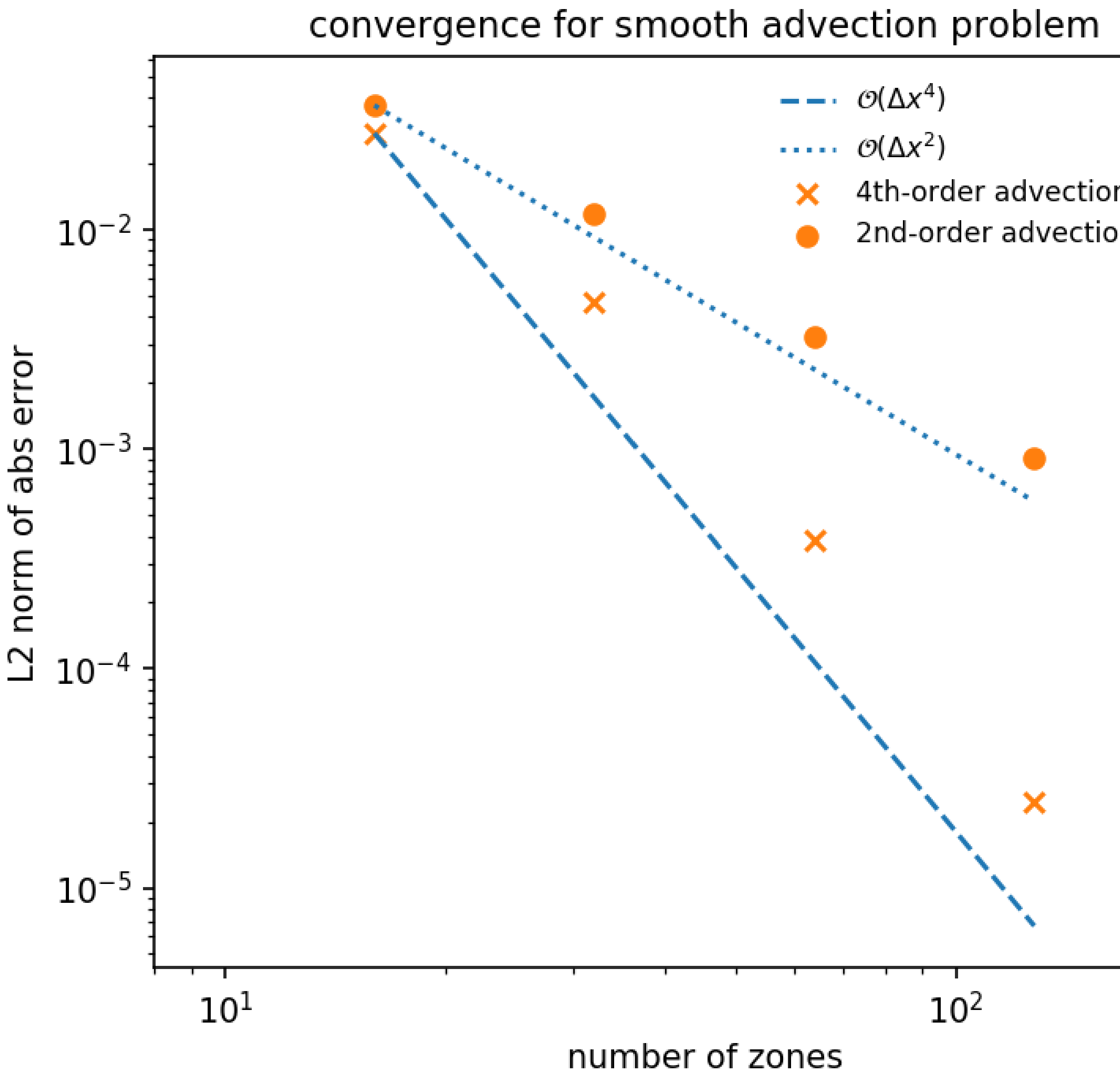
10.7 Examples

10.7.1 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.

10.7.2 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.

10.8 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.

10.8.1 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?)

10.8.2 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 11

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.

11.1 `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	

11.2 `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]
- section: [driver]

option	value	description
cfl	0.8	

- section: [eos]

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

11.3 `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)

11.4 `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)

11.5 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.

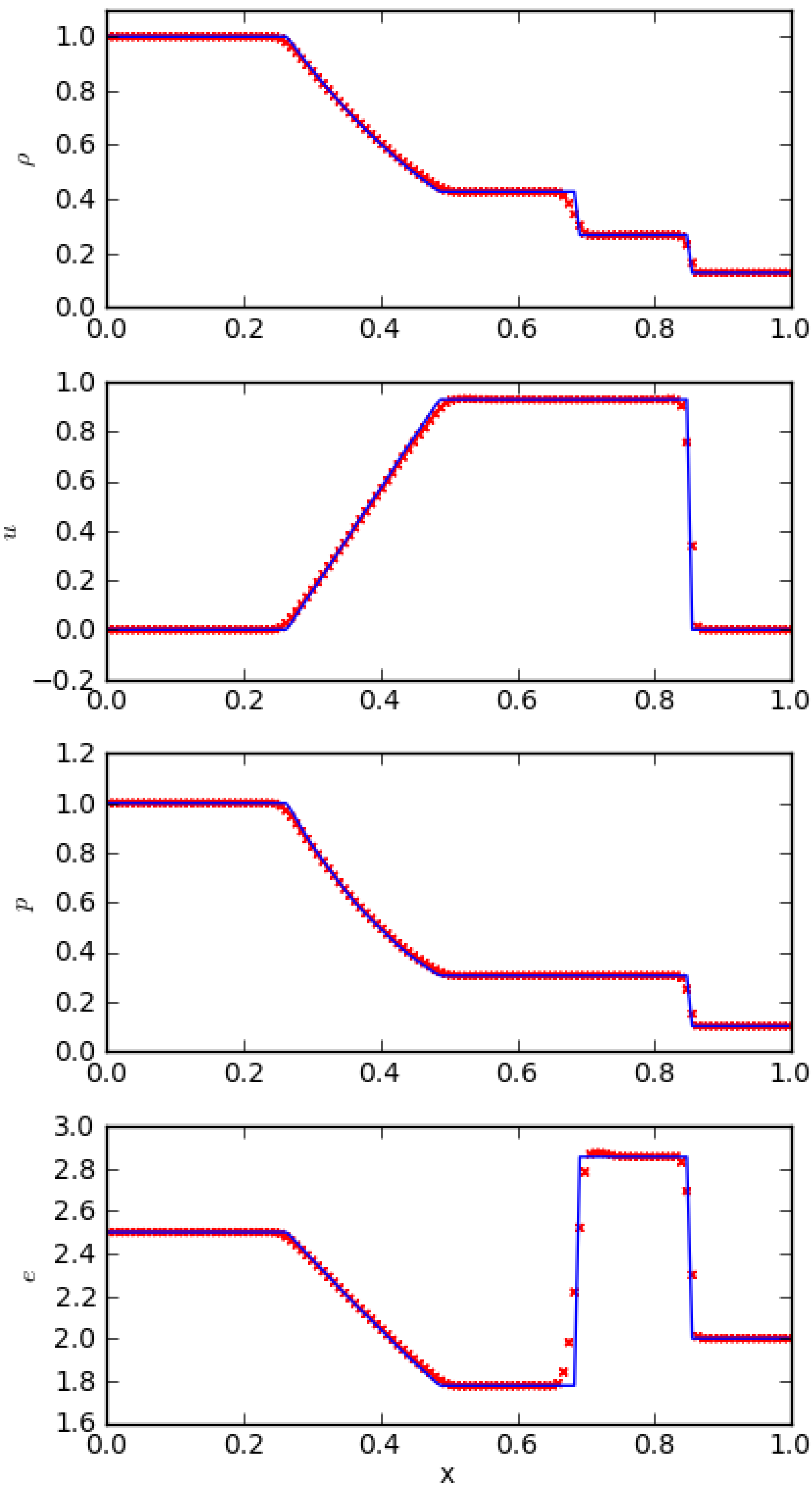
11.5.1 Sod

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.

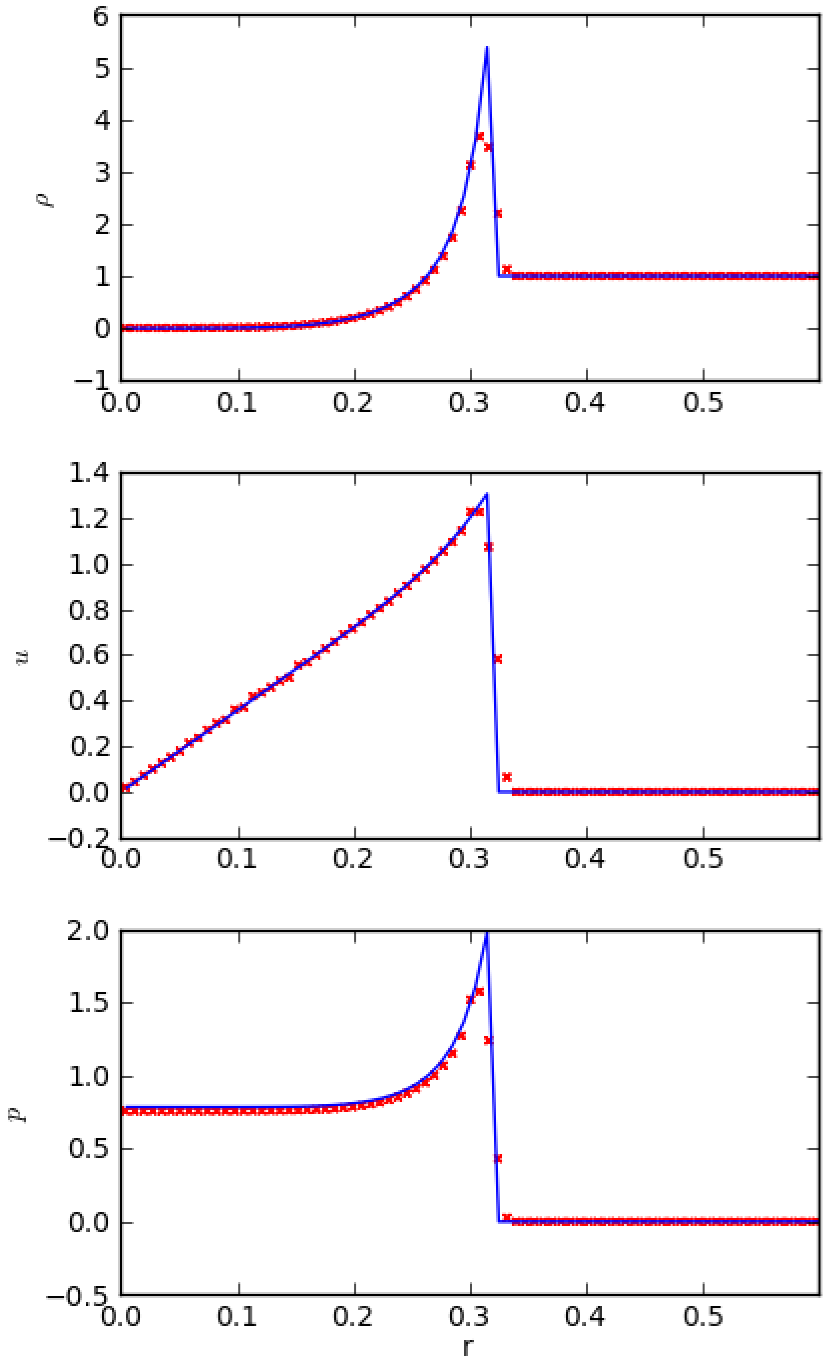
11.5.2 Sedov

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/`

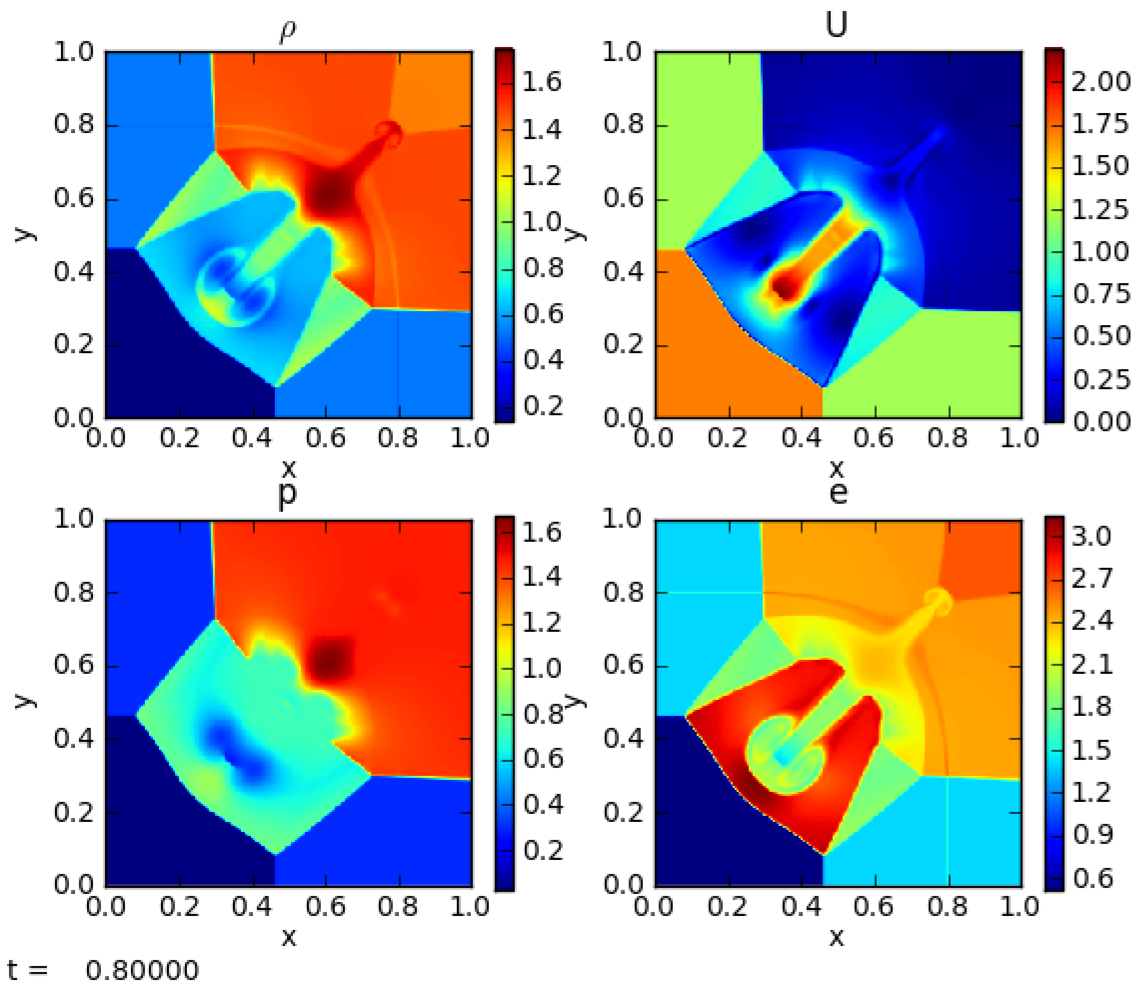


This shows good agreement with the analytic solution.

11.5.3 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
```



11.5.4 rt

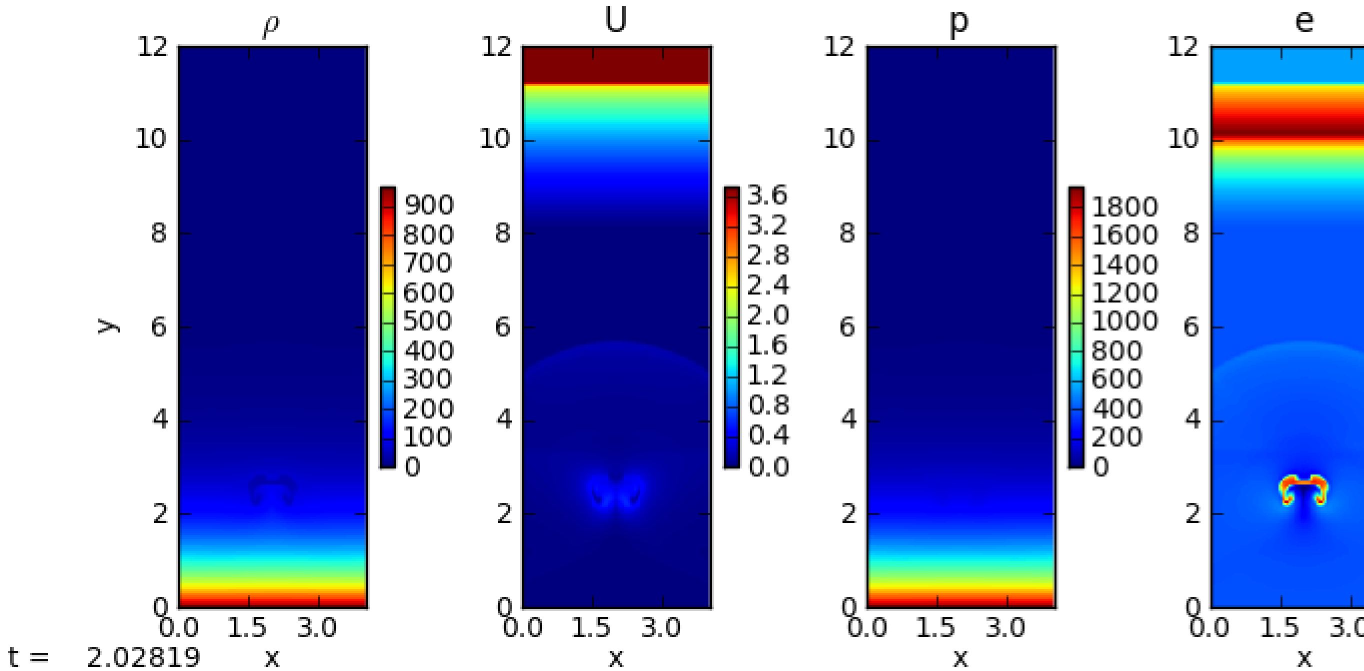
The Rayleigh-Taylor problem puts a dense fluid over a lighter one and perturbs the interface with a sinusoidal velocity. Hydrostatic boundary conditions are used to ensure any initial pressure waves can escape the domain. It is run as:

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

11.5.5 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).

11.6 Exercises

11.6.1 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?

11.6.2 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.

11.7 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 12

Compressible solver comparisons

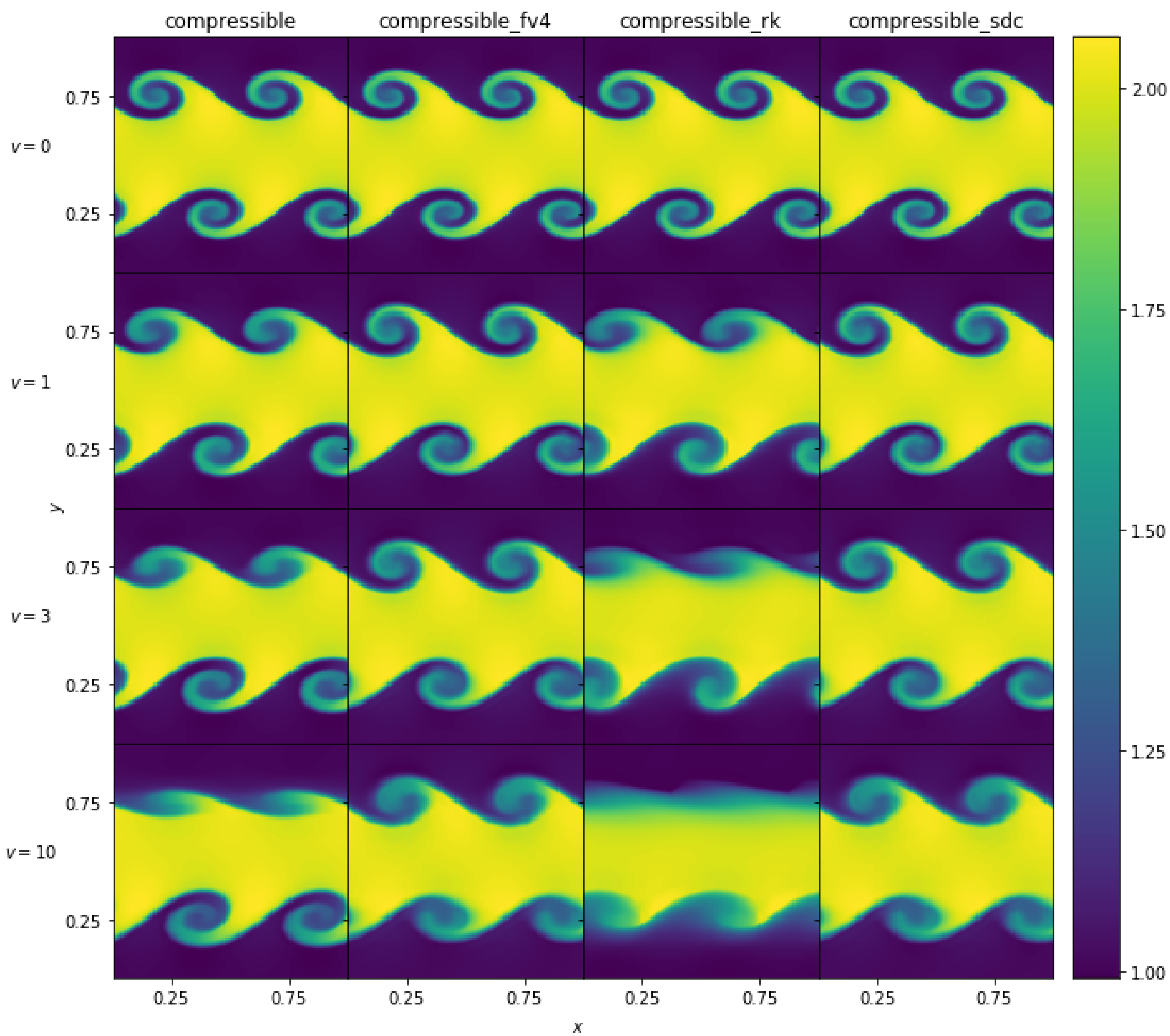
We run various problems run with the different compressible solvers in pyro (standard Riemann, Runge-Kutta, fourth order).

12.1 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
```

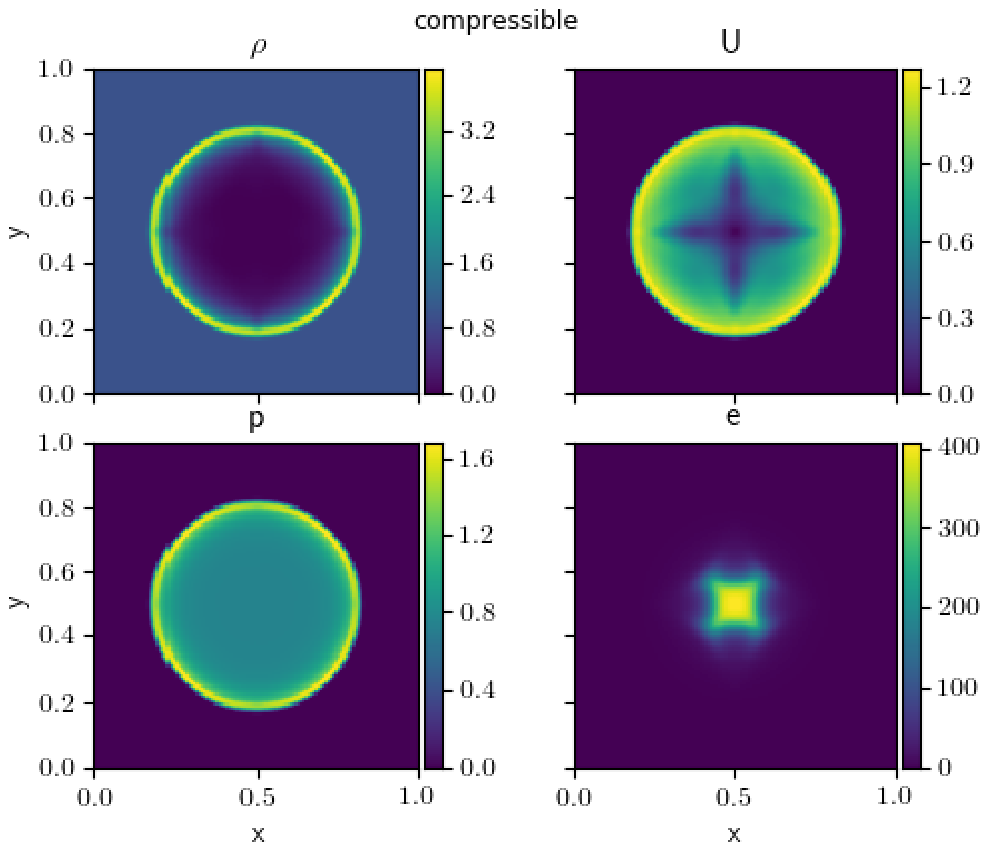


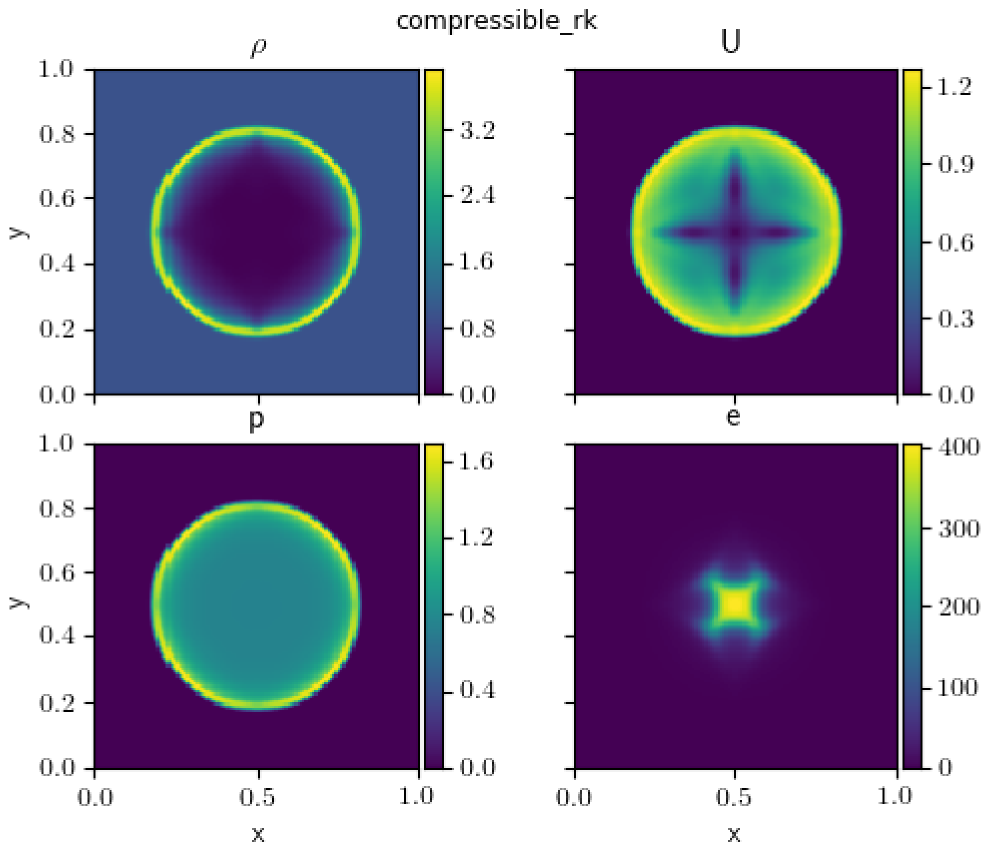
We vary the velocity in the positive y-direction (vbulk) to see how effective the solvers are at preserving the initial shape.

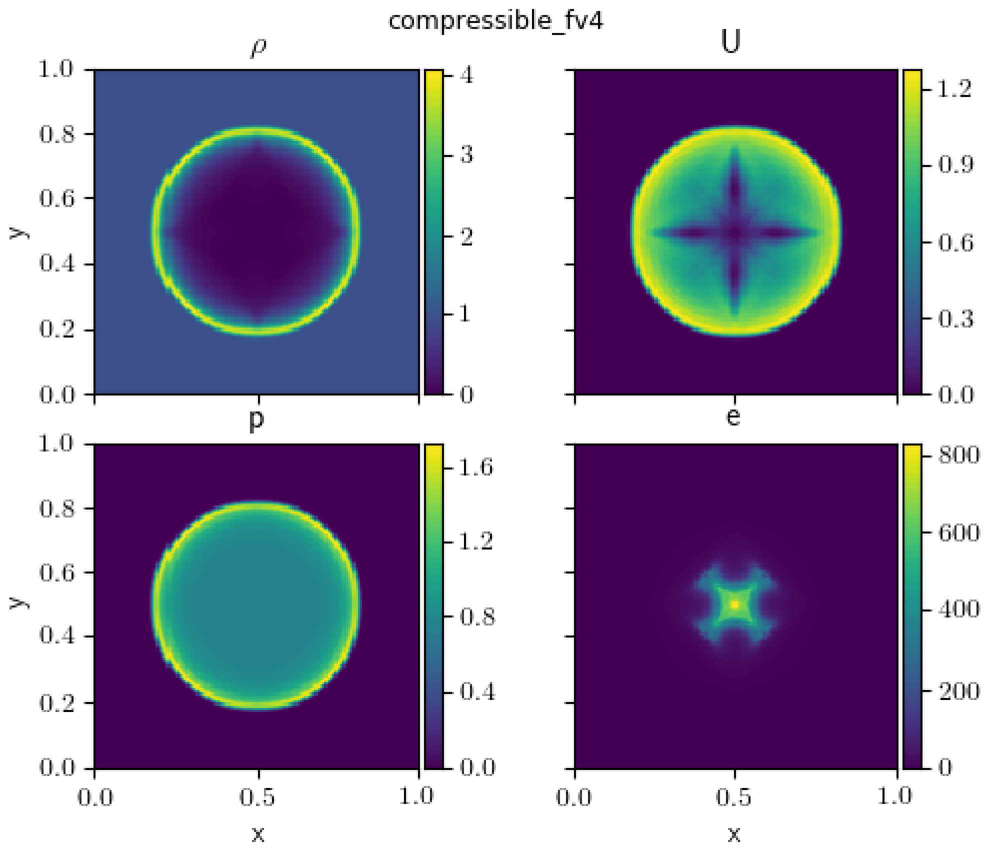
12.2 Sedov

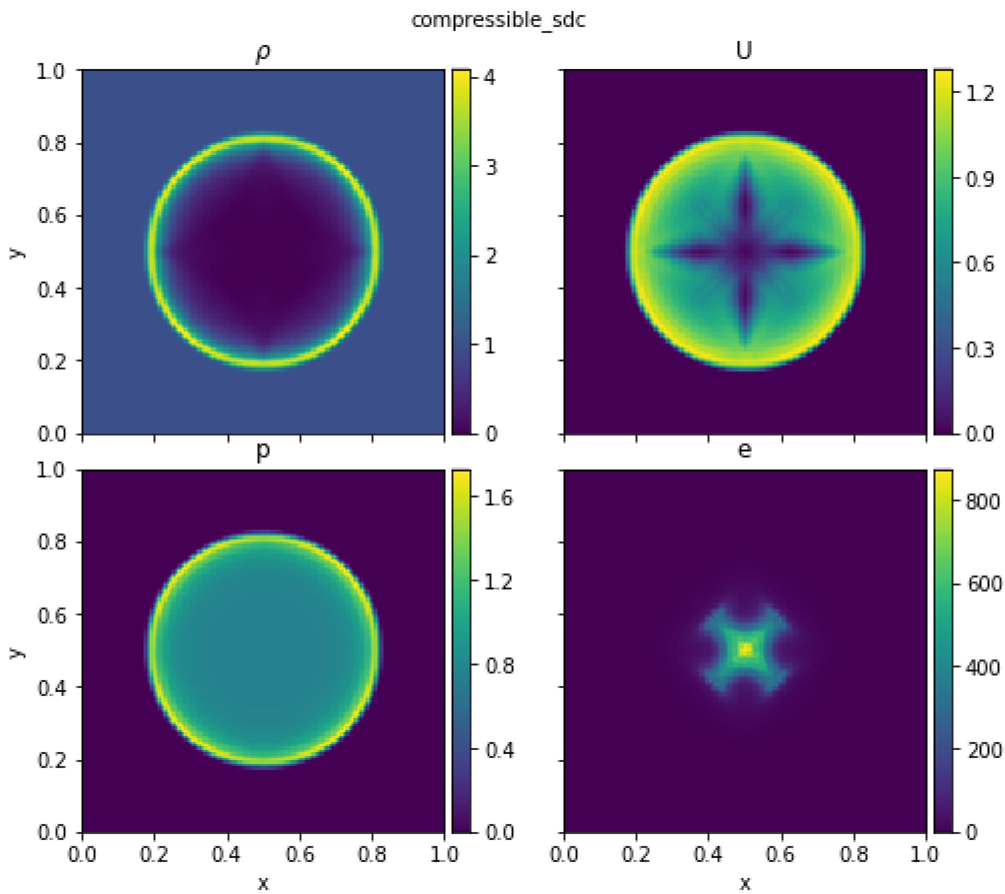
The Sedov problem ran on a 128 x 128 grid, with gamma = 1.4, and until t = 0.1, which can be run as:

```
./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
x
y
0.0
0.2
0.4
0.6
0.8
1.0
0.5
3.2
2.4
1.6
0.8
1.2
0.9
0.6
0.3
0.4
400
300
200
100
0


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

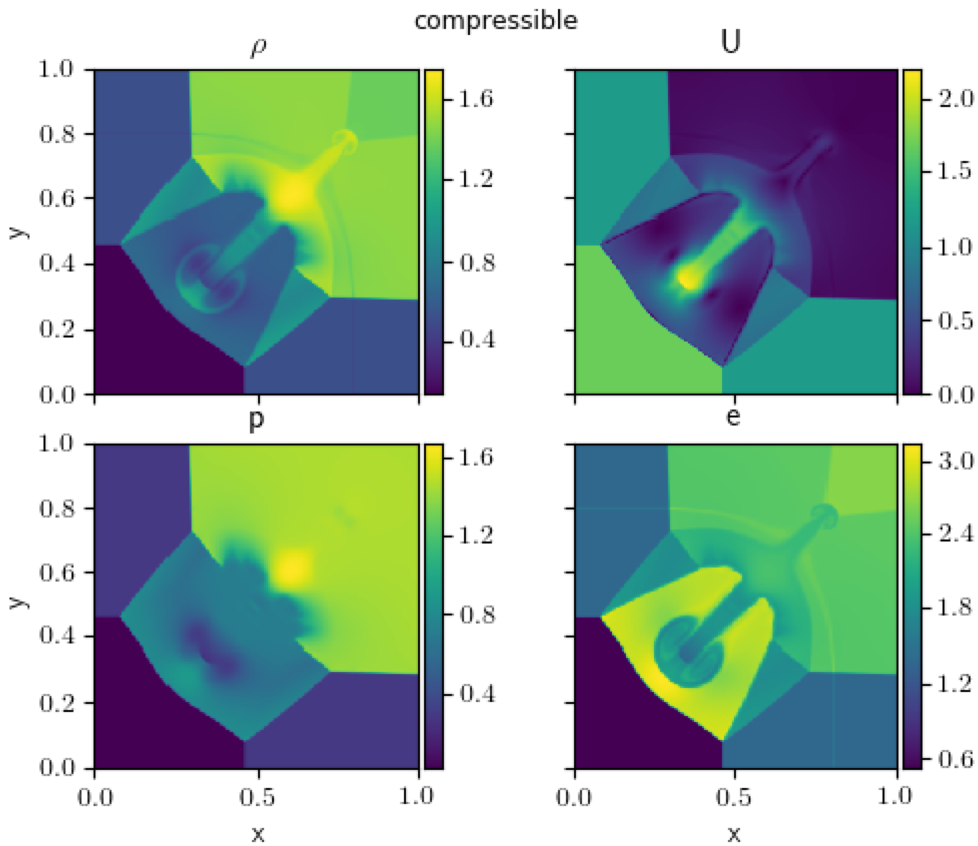

compressible_fv4
ρ
U
p
e
x
y
0.0
0.5
1.0

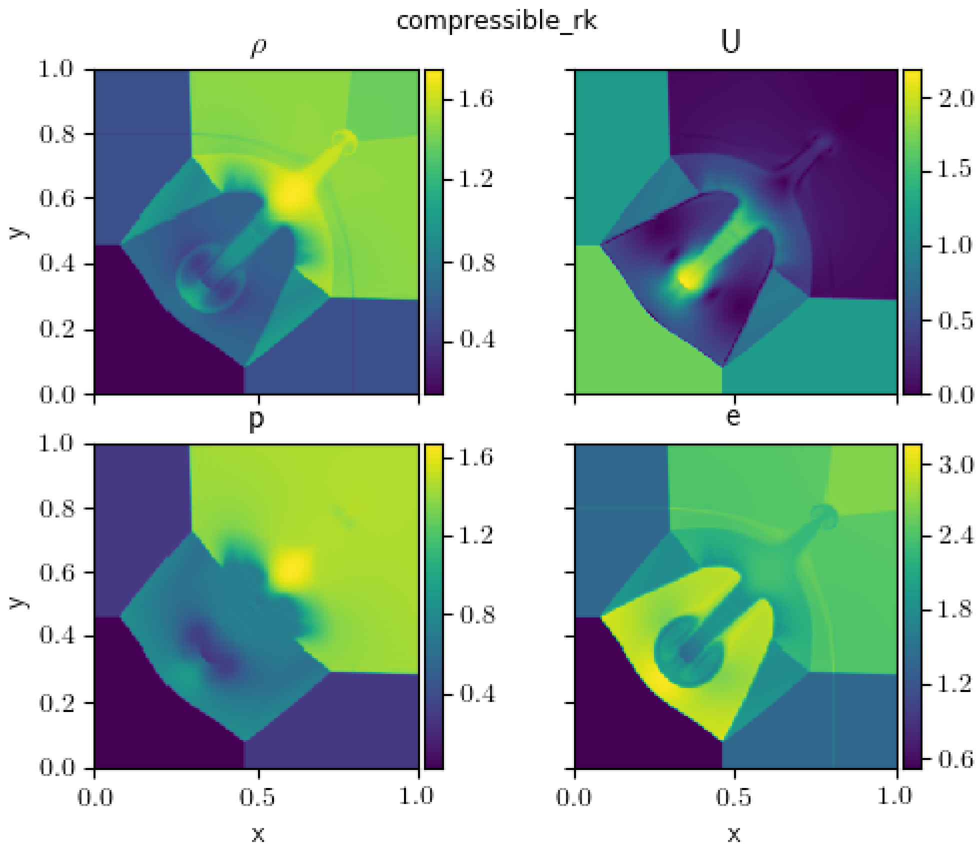


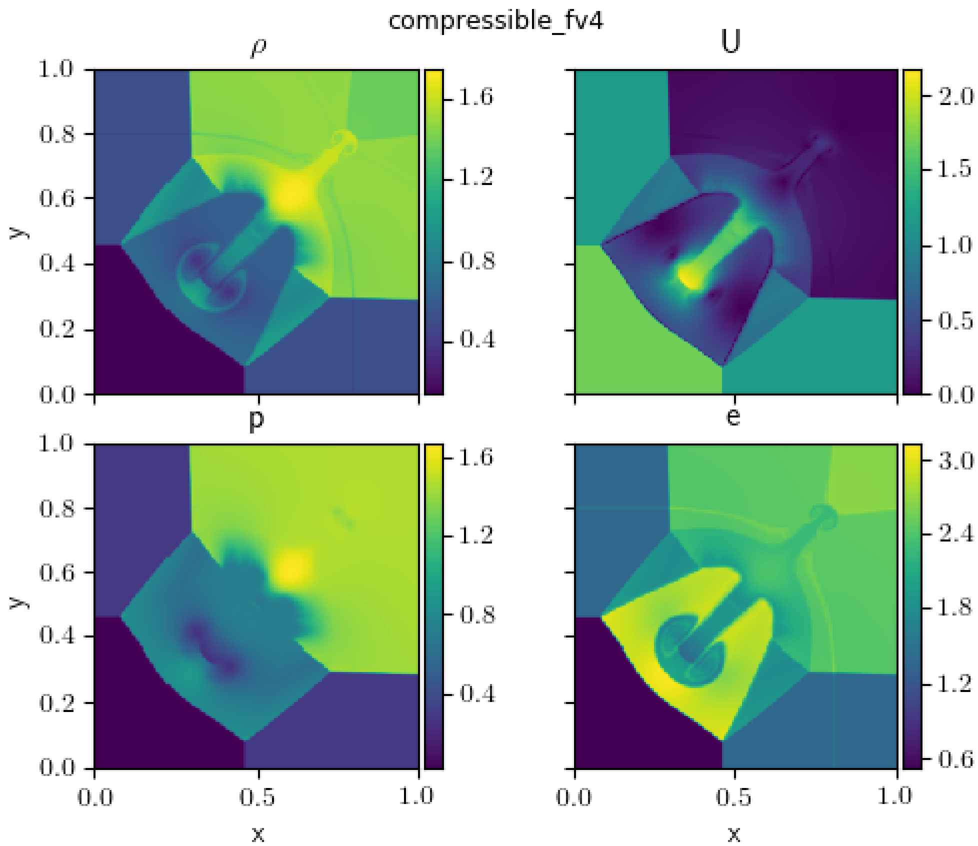
12.3 Quad

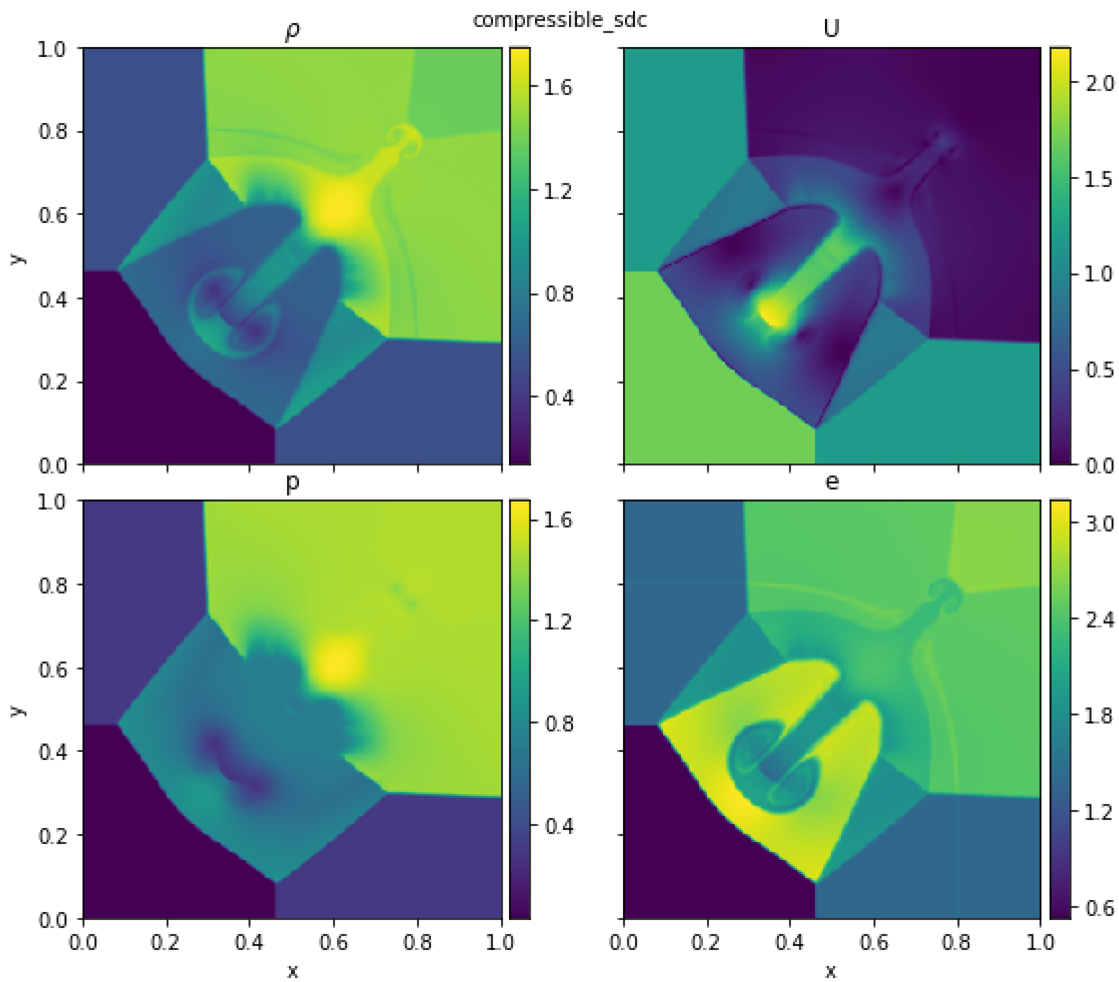
The quad problem ran on a 256 x 256 grid until t = 0.8, which can be run as:

```
./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
y
x
x
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.5
1.0
0.4
0.8
1.2
1.6
0.0
0.5
1.0
1.5
2.0
0.6
1.2
1.8
2.4
3.0


compressible_rk
ρ
U
p
e
y
y
x
x
1.0
0.8
0.6
0.4
0.2
0.0
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
0.0
0.5
1.0

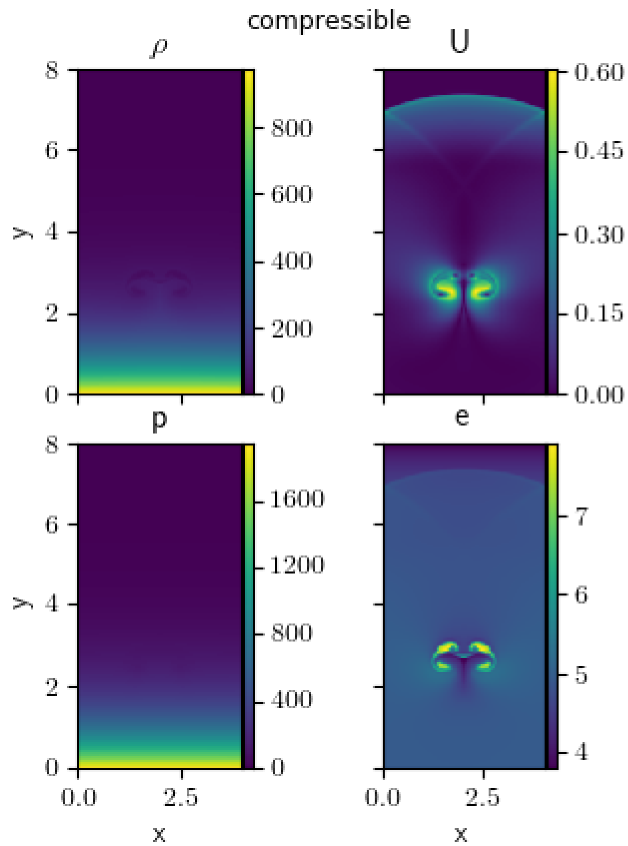

compressible_fv4
ρ
U
p
e
y
x
0.0
0.2
0.4
0.6
0.8
1.0
0.5
1.6
1.2
0.8
0.4
2.0
1.5
1.0
3.0
2.4
1.8
1.2
0.6

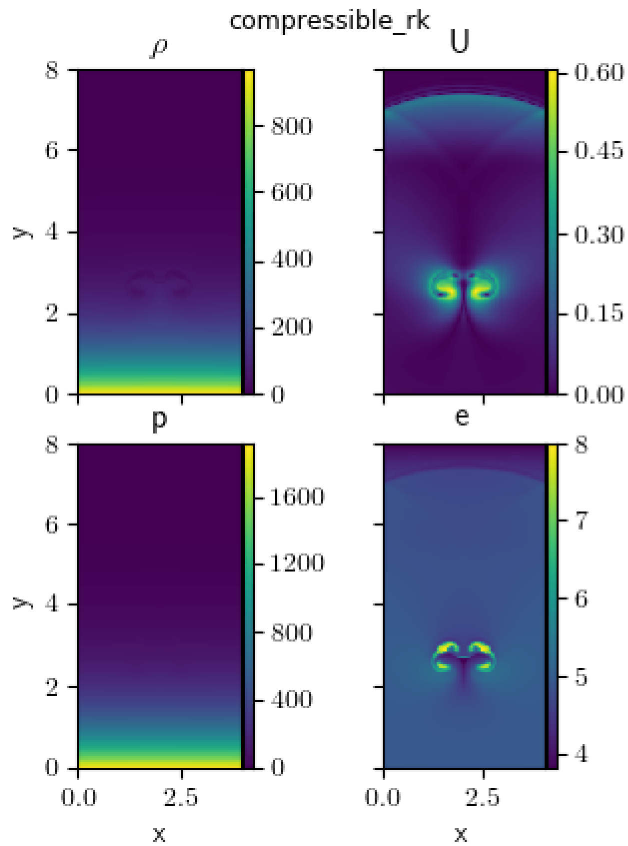


12.4 Bubble

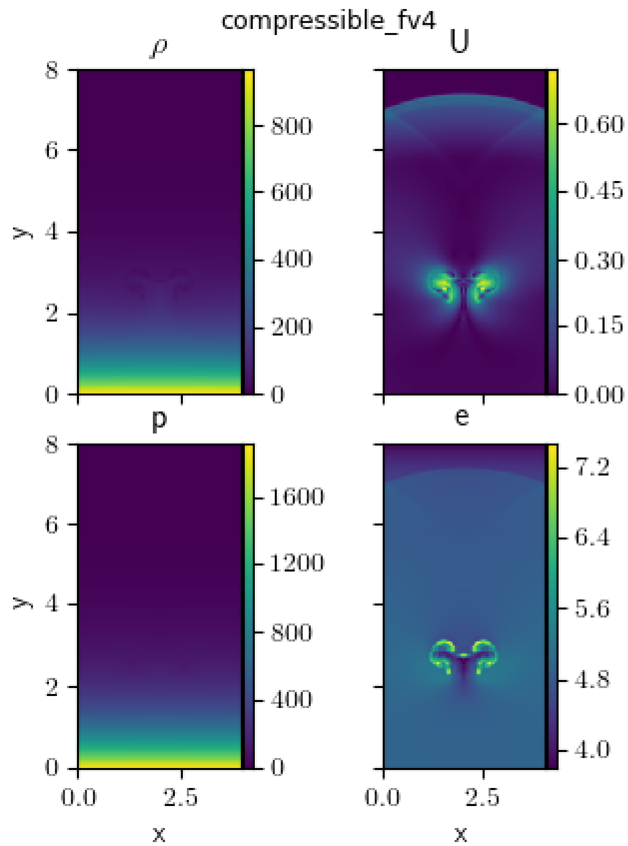
The bubble problem ran on a 128 x 256 grid until t = 3.0, which can be run as:

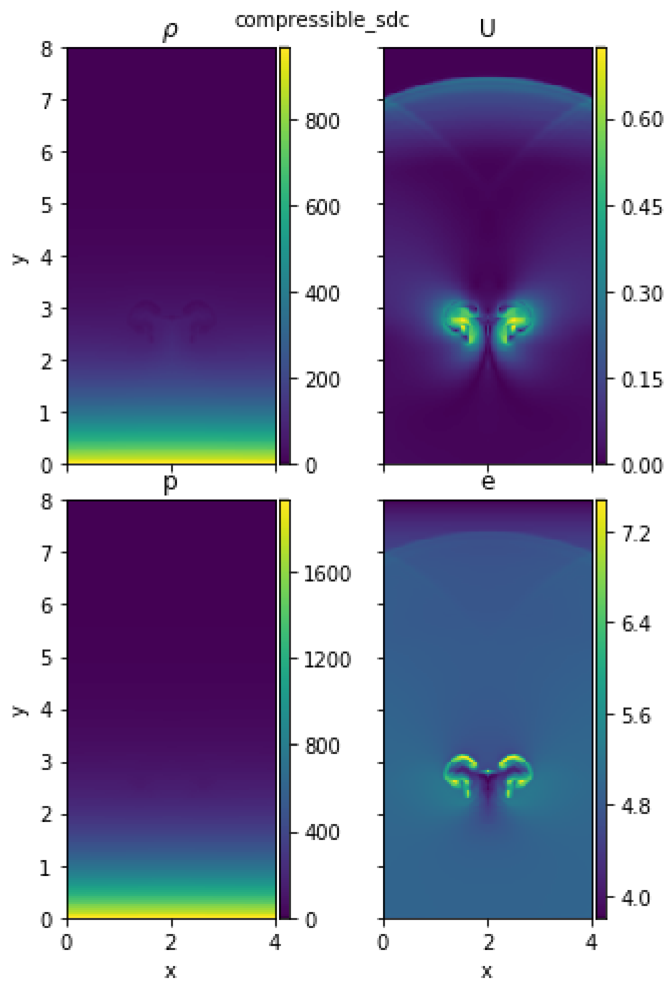
```
./pyro.py compressible bubble inputs.bubble
./pyro.py compressible_rk bubble inputs.bubble
./pyro.py compressible_fv4 bubble inputs.bubble
./pyro.py compressible_sdc bubble inputs.bubble
```


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



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

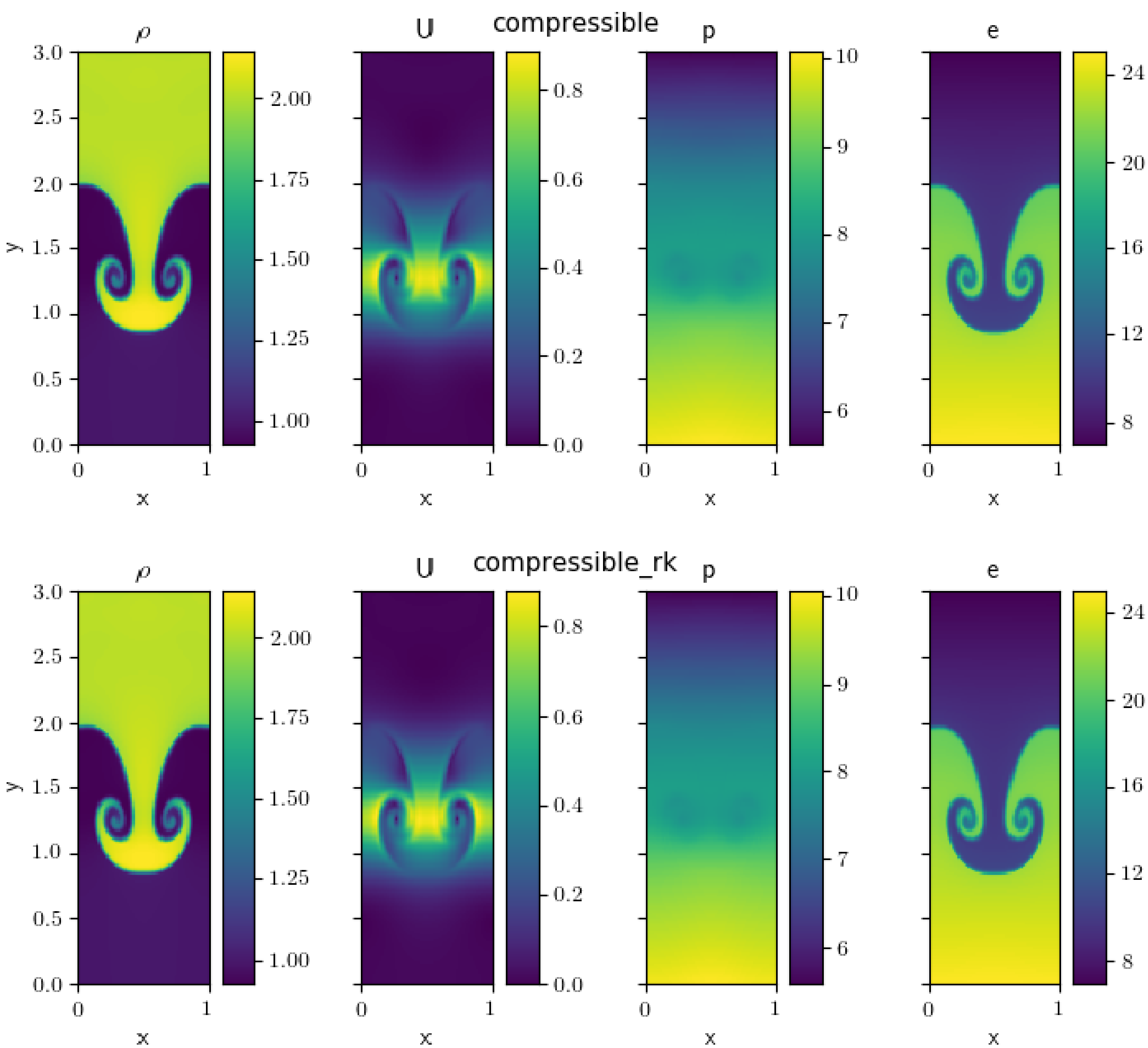

compressible_fv4
ρ
U
p
e
y
x
0.0
2.5
0
2
4
6
8
800
600
400
200
0
0.60
0.45
0.30
0.15
0.00
1600
1200
800
400
0
7.2
6.4
5.6
4.8
4.0

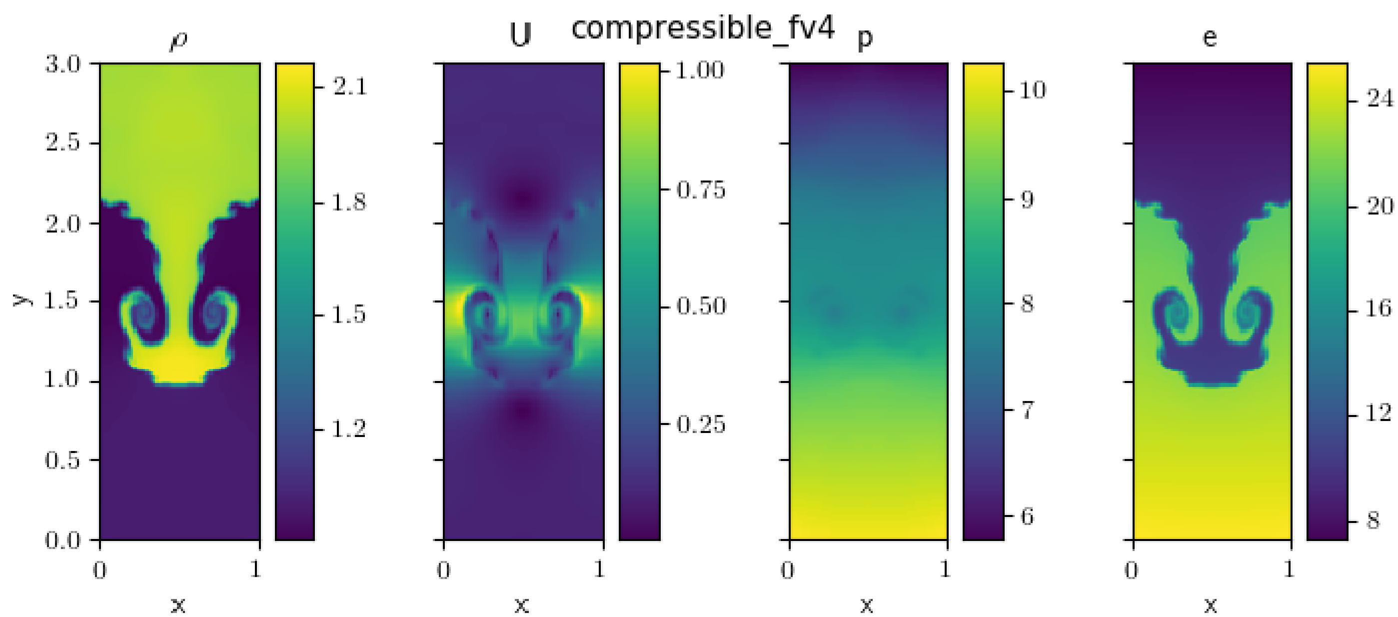


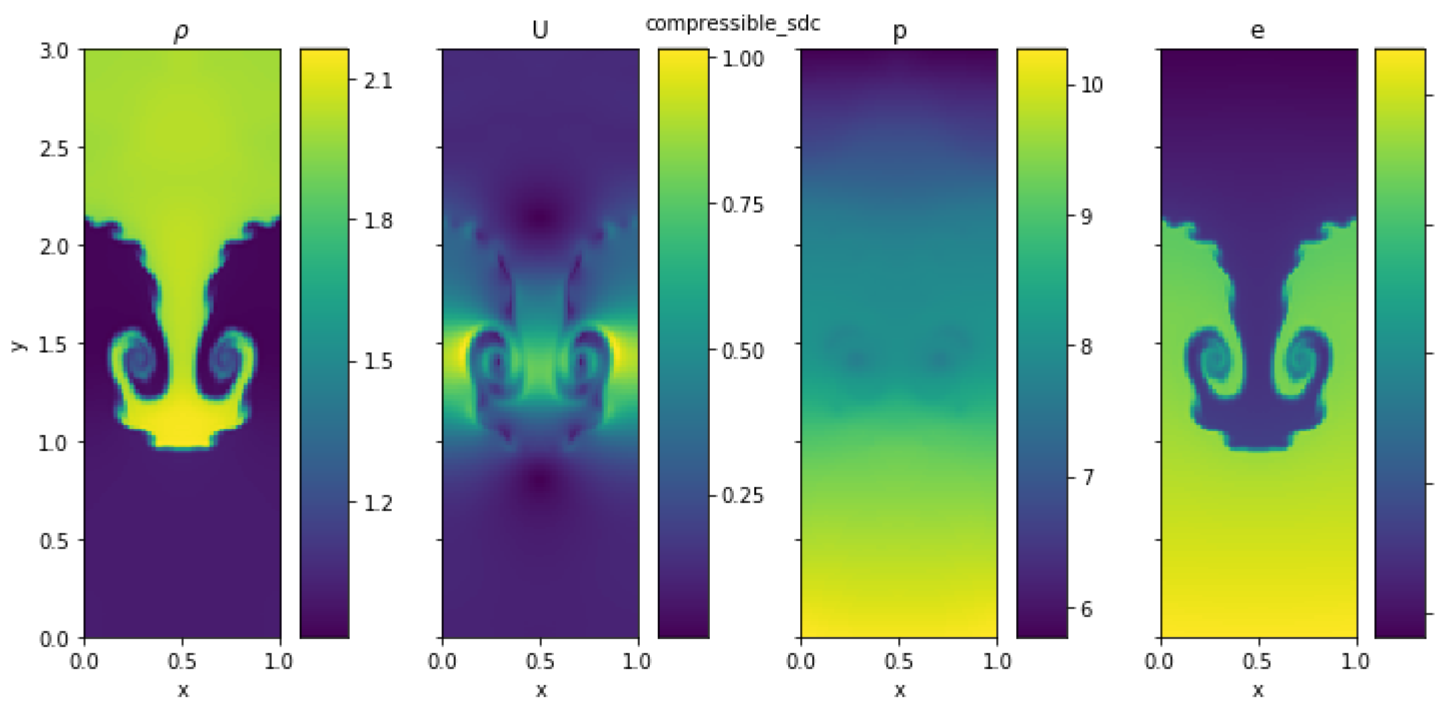
12.5 Rayleigh-Taylor

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
y
x
compressible_rk


compressible_fv4
ρ
U
p
e
x
y


compressible_sdc
ρ
U
p
e
x
y

CHAPTER 13

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.

13.1 Examples

13.1.1 multigrid test

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

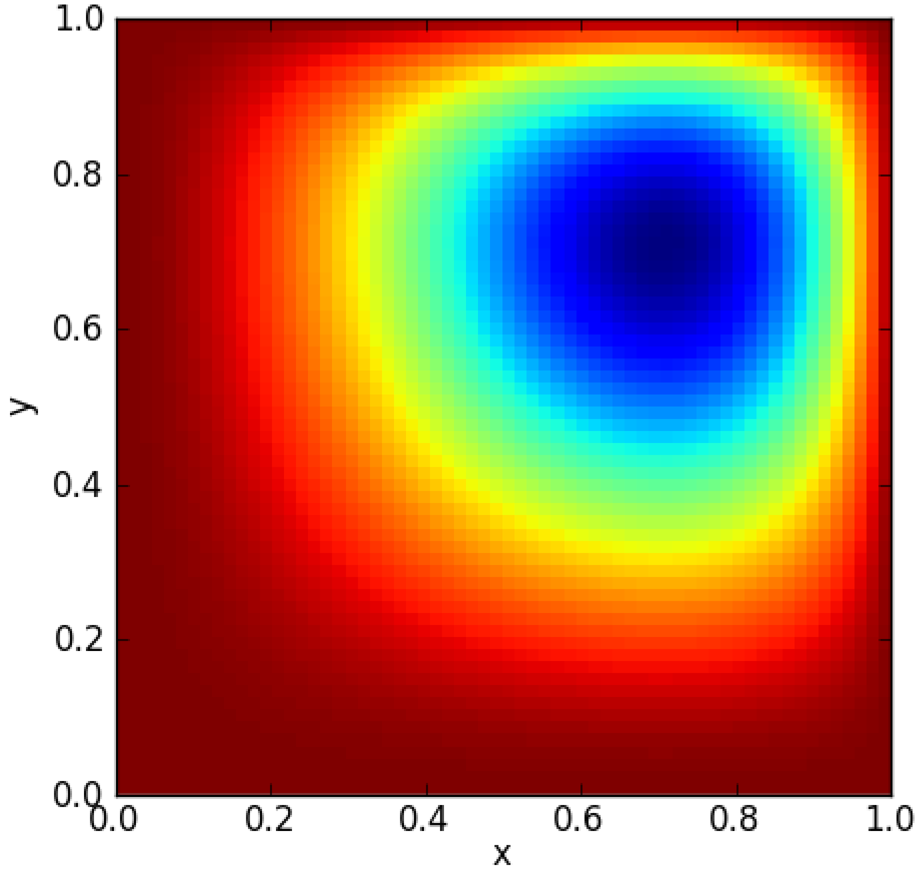
```
./examples/multigrid/mg_test_simple.py
```

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.

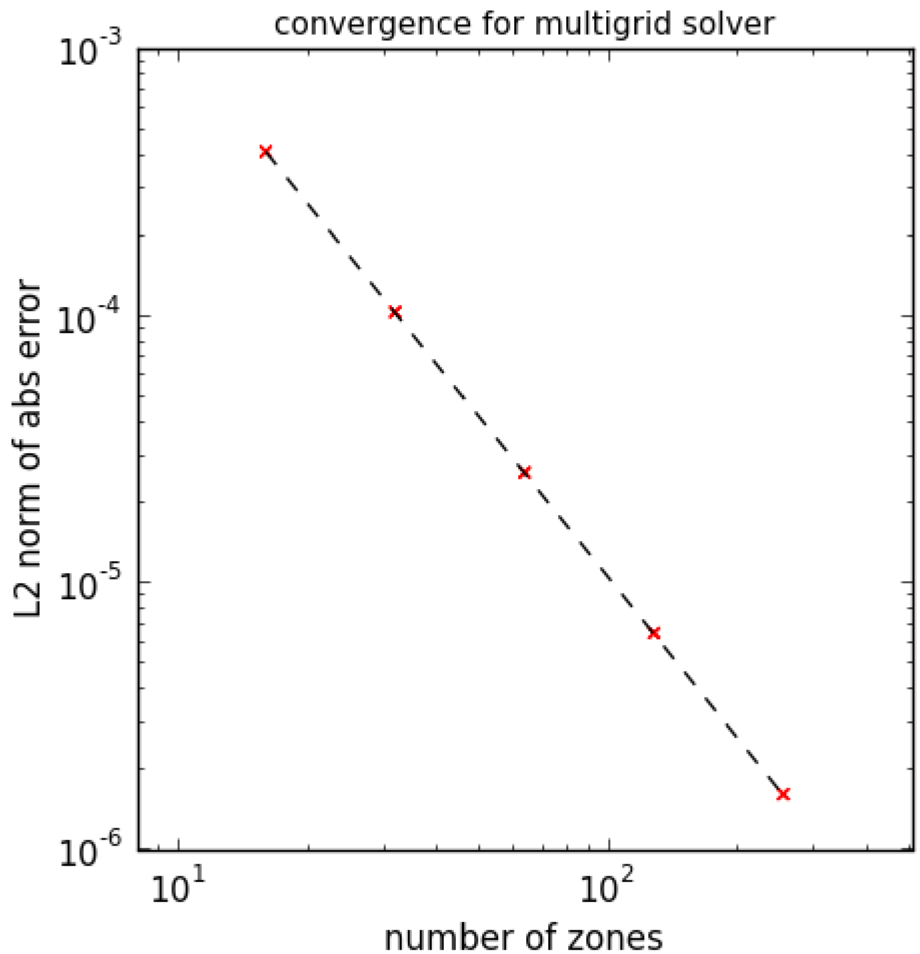
The solution to this is shown below.



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
```

13.1.2 projection

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

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

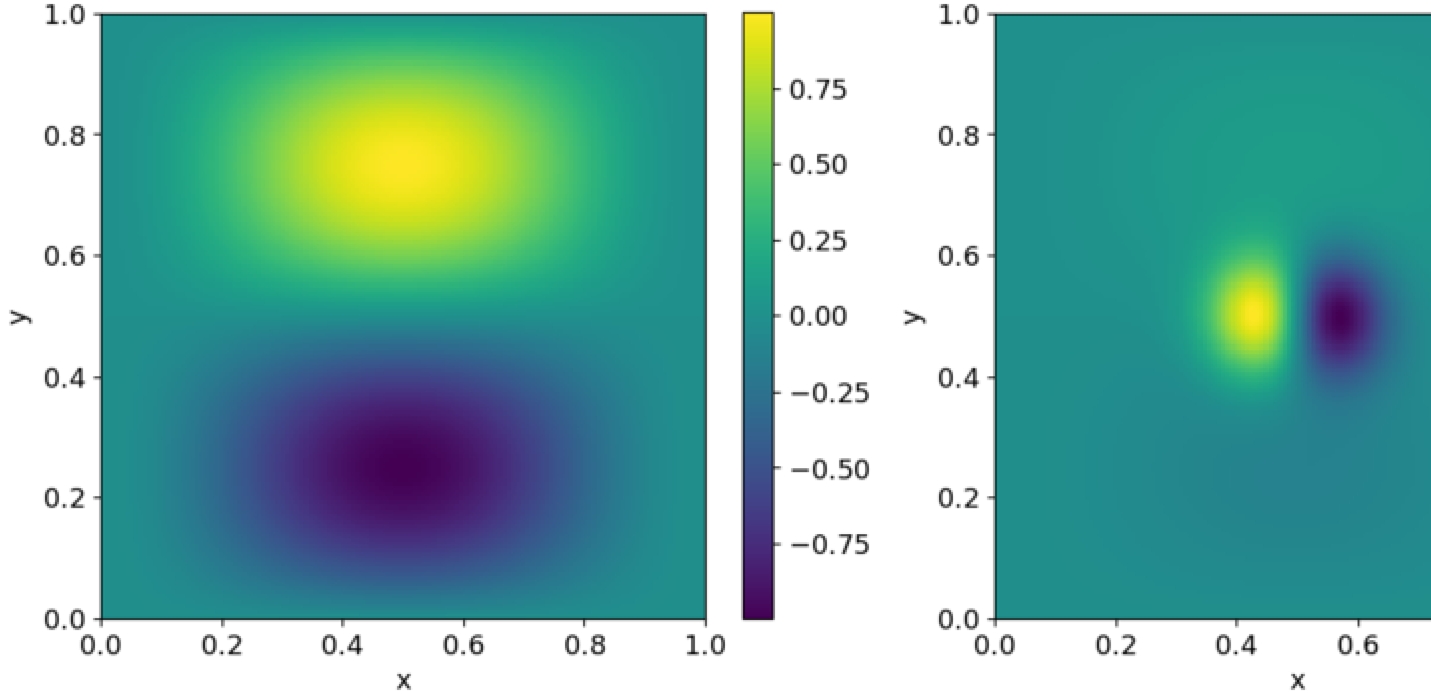
Given a vector field, U, we can decompose it into a divergence free part, U_d, and the gradient of a scalar, ϕ:

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

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.

13.2 Exercises

13.2.1 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.

13.2.2 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 14

Multigrid examples

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

[2]:
```
from __future__ import print_function

import numpy as np

import mesh.boundary as bnd
import mesh.patch as patch
import multigrid.MG as MG
```

14.1 Constant-coefficent Poisson equation

We want to solve

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

on

$$[0, 1] \times [0, 1]$$

with homogeneous Dirichlet boundary conditions (this example comes from "A Multigrid Tutorial").

This has the analytic solution

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

We start by setting up a multigrid object–this needs to know the number of zones our problem is defined on

[3]:
```
nx = ny = 256
mg = MG.CellCenterMG2d(nx, ny,
                       xl_BC_type="dirichlet", xr_BC_type="dirichlet",
                       yl_BC_type="dirichlet", yr_BC_type="dirichlet", verbose=1)
```

```
cc data: nx = 2, ny = 2, ng = 1
         nvars = 3
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet

cc data: nx = 4, ny = 4, ng = 1
         nvars = 3
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet

cc data: nx = 8, ny = 8, ng = 1
         nvars = 3
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet

cc data: nx = 16, ny = 16, ng = 1
         nvars = 3
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet

cc data: nx = 32, ny = 32, ng = 1
         nvars = 3
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
```

(continues on next page)

(continued from previous page)

```
                 f: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
                 r: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet

cc data: nx = 64, ny = 64, ng = 1
         nvars = 3
         variables:
                 v: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
                 f: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
                 r: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet

cc data: nx = 128, ny = 128, ng = 1
         nvars = 3
         variables:
                 v: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
                 f: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
                 r: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet

cc data: nx = 256, ny = 256, ng = 1
         nvars = 3
         variables:
                 v: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
                 f: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
                 r: min:    0.0000000000    max:    0.0000000000
                    BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
```

Next, we initialize the RHS. To make life easier, the `CellCenterMG2d` object has the coordinates of the solution grid (including ghost cells) as `mg.x2d` and `mg.y2d` (these are two-dimensional arrays).

[4]:
```
def rhs(x, y):
    return -2.0*((1.0-6.0*x**2)*y**2*(1.0-y**2) + (1.0-6.0*y**2)*x**2*(1.0-x**2))

mg.init_RHS(rhs(mg.x2d, mg.y2d))
```

```
Source norm =  1.09751581367
```

The last setup step is to initialize the solution–this is the starting point for the solve. Usually we just want to start with all zeros, so we use the `init_zeros()` method

[5]:
```
mg.init_zeros()
```

we can now solve – there are actually two different techniques we can do here. We can just do pure smoothing on the solution grid using `mg.smooth(mg.nlevels-1, N)`, where `N` is the number of smoothing iterations. To get the solution `N` will need to be large and this will take a long time.

Multigrid accelerates the smoothing. We can do a V-cycle multigrid solution using `mg.solve()`

[6]:
```
mg.solve()
```

```
source norm =  1.09751581367
«< beginning V-cycle (cycle 1) »>

  level: 7, grid: 256 x 256
  before G-S, residual L2: 1.097515813669473
  after G-S, residual L2: 1.502308451578657

  level: 6, grid: 128 x 128
  before G-S, residual L2: 1.0616243965458263
  after G-S, residual L2: 1.4321452257629033

  level: 5, grid: 64 x 64
  before G-S, residual L2: 1.011366277976364
  after G-S, residual L2: 1.281872470375375

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.903531158162907
  after G-S, residual L2: 0.9607576999783505

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.6736112182020367
  after G-S, residual L2: 0.4439774050299674

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.30721142286171554
  after G-S, residual L2: 0.0727215591269748

  level: 1, grid: 4 x 4
  before G-S, residual L2: 0.04841813458618458
  after G-S, residual L2: 3.9610700301811246e-05

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 3.925006722484123e-05
  after G-S, residual L2: 1.0370099820862674e-09

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.07010129273961899
  after G-S, residual L2: 0.0008815704830693547

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.4307377377402105
  after G-S, residual L2: 0.007174899576794818

  level: 4, grid: 32 x 32
```

(continues on next page)

(continued from previous page)

```
  before G-S, residual L2: 0.911086486792154
  after G-S, residual L2: 0.01618756602227813

  level: 5, grid: 64 x 64
  before G-S, residual L2: 1.1945438349788615
  after G-S, residual L2: 0.022021327892004925

  level: 6, grid: 128 x 128
  before G-S, residual L2: 1.313456560108626
  after G-S, residual L2: 0.02518650395173617

  level: 7, grid: 256 x 256
  before G-S, residual L2: 1.3618314516335004
  after G-S, residual L2: 0.026870007568672097

cycle 1: relative err = 0.999999999999964, residual err = 0.02448256984911586

«< beginning V-cycle (cycle 2) »>

  level: 7, grid: 256 x 256
  before G-S, residual L2: 0.026870007568672097
  after G-S, residual L2: 0.025790216249923552

  level: 6, grid: 128 x 128
  before G-S, residual L2: 0.018218080204017304
  after G-S, residual L2: 0.023654310121915274

  level: 5, grid: 64 x 64
  before G-S, residual L2: 0.01669077991582338
  after G-S, residual L2: 0.01977335201785163

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.013922595404814862
  after G-S, residual L2: 0.013577568890182053

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.009518306167970536
  after G-S, residual L2: 0.006115159484497302

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.004244630812032651
  after G-S, residual L2: 0.0010674120586864006

  level: 1, grid: 4 x 4
  before G-S, residual L2: 0.0007108144252738053
  after G-S, residual L2: 5.818246254772703e-07

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 5.765281065294632e-07
  after G-S, residual L2: 1.5231212503339452e-11

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.0010291471590693868
  after G-S, residual L2: 1.2950948742201083e-05
```

(continues on next page)

(continued from previous page)

```
  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.006239446983842889
  after G-S, residual L2: 0.00010483463130232172

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.014573363314854
  after G-S, residual L2: 0.00026233988398787004

  level: 5, grid: 64 x 64
  before G-S, residual L2: 0.021564270263952755
  after G-S, residual L2: 0.0003944827058086955

  level: 6, grid: 128 x 128
  before G-S, residual L2: 0.02579092712136628
  after G-S, residual L2: 0.00048636495715121916

  level: 7, grid: 256 x 256
  before G-S, residual L2: 0.028051324215592862
  after G-S, residual L2: 0.0005440874957950154

cycle 2: relative err = 13.739483825281054, residual err = 0.0004957445615074047

«< beginning V-cycle (cycle 3) »>

  level: 7, grid: 256 x 256
  before G-S, residual L2: 0.0005440874957950154
  after G-S, residual L2: 0.0005095844930046698

  level: 6, grid: 128 x 128
  before G-S, residual L2: 0.0003597879816772893
  after G-S, residual L2: 0.00044648485218937167

  level: 5, grid: 64 x 64
  before G-S, residual L2: 0.0003147892995472901
  after G-S, residual L2: 0.0003492541721056348

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.0002457276904804801
  after G-S, residual L2: 0.00022232862524233384

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.0001558932199490972
  after G-S, residual L2: 9.511093023364078e-05

  level: 2, grid: 8 x 8
  before G-S, residual L2: 6.616899520585456e-05
  after G-S, residual L2: 1.711006102346096e-05

  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.139522901981679e-05
  after G-S, residual L2: 9.33004809910226e-09

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
```

(continues on next page)

(continued from previous page)

```
  before G-S, residual L2: 9.245125097272049e-09
  after G-S, residual L2: 2.442311694447821e-13

  level: 2, grid: 8 x 8
  before G-S, residual L2: 1.64991725637487e-05
  after G-S, residual L2: 2.0771258971860784e-07

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.00010097720436460624
  after G-S, residual L2: 1.7241727900979902e-06

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.0002575410544503153
  after G-S, residual L2: 4.766282851613449e-06

  level: 5, grid: 64 x 64
  before G-S, residual L2: 0.00041133882050328275
  after G-S, residual L2: 7.600616845344458e-06

  level: 6, grid: 128 x 128
  before G-S, residual L2: 0.0005232809692242086
  after G-S, residual L2: 9.860758095018993e-06

  level: 7, grid: 256 x 256
  before G-S, residual L2: 0.0005945070122423073
  after G-S, residual L2: 1.1466134915427874e-05

cycle 3: relative err = 34.347638624909216, residual err = 1.0447352805871284e-05

«< beginning V-cycle (cycle 4) »>

  level: 7, grid: 256 x 256
  before G-S, residual L2: 1.1466134915427874e-05
  after G-S, residual L2: 1.054466722279011e-05

  level: 6, grid: 128 x 128
  before G-S, residual L2: 7.442814693866286e-06
  after G-S, residual L2: 8.955050475722099e-06

  level: 5, grid: 64 x 64
  before G-S, residual L2: 6.311313968968047e-06
  after G-S, residual L2: 6.734553609148436e-06

  level: 4, grid: 32 x 32
  before G-S, residual L2: 4.737984987500691e-06
  after G-S, residual L2: 4.091799997658277e-06

  level: 3, grid: 16 x 16
  before G-S, residual L2: 2.871028473858937e-06
  after G-S, residual L2: 1.6319551993366253e-06

  level: 2, grid: 8 x 8
  before G-S, residual L2: 1.1372178077508109e-06
  after G-S, residual L2: 2.961040430099916e-07

  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.9721864323458624e-07
```

(continues on next page)

(continued from previous page)

```
  after G-S, residual L2: 1.61503943872384e-10

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.6003411195777404e-10
  after G-S, residual L2: 4.2274326344473505e-15

  level: 2, grid: 8 x 8
  before G-S, residual L2: 2.855691101825338e-07
  after G-S, residual L2: 3.5961118754371857e-09

  level: 3, grid: 16 x 16
  before G-S, residual L2: 1.7893831203170535e-06
  after G-S, residual L2: 3.1136282101831173e-08

  level: 4, grid: 32 x 32
  before G-S, residual L2: 4.97129807196115e-06
  after G-S, residual L2: 9.544819739422644e-08

  level: 5, grid: 64 x 64
  before G-S, residual L2: 8.281644276572538e-06
  after G-S, residual L2: 1.56637783149839e-07

  level: 6, grid: 128 x 128
  before G-S, residual L2: 1.0888850082357996e-05
  after G-S, residual L2: 2.0777271327080248e-07

  level: 7, grid: 256 x 256
  before G-S, residual L2: 1.2717522622400765e-05
  after G-S, residual L2: 2.464531349025277e-07

cycle 4: relative err = 0.17409776671446628, residual err = 2.24555429482631e-07

«< beginning V-cycle (cycle 5) »>

  level: 7, grid: 256 x 256
  before G-S, residual L2: 2.464531349025277e-07
  after G-S, residual L2: 2.2491138140311698e-07

  level: 6, grid: 128 x 128
  before G-S, residual L2: 1.5874562191875262e-07
  after G-S, residual L2: 1.886249099391391e-07

  level: 5, grid: 64 x 64
  before G-S, residual L2: 1.3294481979637655e-07
  after G-S, residual L2: 1.397710191717015e-07

  level: 4, grid: 32 x 32
  before G-S, residual L2: 9.836928907527788e-08
  after G-S, residual L2: 8.269030961692836e-08

  level: 3, grid: 16 x 16
  before G-S, residual L2: 5.8062531341283565e-08
  after G-S, residual L2: 3.034725896415429e-08
```

(continues on next page)

(continued from previous page)

```
  level: 2, grid: 8 x 8
  before G-S, residual L2: 2.116912379336852e-08
  after G-S, residual L2: 5.467519592468213e-09

  level: 1, grid: 4 x 4
  before G-S, residual L2: 3.6418116003284676e-09
  after G-S, residual L2: 2.982625229812215e-12

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 2.955484162036181e-12
  after G-S, residual L2: 7.806739482450516e-17

  level: 2, grid: 8 x 8
  before G-S, residual L2: 5.273610709946236e-09
  after G-S, residual L2: 6.642323465658688e-11

  level: 3, grid: 16 x 16
  before G-S, residual L2: 3.4146989205844565e-08
  after G-S, residual L2: 6.052228076583688e-10

  level: 4, grid: 32 x 32
  before G-S, residual L2: 1.031248597196911e-07
  after G-S, residual L2: 2.0541497445308587e-09

  level: 5, grid: 64 x 64
  before G-S, residual L2: 1.7585349306604133e-07
  after G-S, residual L2: 3.421022608879089e-09

  level: 6, grid: 128 x 128
  before G-S, residual L2: 2.3383756442516674e-07
  after G-S, residual L2: 4.552170797983864e-09

  level: 7, grid: 256 x 256
  before G-S, residual L2: 2.7592842790687426e-07
  after G-S, residual L2: 5.41488950707315e-09

cycle 5: relative err = 0.005391244339065405, residual err = 4.933769007818501e-09

«< beginning V-cycle (cycle 6) »>

  level: 7, grid: 256 x 256
  before G-S, residual L2: 5.41488950707315e-09
  after G-S, residual L2: 4.948141362729419e-09

  level: 6, grid: 128 x 128
  before G-S, residual L2: 3.4929583962703016e-09
  after G-S, residual L2: 4.154445183511443e-09

  level: 5, grid: 64 x 64
  before G-S, residual L2: 2.9288841397931397e-09
  after G-S, residual L2: 3.074779198797186e-09

  level: 4, grid: 32 x 32
  before G-S, residual L2: 2.164991235492634e-09
```

(continues on next page)

(continued from previous page)

```
  after G-S, residual L2: 1.788028730183651e-09

  level: 3, grid: 16 x 16
  before G-S, residual L2: 1.2562223343389894e-09
  after G-S, residual L2: 6.021983813990021e-10

  level: 2, grid: 8 x 8
  before G-S, residual L2: 4.2028073688787063e-10
  after G-S, residual L2: 1.0655724637281067e-10

  level: 1, grid: 4 x 4
  before G-S, residual L2: 7.097871736854444e-11
  after G-S, residual L2: 5.813506543301849e-14

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 5.760611936011378e-14
  after G-S, residual L2: 1.521555112430923e-18

  level: 2, grid: 8 x 8
  before G-S, residual L2: 1.027891920456506e-10
  after G-S, residual L2: 1.294879454701896e-12

  level: 3, grid: 16 x 16
  before G-S, residual L2: 6.914011940773812e-10
  after G-S, residual L2: 1.2453691230551983e-11

  level: 4, grid: 32 x 32
  before G-S, residual L2: 2.2570491487662195e-09
  after G-S, residual L2: 4.639035392364569e-11

  level: 5, grid: 64 x 64
  before G-S, residual L2: 3.908967396962745e-09
  after G-S, residual L2: 7.803740782474827e-11

  level: 6, grid: 128 x 128
  before G-S, residual L2: 5.196394306272565e-09
  after G-S, residual L2: 1.033274523100204e-10

  level: 7, grid: 256 x 256
  before G-S, residual L2: 6.117636729623554e-09
  after G-S, residual L2: 1.2199402602477584e-10

cycle 6: relative err = 7.51413991329132e-05, residual err = 1.111546863428753e-10

«< beginning V-cycle (cycle 7) »>

  level: 7, grid: 256 x 256
  before G-S, residual L2: 1.2199402602477584e-10
  after G-S, residual L2: 1.121992266879251e-10

  level: 6, grid: 128 x 128
  before G-S, residual L2: 7.921861122082639e-11
  after G-S, residual L2: 9.493449600138316e-11
```

(continues on next page)

(continued from previous page)

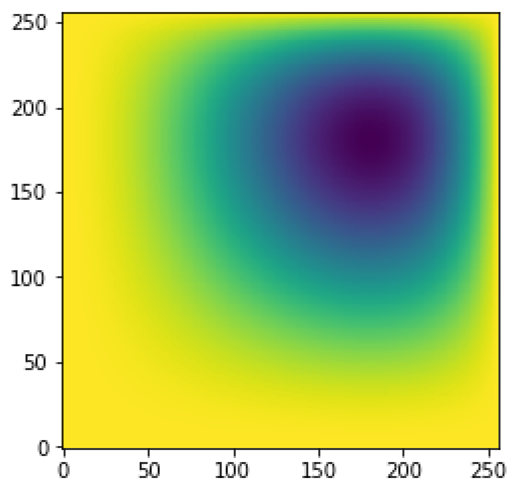
```
  level: 5, grid: 64 x 64
  before G-S, residual L2: 6.694993398453784e-11
  after G-S, residual L2: 7.050995614737483e-11

  level: 4, grid: 32 x 32
  before G-S, residual L2: 4.9666563586565975e-11
  after G-S, residual L2: 4.045094776680348e-11

  level: 3, grid: 16 x 16
  before G-S, residual L2: 2.843147343834713e-11
  after G-S, residual L2: 1.2576313722677801e-11

  level: 2, grid: 8 x 8
  before G-S, residual L2: 8.777954081387978e-12
  after G-S, residual L2: 2.170559196862902e-12

  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.445876195415056e-12
  after G-S, residual L2: 1.1842925278593641e-15

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.1735184729034125e-15
  after G-S, residual L2: 3.0994757710835167e-20

  level: 2, grid: 8 x 8
  before G-S, residual L2: 2.094012660676073e-12
  after G-S, residual L2: 2.6382579574150587e-14

  level: 3, grid: 16 x 16
  before G-S, residual L2: 1.466147487151147e-11
  after G-S, residual L2: 2.6760553592700965e-13

  level: 4, grid: 32 x 32
  before G-S, residual L2: 5.130705216489902e-11
  after G-S, residual L2: 1.0810419626613159e-12

  level: 5, grid: 64 x 64
  before G-S, residual L2: 9.001551103280705e-11
  after G-S, residual L2: 1.8342879121275396e-12

  level: 6, grid: 128 x 128
  before G-S, residual L2: 1.1914921193827463e-10
  after G-S, residual L2: 2.4124327865487605e-12

  level: 7, grid: 256 x 256
  before G-S, residual L2: 1.3907209384461257e-10
  after G-S, residual L2: 2.8429898342353533e-12

cycle 7: relative err = 7.062255558417692e-07, residual err = 2.590386214782638e-12
```

We can access the solution on the finest grid using `get_solution()`

```
[7]: phi = mg.get_solution()
```

```
[8]: plt.imshow(np.transpose(phi.v()), origin="lower")
```

```
[8]: <matplotlib.image.AxesImage at 0x7f7c47a0cc50>
```

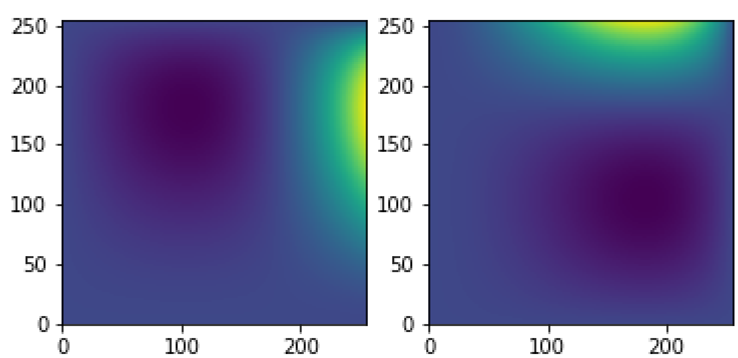


we can also get the gradient of the solution

```
[9]: gx, gy = mg.get_solution_gradient()
```

```
[10]: plt.subplot(121)
plt.imshow(np.transpose(gx.v()), origin="lower")
plt.subplot(122)
plt.imshow(np.transpose(gy.v()), origin="lower")
```

```
[10]: <matplotlib.image.AxesImage at 0x7f7c47a345f8>
```



14.2 General linear elliptic equation

The `GeneralMG2d` class implements support for a general elliptic equation of the form:

$$\alpha\phi + \nabla \cdot (\beta\nabla\phi) + \gamma \cdot \nabla\phi = f$$

with inhomogeneous boundary condtions.

It subclasses the `CellCenterMG2d` class, and the basic interface is the same

We will solve the above with

$$\alpha = 10 \tag{14.1}$$
$$\beta = xy + 1 \tag{14.2}$$
$$\gamma = \hat{x} + \hat{y} \tag{14.3}$$

and

$$f = -\frac{\pi}{2}(x+1)\sin\left(\frac{\pi y}{2}\right)\cos\left(\frac{\pi x}{2}\right) \tag{14.4}$$
$$-\frac{\pi}{2}(y+1)\sin\left(\frac{\pi x}{2}\right)\cos\left(\frac{\pi y}{2}\right) \tag{14.5}$$
$$+\left(\frac{-\pi^2(xy+1)}{2}+10\right)\cos\left(\frac{\pi x}{2}\right)\cos\left(\frac{\pi y}{2}\right) \tag{14.6}$$

on $[0,1]\times[0,1]$ with boundary conditions:

$$\phi(x=0) = \cos(\pi y/2) \tag{14.7}$$
$$\phi(x=1) = 0 \tag{14.8}$$
$$\phi(y=0) = \cos(\pi x/2) \tag{14.9}$$
$$\phi(y=1) = 0 \tag{14.10}$$

This has the exact solution:

$$\phi = \cos(\pi x/2)\cos(\pi y/2)$$

[11]:
```
import multigrid.general_MG as gMG
```

For reference, we'll define a function providing the analytic solution

[12]:
```
def true(x,y):
    return np.cos(np.pi*x/2.0)*np.cos(np.pi*y/2.0)
```

Now the coefficents–note that since γ is a vector, we have a different function for each component

[13]:
```
def alpha(x,y):
    return 10.0*np.ones_like(x)

def beta(x,y):
    return x*y + 1.0

def gamma_x(x,y):
    return np.ones_like(x)

def gamma_y(x,y):
    return np.ones_like(x)
```

and the righthand side function

[14]:
```
def f(x,y):
    return -0.5*np.pi*(x + 1.0)*np.sin(np.pi*y/2.0)*np.cos(np.pi*x/2.0) - \
            0.5*np.pi*(y + 1.0)*np.sin(np.pi*x/2.0)*np.cos(np.pi*y/2.0) + \
            (-np.pi**2*(x*y+1.0)/2.0 + 10.0)*np.cos(np.pi*x/2.0)*np.cos(np.pi*y/2.0)
```

Our inhomogeneous boundary conditions require a function that can be evaluated on the boundary to give the value

[15]:
```
def xl_func(y):
    return np.cos(np.pi*y/2.0)

def yl_func(x):
    return np.cos(np.pi*x/2.0)
```

Now we can setup our grid object and the coefficients, which are stored as a `CellCenter2d` object. Note, the coefficients do not need to have the same boundary conditions as ϕ (and for real problems, they may not). The one that matters the most is β, since that will need to be averaged to the edges of the domain, so the boundary conditions on the coefficients are important.

Here we use Neumann boundary conditions

[16]:
```
import mesh.patch as patch

nx = ny = 128

g = patch.Grid2d(nx, ny, ng=1)
d = patch.CellCenterData2d(g)

bc_c = bnd.BC(xlb="neumann", xrb="neumann",
              ylb="neumann", yrb="neumann")

d.register_var("alpha", bc_c)
d.register_var("beta", bc_c)
d.register_var("gamma_x", bc_c)
d.register_var("gamma_y", bc_c)
d.create()

a = d.get_var("alpha")
a[:,:] = alpha(g.x2d, g.y2d)

b = d.get_var("beta")
b[:,:] = beta(g.x2d, g.y2d)

gx = d.get_var("gamma_x")
gx[:,:] = gamma_x(g.x2d, g.y2d)

gy = d.get_var("gamma_y")
gy[:,:] = gamma_y(g.x2d, g.y2d)
```

Now we can setup the multigrid object

[17]:
```
a = gMG.GeneralMG2d(nx, ny,
                    xl_BC_type="dirichlet", yl_BC_type="dirichlet",
                    xr_BC_type="dirichlet", yr_BC_type="dirichlet",
                    xl_BC=xl_func,
                    yl_BC=yl_func,
                    coeffs=d,
                    verbose=1, vis=0, true_function=true)
```

```
cc data: nx = 2, ny = 2, ng = 1
         nvars = 7
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
```

(continues on next page)

(continued from previous page)

```
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
           alpha: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
            beta: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_x: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_y: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann

cc data: nx = 4, ny = 4, ng = 1
         nvars = 7
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
           alpha: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
            beta: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_x: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_y: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann

cc data: nx = 8, ny = 8, ng = 1
         nvars = 7
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
           alpha: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
            beta: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_x: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_y: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
```

(continues on next page)

(continued from previous page)

```
cc data: nx = 16, ny = 16, ng = 1
         nvars = 7
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
           alpha: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
            beta: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_x: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_y: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann

cc data: nx = 32, ny = 32, ng = 1
         nvars = 7
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
           alpha: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
            beta: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_x: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_y: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann

cc data: nx = 64, ny = 64, ng = 1
         nvars = 7
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
           alpha: min:    0.0000000000    max:    0.0000000000
                  BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
            beta: min:    0.0000000000    max:    0.0000000000
```

(continues on next page)

(continued from previous page)

```
                   BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_x: min:    0.0000000000    max:    0.0000000000
                   BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_y: min:    0.0000000000    max:    0.0000000000
                   BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann

cc data: nx = 128, ny = 128, ng = 1
         nvars = 7
         variables:
               v: min:    0.0000000000    max:    0.0000000000
                   BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               f: min:    0.0000000000    max:    0.0000000000
                   BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
               r: min:    0.0000000000    max:    0.0000000000
                   BCs: -x: dirichlet    +x: dirichlet    -y: dirichlet    +y:␣
↪dirichlet
           alpha: min:    0.0000000000    max:    0.0000000000
                   BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
            beta: min:    0.0000000000    max:    0.0000000000
                   BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_x: min:    0.0000000000    max:    0.0000000000
                   BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
         gamma_y: min:    0.0000000000    max:    0.0000000000
                   BCs: -x: neumann      +x: neumann      -y: neumann      +y: neumann
```

just as before, we specify the righthand side and initialize the solution

[18]:
```
a.init_zeros()
a.init_RHS(f(a.x2d, a.y2d))
```

```
Source norm =  1.77518149234
```

and we can solve it

[19]:
```
a.solve(rtol=1.e-10)
```

```
source norm =  1.77518149234
«< beginning V-cycle (cycle 1) »>

  level: 6, grid: 128 x 128
  before G-S, residual L2: 1.775181492337501
  after G-S, residual L2: 188.9332667507471

  level: 5, grid: 64 x 64
  before G-S, residual L2: 129.93801550392874
  after G-S, residual L2: 56.28708770794368

  level: 4, grid: 32 x 32
  before G-S, residual L2: 38.88692621665778
  after G-S, residual L2: 18.722754099081875

  level: 3, grid: 16 x 16
  before G-S, residual L2: 12.92606814051491
  after G-S, residual L2: 6.741858401611561
```

(continues on next page)

(continued from previous page)

```
  level: 2, grid: 8 x 8
  before G-S, residual L2: 4.646478379380238
  after G-S, residual L2: 2.065126154146587

  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.3745334259197384
  after G-S, residual L2: 0.02244519721859215

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 0.031252520872477096
  after G-S, residual L2: 8.232822131685586e-05

  level: 2, grid: 8 x 8
  before G-S, residual L2: 2.8059768631102893
  after G-S, residual L2: 0.07481536016730024

  level: 3, grid: 16 x 16
  before G-S, residual L2: 8.772402436595382
  after G-S, residual L2: 0.24361942694526875

  level: 4, grid: 32 x 32
  before G-S, residual L2: 19.591011324351037
  after G-S, residual L2: 0.5448263647958976

  level: 5, grid: 64 x 64
  before G-S, residual L2: 50.4641088994847
  after G-S, residual L2: 1.3597629173942398

  level: 6, grid: 128 x 128
  before G-S, residual L2: 160.2131163846867
  after G-S, residual L2: 4.125142056231141

cycle 1: relative err = 0.9999999999999981, residual err = 2.3237860883730193

«< beginning V-cycle (cycle 2) »>

  level: 6, grid: 128 x 128
  before G-S, residual L2: 4.125142056231141
  after G-S, residual L2: 2.4247311846143957

  level: 5, grid: 64 x 64
  before G-S, residual L2: 1.6915411385849393
  after G-S, residual L2: 1.0486241094402862

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.7283416353571861
  after G-S, residual L2: 0.45548181093652995

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.3165327512850198
  after G-S, residual L2: 0.22128563126748008

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.15332496186655512
```

(continues on next page)

(continued from previous page)

```
  after G-S, residual L2: 0.0747196881784426

  level: 1, grid: 4 x 4
  before G-S, residual L2: 0.04974939187294444
  after G-S, residual L2: 0.0008133572860410457

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 0.0011325179143730458
  after G-S, residual L2: 2.98337783917788e-06

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.10152627387884022
  after G-S, residual L2: 0.0027007047002410374

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.29814672415595245
  after G-S, residual L2: 0.00819910795226899

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.5218848114624619
  after G-S, residual L2: 0.014956130961989498

  level: 5, grid: 64 x 64
  before G-S, residual L2: 0.9910630869231989
  after G-S, residual L2: 0.028422939317571984

  level: 6, grid: 128 x 128
  before G-S, residual L2: 2.044187745817752
  after G-S, residual L2: 0.058293826018797935

cycle 2: relative err = 0.036315310129800826, residual err = 0.032838234439926776

«< beginning V-cycle (cycle 3) »>

  level: 6, grid: 128 x 128
  before G-S, residual L2: 0.058293826018797935
  after G-S, residual L2: 0.0417201187072595

  level: 5, grid: 64 x 64
  before G-S, residual L2: 0.029246699093099564
  after G-S, residual L2: 0.023356326397591495

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.016306296792818056
  after G-S, residual L2: 0.012906629461195234

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.009011110787953703
  after G-S, residual L2: 0.007315262938908486

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.005081499522859323
  after G-S, residual L2: 0.002562526517155576
```

(continues on next page)

(continued from previous page)

```
  level: 1, grid: 4 x 4
  before G-S, residual L2: 0.0017064130732665692
  after G-S, residual L2: 2.7912387046731474e-05

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 3.886526925433118e-05
  after G-S, residual L2: 1.0238217009484441e-07

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.0034819145217789937
  after G-S, residual L2: 9.252096659805176e-05

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.01006499034870321
  after G-S, residual L2: 0.0002744054418255884

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.016032310448838724
  after G-S, residual L2: 0.0004558226543272663

  level: 5, grid: 64 x 64
  before G-S, residual L2: 0.024303743880186898
  after G-S, residual L2: 0.0007098551729201239

  level: 6, grid: 128 x 128
  before G-S, residual L2: 0.037775318915862
  after G-S, residual L2: 0.0011035122819927912

cycle 3: relative err = 0.0012532978372415335, residual err = 0.0006216334987470617

«< beginning V-cycle (cycle 4) »>

  level: 6, grid: 128 x 128
  before G-S, residual L2: 0.0011035122819927912
  after G-S, residual L2: 0.0008898317346917108

  level: 5, grid: 64 x 64
  before G-S, residual L2: 0.0006257398720776081
  after G-S, residual L2: 0.000607740119084607

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.00042604165447901086
  after G-S, residual L2: 0.00039767401825608673

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.0002784624522907369
  after G-S, residual L2: 0.00024268300992319052

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.0001688184030119159
  after G-S, residual L2: 8.63435239999583e-05

  level: 1, grid: 4 x 4
  before G-S, residual L2: 5.750132804390505e-05
```

(continues on next page)

(continued from previous page)

```
  after G-S, residual L2: 9.407985171344554e-07

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.3099714803222558e-06
  after G-S, residual L2: 3.450833950914012e-09

  level: 2, grid: 8 x 8
  before G-S, residual L2: 0.00011732421042687768
  after G-S, residual L2: 3.1157531467636086e-06

  level: 3, grid: 16 x 16
  before G-S, residual L2: 0.00033850867119400885
  after G-S, residual L2: 9.17760188796962e-06

  level: 4, grid: 32 x 32
  before G-S, residual L2: 0.0005249527904418192
  after G-S, residual L2: 1.4651643230958405e-05

  level: 5, grid: 64 x 64
  before G-S, residual L2: 0.0007080871923387015
  after G-S, residual L2: 2.0290645679943462e-05

  level: 6, grid: 128 x 128
  before G-S, residual L2: 0.0009185166830535544
  after G-S, residual L2: 2.6570300453995103e-05

cycle 4: relative err = 4.2574662963457396e-05, residual err = 1.4967652923762853e-05

«< beginning V-cycle (cycle 5) »>

  level: 6, grid: 128 x 128
  before G-S, residual L2: 2.6570300453995103e-05
  after G-S, residual L2: 2.3098223923757352e-05

  level: 5, grid: 64 x 64
  before G-S, residual L2: 1.6274857395354832e-05
  after G-S, residual L2: 1.7906142642175535e-05

  level: 4, grid: 32 x 32
  before G-S, residual L2: 1.258588239896169e-05
  after G-S, residual L2: 1.2880701433730278e-05

  level: 3, grid: 16 x 16
  before G-S, residual L2: 9.035061892671461e-06
  after G-S, residual L2: 8.10300318788889e-06

  level: 2, grid: 8 x 8
  before G-S, residual L2: 5.641504287378599e-06
  after G-S, residual L2: 2.9012129063955126e-06

  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.932169517574082e-06
  after G-S, residual L2: 3.161675601835735e-08
```

(continues on next page)

(continued from previous page)

```
  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 4.4023320992879136e-08
  after G-S, residual L2: 1.1596974313938014e-10

  level: 2, grid: 8 x 8
  before G-S, residual L2: 3.9422658747144435e-06
  after G-S, residual L2: 1.0466257645445924e-07

  level: 3, grid: 16 x 16
  before G-S, residual L2: 1.1405869020431955e-05
  after G-S, residual L2: 3.0819546585464564e-07

  level: 4, grid: 32 x 32
  before G-S, residual L2: 1.7696025211842327e-05
  after G-S, residual L2: 4.853326074858634e-07

  level: 5, grid: 64 x 64
  before G-S, residual L2: 2.281722184794443e-05
  after G-S, residual L2: 6.339093026629609e-07

  level: 6, grid: 128 x 128
  before G-S, residual L2: 2.7204506586512792e-05
  after G-S, residual L2: 7.61736677407384e-07

cycle 5: relative err = 1.4372233555992132e-06, residual err = 4.2910354839513e-07

«< beginning V-cycle (cycle 6) »>

  level: 6, grid: 128 x 128
  before G-S, residual L2: 7.61736677407384e-07
  after G-S, residual L2: 6.887955287148536e-07

  level: 5, grid: 64 x 64
  before G-S, residual L2: 4.858303580829294e-07
  after G-S, residual L2: 5.698844682533653e-07

  level: 4, grid: 32 x 32
  before G-S, residual L2: 4.011448592273346e-07
  after G-S, residual L2: 4.2887305175998083e-07

  level: 3, grid: 16 x 16
  before G-S, residual L2: 3.011320287970724e-07
  after G-S, residual L2: 2.7229135972437344e-07

  level: 2, grid: 8 x 8
  before G-S, residual L2: 1.8967555884605451e-07
  after G-S, residual L2: 9.770491553515245e-08

  level: 1, grid: 4 x 4
  before G-S, residual L2: 6.507167357899105e-08
  after G-S, residual L2: 1.0648579116334552e-09

  bottom solve:
  level: 0, grid: 2 x 2
```

(continues on next page)

(continued from previous page)

```
  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.4827137294363792e-09
  after G-S, residual L2: 3.9058805523605475e-12

  level: 2, grid: 8 x 8
  before G-S, residual L2: 1.3276705475319977e-07
  after G-S, residual L2: 3.524245793876337e-09

  level: 3, grid: 16 x 16
  before G-S, residual L2: 3.8563144896921417e-07
  after G-S, residual L2: 1.0398885077513769e-08

  level: 4, grid: 32 x 32
  before G-S, residual L2: 6.038836850187365e-07
  after G-S, residual L2: 1.6338312481157817e-08

  level: 5, grid: 64 x 64
  before G-S, residual L2: 7.682416346530921e-07
  after G-S, residual L2: 2.0772116210685317e-08

  level: 6, grid: 128 x 128
  before G-S, residual L2: 8.865086230602598e-07
  after G-S, residual L2: 2.401923227919822e-08

cycle 6: relative err = 4.8492598977484135e-08, residual err = 1.353057835656594e-08

«< beginning V-cycle (cycle 7) »>

  level: 6, grid: 128 x 128
  before G-S, residual L2: 2.401923227919822e-08
  after G-S, residual L2: 2.2125290070425652e-08

  level: 5, grid: 64 x 64
  before G-S, residual L2: 1.5613809613835955e-08
  after G-S, residual L2: 1.8869606239963252e-08

  level: 4, grid: 32 x 32
  before G-S, residual L2: 1.3292687837677291e-08
  after G-S, residual L2: 1.4485742520315527e-08

  level: 3, grid: 16 x 16
  before G-S, residual L2: 1.0177212111802273e-08
  after G-S, residual L2: 9.198083791538658e-09

  level: 2, grid: 8 x 8
  before G-S, residual L2: 6.409467335640698e-09
  after G-S, residual L2: 3.3018379633629456e-09

  level: 1, grid: 4 x 4
  before G-S, residual L2: 2.1990607567876347e-09
  after G-S, residual L2: 3.598750197454369e-11

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
```

(continues on next page)

(continued from previous page)

```
  before G-S, residual L2: 5.010919630110133e-11
  after G-S, residual L2: 1.3200151156453123e-13

  level: 2, grid: 8 x 8
  before G-S, residual L2: 4.48679228107323e-09
  after G-S, residual L2: 1.1908945622999935e-10

  level: 3, grid: 16 x 16
  before G-S, residual L2: 1.3081162779667808e-08
  after G-S, residual L2: 3.522982496836639e-10

  level: 4, grid: 32 x 32
  before G-S, residual L2: 2.0705037621548675e-08
  after G-S, residual L2: 5.546643639307605e-10

  level: 5, grid: 64 x 64
  before G-S, residual L2: 2.6280822057541362e-08
  after G-S, residual L2: 6.964954384251476e-10

  level: 6, grid: 128 x 128
  before G-S, residual L2: 2.994449911367404e-08
  after G-S, residual L2: 7.914383325620475e-10

cycle 7: relative err = 1.6392150533299687e-09, residual err = 4.4583516444840087e-10

«< beginning V-cycle (cycle 8) »>

  level: 6, grid: 128 x 128
  before G-S, residual L2: 7.914383325620475e-10
  after G-S, residual L2: 7.355629304356289e-10

  level: 5, grid: 64 x 64
  before G-S, residual L2: 5.19218220597571e-10
  after G-S, residual L2: 6.364663261794707e-10

  level: 4, grid: 32 x 32
  before G-S, residual L2: 4.485504928535875e-10
  after G-S, residual L2: 4.928237246176745e-10

  level: 3, grid: 16 x 16
  before G-S, residual L2: 3.4637122000064977e-10
  after G-S, residual L2: 3.1194162913950586e-10

  level: 2, grid: 8 x 8
  before G-S, residual L2: 2.174181615639314e-10
  after G-S, residual L2: 1.1194514367241423e-10

  level: 1, grid: 4 x 4
  before G-S, residual L2: 7.455734323986808e-11
  after G-S, residual L2: 1.2201499216239134e-12

  bottom solve:
  level: 0, grid: 2 x 2

  level: 1, grid: 4 x 4
  before G-S, residual L2: 1.6989436916357301e-12
  after G-S, residual L2: 4.475487188247986e-15
```

(continues on next page)

(continued from previous page)

```
  level: 2, grid: 8 x 8
  before G-S, residual L2: 1.521214490284944e-10
  after G-S, residual L2: 4.037434677870943e-12

  level: 3, grid: 16 x 16
  before G-S, residual L2: 4.4491498629640967e-10
  after G-S, residual L2: 1.197248120085576e-11

  level: 4, grid: 32 x 32
  before G-S, residual L2: 7.109792371905777e-10
  after G-S, residual L2: 1.8912335700376235e-11

  level: 5, grid: 64 x 64
  before G-S, residual L2: 9.034017109357381e-10
  after G-S, residual L2: 2.3606466325271617e-11

  level: 6, grid: 128 x 128
  before G-S, residual L2: 1.0238349148814258e-09
  after G-S, residual L2: 2.678477889744364e-11

cycle 8: relative err = 5.555107077431201e-11, residual err = 1.5088473495842003e-11
```

We can compare to the true solution

[20]:
```
v = a.get_solution()
b = true(a.x2d, a.y2d)
e = v - b
```

The norm of the error is

[21]:
```
e.norm()
```

[21]:
```
1.6719344048744095e-05
```

[ ]:
```

```

CHAPTER 15

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]

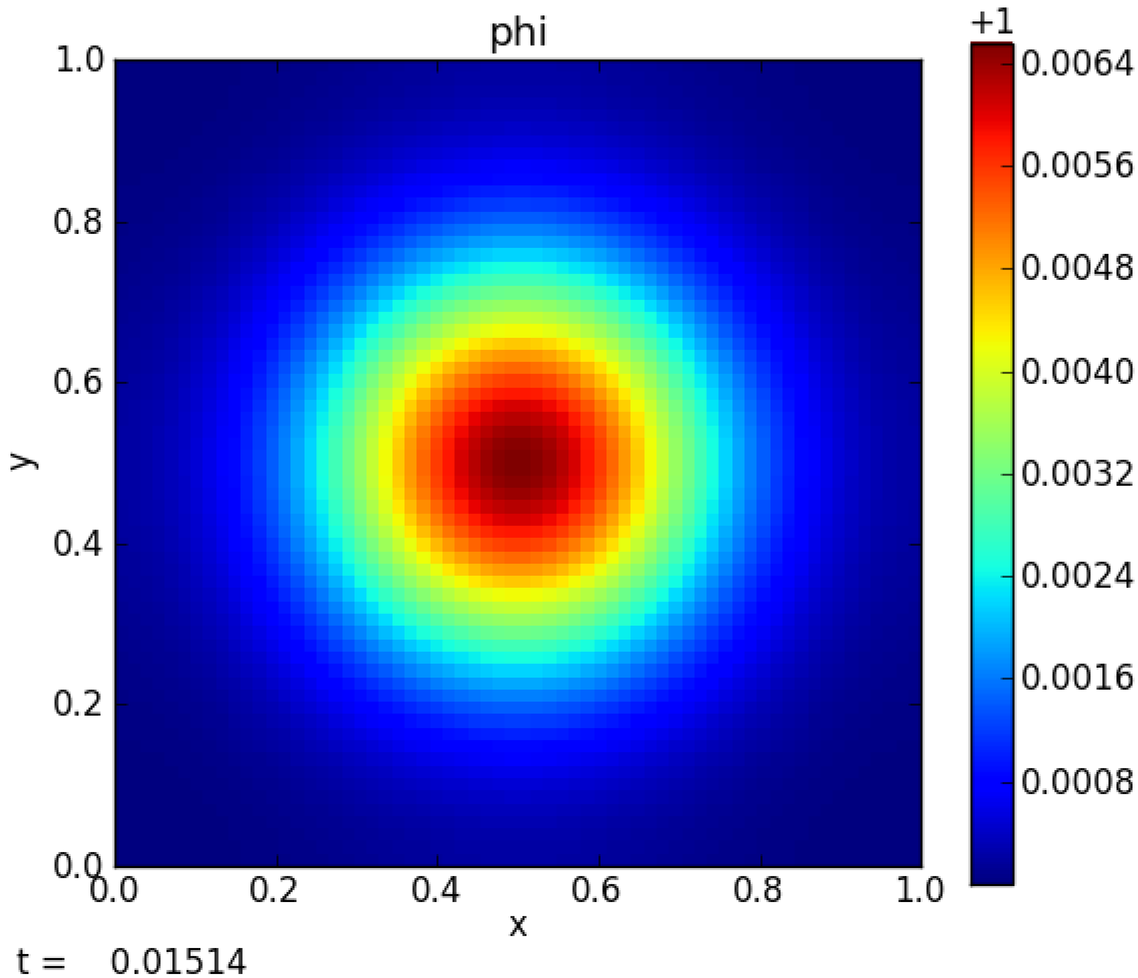
option	value	description
cfl	`0.8`	diffusion CFL number

15.1 Examples

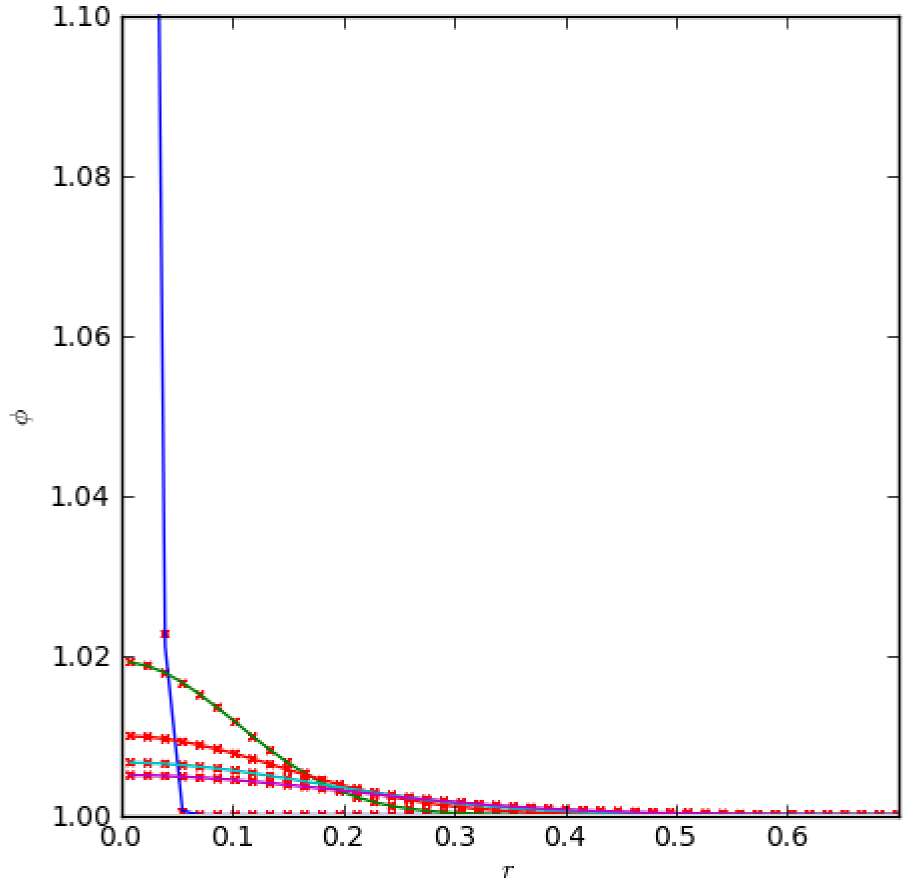
15.1.1 gaussian

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.



15.2 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.

15.2.1 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.

15.2.2 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 16

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]

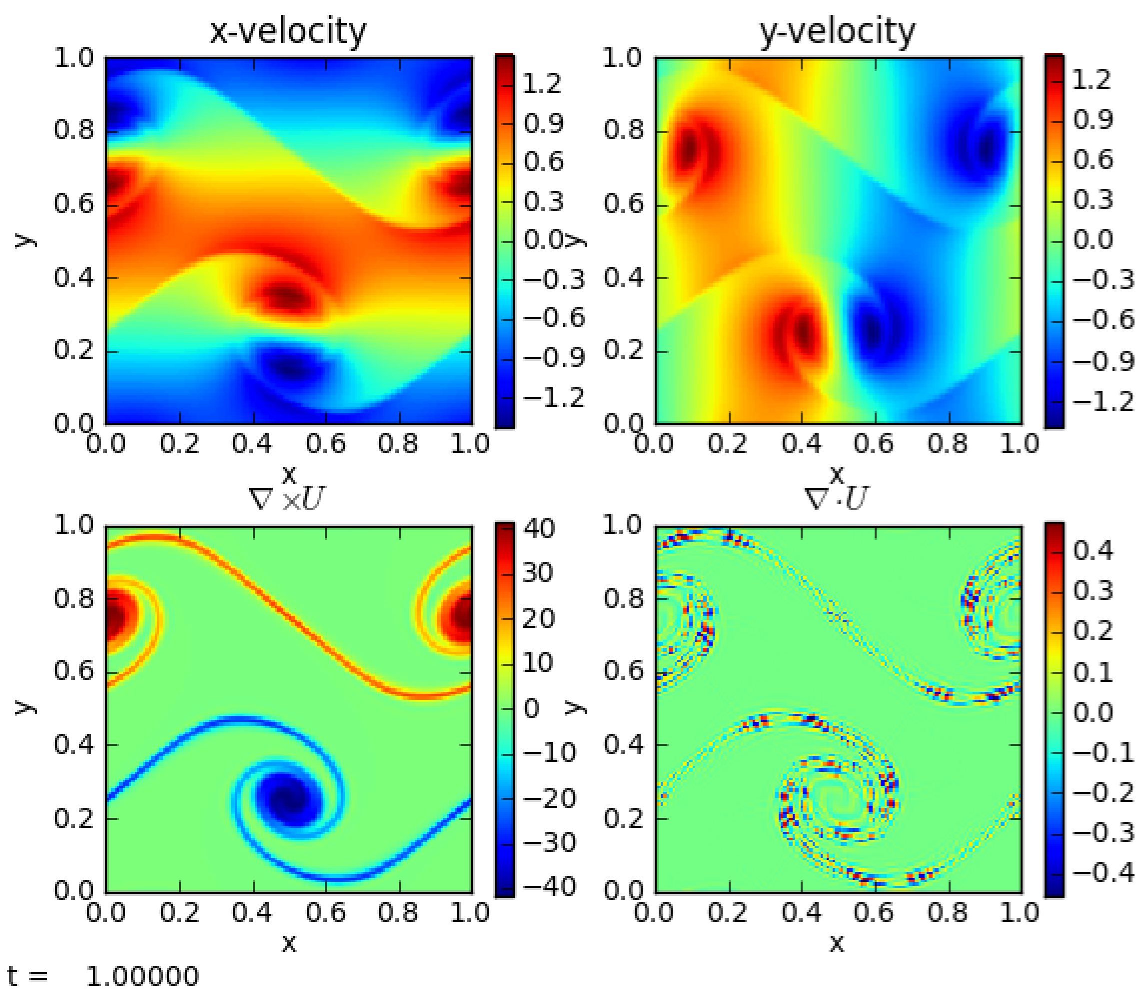
option	value	description
do_particles	`0`	
particle_generator	`grid`	

16.1 Examples

16.1.1 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
```



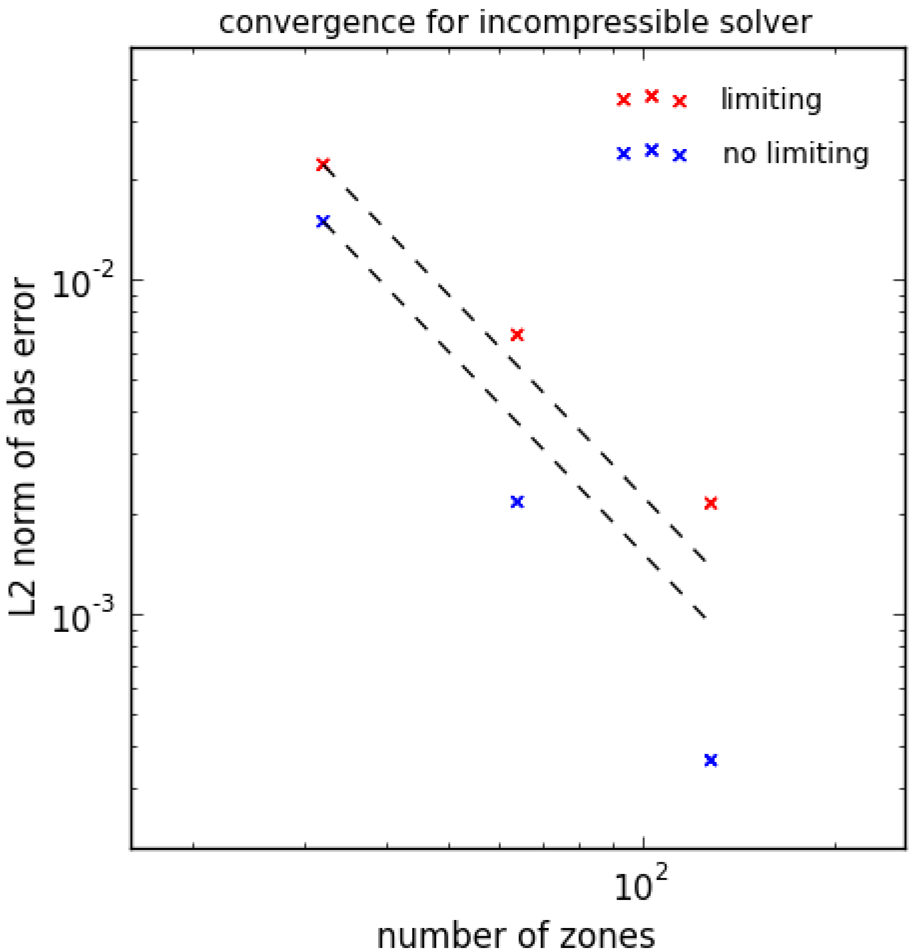
The vorticity panel (lower left) is what is usually shown in papers. Note that the velocity divergence is not zero—this is because we are using an approximate projection.

16.1.2 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.

16.2 Exercises

16.2.1 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.

16.2.2 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 ρ now enters).

16.3 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 17

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`	

17.1 Examples

17.1.1 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 18

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

18.1 Example problems

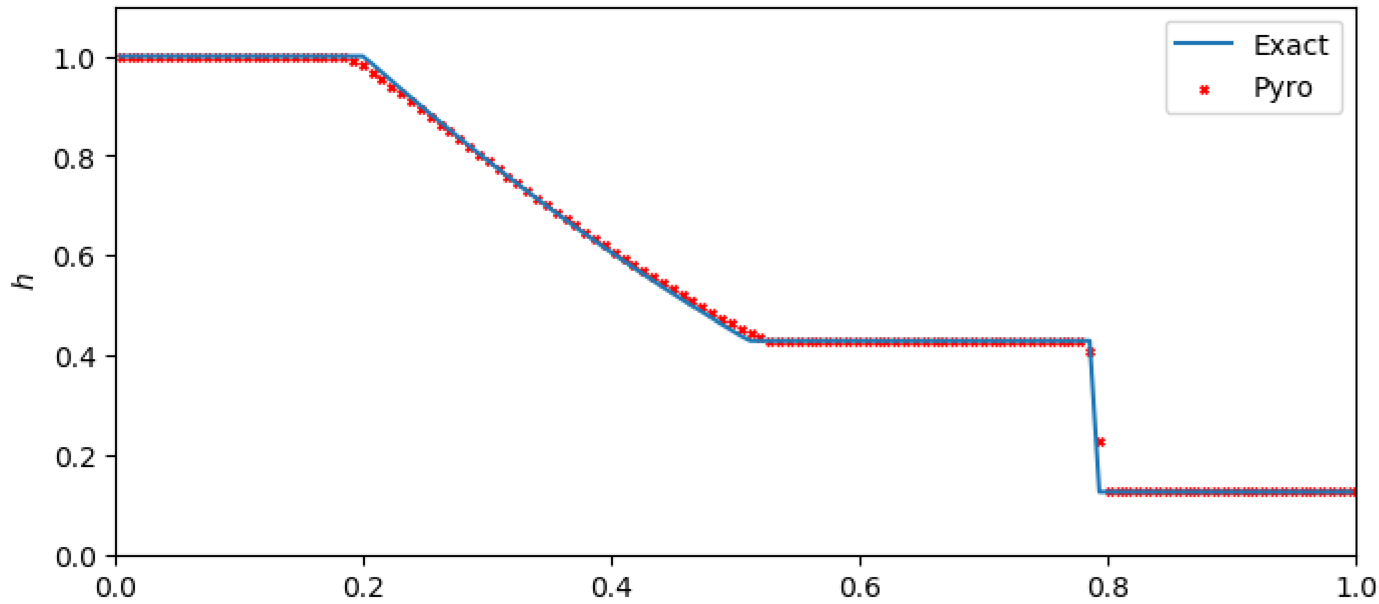
18.1.1 dam

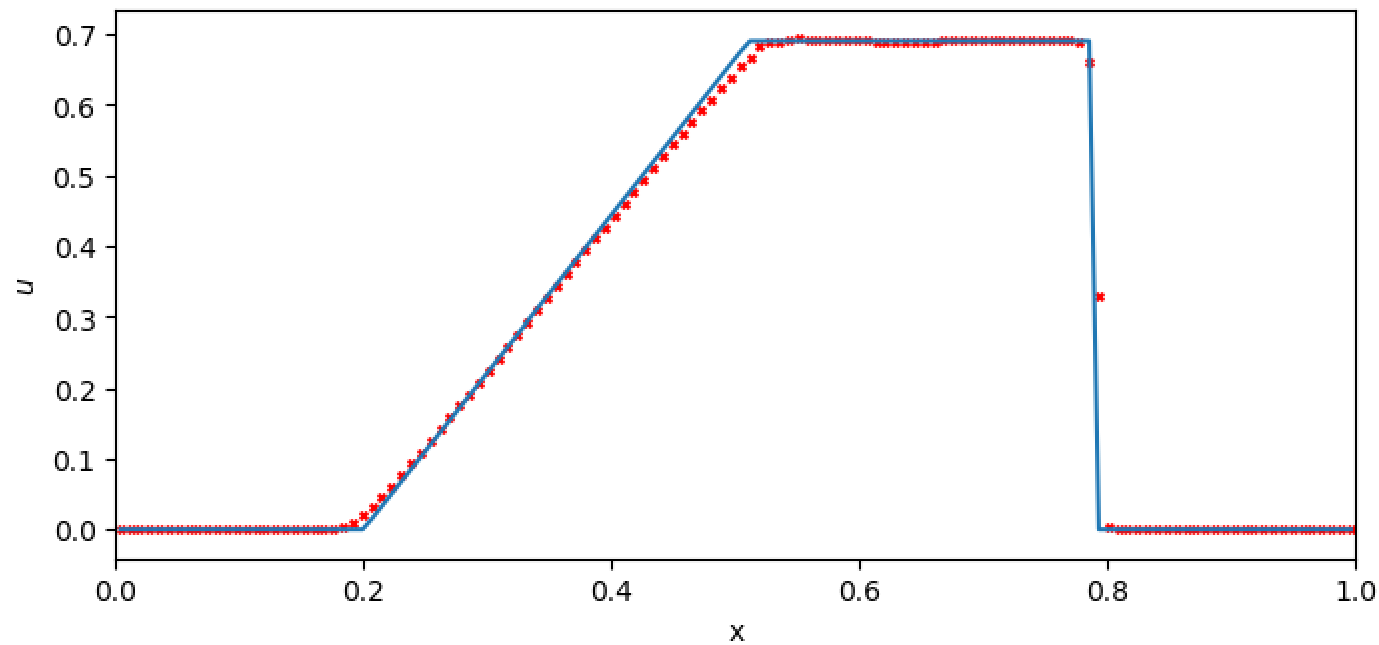
The dam break problem is a standard hydrodynamics problem, analagous to the Sod shock tube problem in compressible hydrodynamics. It considers a one-multidimensional problem of two regions of fluid at different heights, initially separated by a dam. The problem then models the evolution of the system 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.

18.1.2 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
```

18.1.3 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
```

18.2 Exercises

18.2.1 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?

18.2.2 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 19

Particles

A solver for modelling particles.

19.1 `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	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)
```

19.2 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:

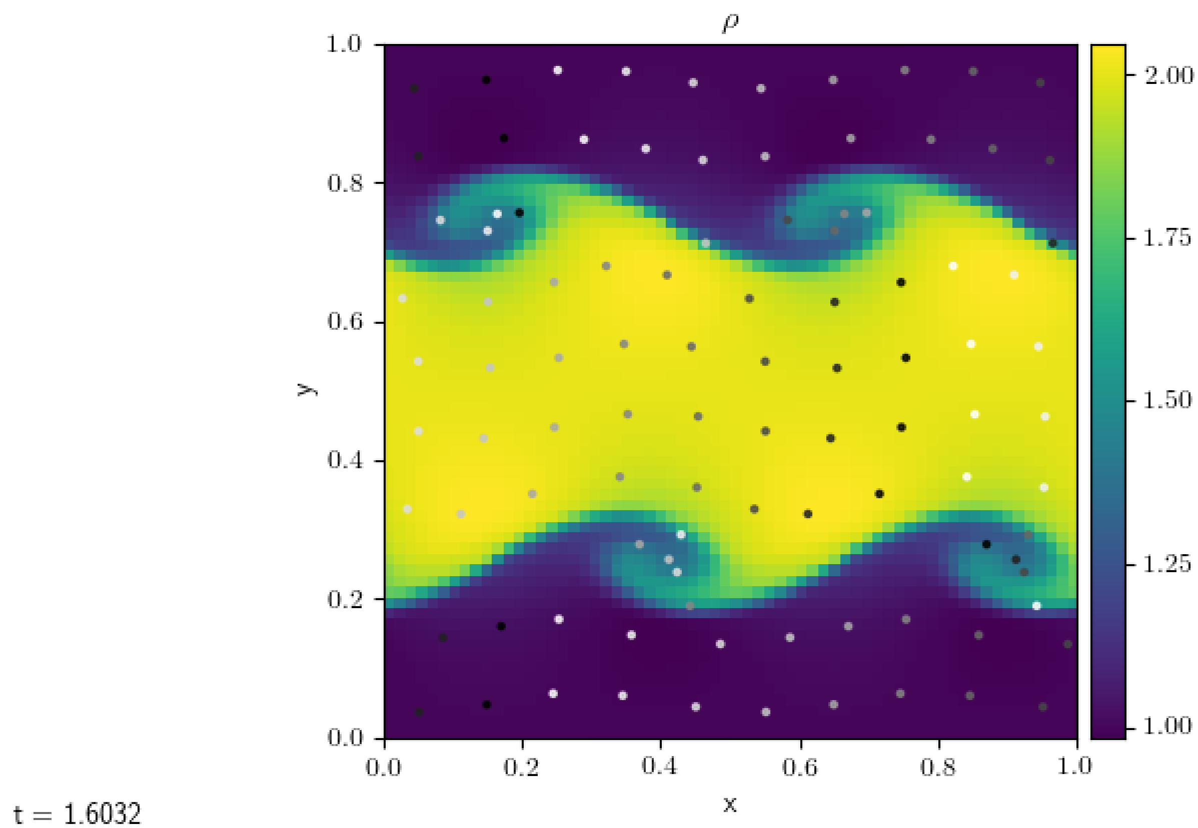
```
particle_positions = particles.get_positions()

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

# plot particles
ax.scatter(particle_positions[:, 0],
```

(continues on next page)

(continued from previous page)

```
    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.



CHAPTER 20

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 21

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.

21.1 Unit tests

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

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

21.2 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 22

Contributing and getting help

22.1 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.

22.2 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.

22.3 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 *master* 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.*

22.4 Mailing list

There is a public mailing list for discussing pyro. Visit the pyro-code@googlegroups.com and subscribe. You can then send questions to that list.

CHAPTER 23

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:2014,
   author = {{Zingale}, M.},
    title = "{pyro: A teaching code for computational astrophysical hydrodynamics}",
  journal = {Astronomy and Computing},
archivePrefix = "arXiv",
   eprint = {1306.6883},
 primaryClass = "astro-ph.IM",
 keywords = {Hydrodynamics, Methods: numerical},
     year = 2014,
    month = oct,
   volume = 6,
    pages = {52--62},
      doi = {10.1016/j.ascom.2014.07.003},
   adsurl = {http://adsabs.harvard.edu/abs/2014A%26C.....6...52Z},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
}
```

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

CHAPTER 24

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 25

pyro2

25.1 advection package

The pyro advection solver. This implements a second-order, unsplit method for linear advection based on the Colella 1990 paper.

25.1.1 Subpackages

advection.problems package

Submodules

advection.problems.smooth module

`advection.problems.smooth.`**`finalize`**`()`
print out any information to the user at the end of the run

`advection.problems.smooth.`**`init_data`**`(`*my_data*, *rp*`)`
initialize the smooth advection problem

advection.problems.test module

`advection.problems.test.`**`finalize`**`()`
print out any information to the user at the end of the run

`advection.problems.test.`**`init_data`**`(`*my_data*, *rp*`)`
an init routine for unit testing

advection.problems.tophat module

`advection.problems.tophat.finalize()`
print out any information to the user at the end of the run

`advection.problems.tophat.init_data(myd, rp)`
initialize the tophat advection problem

25.1.2 Submodules

25.1.3 advection.advective_fluxes module

`advection.advective_fluxes.unsplit_fluxes(my_data, rp, dt, scalar_name)`
Construct the fluxes through the interfaces for the linear advection equation:

$$a_t + ua_x + va_y = 0$$

We use a second-order (piecewise linear) unsplit Godunov method (following Colella 1990).

In the pure advection case, there is no Riemann problem we need to solve – we just simply do upwinding. So there is only one 'state' at each interface, and the zone the information comes from depends on the sign of the velocity.

Our convection is that the fluxes are going to be defined on the left edge of the computational zones:

```
 |             |             |             |
 |             |             |             |
-+------+------+------+------+------+------+--
 |     i-1     |      i      |     i+1     |

          a_l,i  a_r,i   a_l,i+1
```

a_r,i and a_l,i+1 are computed using the information in zone i,j.

Parameters

my_data [CellCenterData2d object] The data object containing the grid and advective scalar that we are advecting.

rp [RuntimeParameters object] The runtime parameters for the simulation

dt [float] The timestep we are advancing through.

scalar_name [str] The name of the variable contained in my_data that we are advecting

Returns

out [ndarray, ndarray] The fluxes on the x- and y-interfaces

25.1.4 advection.simulation module

class `advection.simulation.Simulation(solver_name, problem_name, rp, timers=None, data_class=<class 'mesh.patch.CellCenterData2d'>)`
Bases: *simulation_null.NullSimulation*

`dovis()`
Do runtime visualization.

evolve()
Evolve the linear advection equation through one timestep. We only consider the "density" variable in the CellCenterData2d object that is part of the Simulation.

initialize()
Initialize the grid and variables for advection and set the initial conditions for the chosen problem.

method_compute_timestep()
Compute the advective timestep (CFL) constraint. We use the driver.cfl parameter to control what fraction of the CFL step we actually take.

25.2 advection_fv4 package

The pyro fourth-order accurate advection solver. This implements a the method of McCorquodale and Colella (2011), with 4th order accurate spatial reconstruction together with 4th order Runge-Kutta time integration.

25.2.1 Subpackages

advection_fv4.problems package

Submodules

advection_fv4.problems.smooth module

advection_fv4.problems.smooth.**finalize**()
print out any information to the user at the end of the run

advection_fv4.problems.smooth.**init_data**(*my_data*, *rp*)
initialize the smooth advection problem

25.2.2 Submodules

25.2.3 advection_fv4.fluxes module

advection_fv4.fluxes.**fluxes**(*my_data*, *rp*, *dt*)
Construct the fluxes through the interfaces for the linear advection equation:

$$a_t + ua_x + va_y = 0$$

We use a fourth-order Godunov method to construct the interface states, using Runge-Kutta integration. Since this is 4th-order, we need to be aware of the difference between a face-average and face-center for the fluxes.

In the pure advection case, there is no Riemann problem we need to solve – we just simply do upwinding. So there is only one 'state' at each interface, and the zone the information comes from depends on the sign of the velocity.

Our convection is that the fluxes are going to be defined on the left edge of the computational zones:

```
 |             |             |             |
 |             |             |             |
-+------+------+------+------+------+------+--
 |     i-1     |      i      |     i+1     |
```

(continues on next page)

(continued from previous page)

```
        a_l,i  a_r,i   a_l,i+1
```

a_r,i and a_l,i+1 are computed using the information in zone i,j.

Parameters

my_data [FV object] The data object containing the grid and advective scalar that we are advecting.

rp [RuntimeParameters object] The runtime parameters for the simulation

dt [float] The timestep we are advancing through.

scalar_name [str] The name of the variable contained in my_data that we are advecting

Returns

out [ndarray, ndarray] The fluxes averaged over the x and y faces

25.2.4 advection_fv4.interface module

`advection_fv4.interface.`**`states`**

Predict the cell-centered state to the edges in one-dimension using the reconstructed, limited slopes. We use a fourth-order Godunov method.

Our convention here is that:

`al[i,j]` will be $al_{i-1/2,j}$,

`al[i+1,j]` will be $al_{i+1/2,j}$.

Parameters

a [ndarray] The cell-centered state.

ng [int] The number of ghost cells

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

Returns

out [ndarray, ndarray] The state predicted to the left and right edges.

`advection_fv4.interface.`**`states_nolimit`**

Predict the cell-centered state to the edges in one-dimension using the reconstructed slopes (and without slope limiting). We use a fourth-order Godunov method.

Our convention here is that:

`al[i,j]` will be $al_{i-1/2,j}$,

`al[i+1,j]` will be $al_{i+1/2,j}$.

Parameters

a [ndarray] The cell-centered state.

ng [int] The number of ghost cells

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

Returns

out [ndarray, ndarray] The state predicted to the left and right edges.

25.2.5 advection_fv4.simulation module

class advection_fv4.simulation.**Simulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: *advection_rk.simulation.Simulation*

initialize()

Initialize the grid and variables for advection and set the initial conditions for the chosen problem.

substep(*myd*)

take a single substep in the RK timestepping starting with the conservative state defined as part of myd

25.3 advection_nonuniform package

The pyro advection solver. This implements a second-order, unsplit method for linear advection based on the Colella 1990 paper.

25.3.1 Subpackages

advection_nonuniform.problems package

Submodules

advection_nonuniform.problems.slotted module

advection_nonuniform.problems.slotted.**finalize**()

print out any information to the user at the end of the run

advection_nonuniform.problems.slotted.**init_data**(*my_data*, *rp*)

initialize the slotted advection problem

25.3.2 Submodules

25.3.3 advection_nonuniform.advective_fluxes module

advection_nonuniform.advective_fluxes.**unsplit_fluxes**(*my_data*, *rp*, *dt*, *scalar_name*)

Construct the fluxes through the interfaces for the linear advection equation:

$$a_t + ua_x + va_y = 0$$

We use a second-order (piecewise linear) unsplit Godunov method (following Colella 1990).

In the pure advection case, there is no Riemann problem we need to solve – we just simply do upwinding. So there is only one 'state' at each interface, and the zone the information comes from depends on the sign of the velocity.

Our convection is that the fluxes are going to be defined on the left edge of the computational zones:

```
 |             |             |             |
 |             |             |             |
-+------+------+------+------+------+------+--
 |     i-1     |      i      |     i+1     |

          a_l,i  a_r,i   a_l,i+1
```

a_r,i and a_l,i+1 are computed using the information in zone i,j.

Parameters

my_data [CellCenterData2d object] The data object containing the grid and advective scalar that we are advecting.

rp [RuntimeParameters object] The runtime parameters for the simulation

dt [float] The timestep we are advancing through.

scalar_name [str] The name of the variable contained in my_data that we are advecting

Returns

out [ndarray, ndarray] The fluxes on the x- and y-interfaces

25.3.4 advection_nonuniform.simulation module

class advection_nonuniform.simulation.**Simulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: *simulation_null.NullSimulation*

dovis()

Do runtime visualization.

evolve()

Evolve the linear advection equation through one timestep. We only consider the "density" variable in the CellCenterData2d object that is part of the Simulation.

initialize()

Initialize the grid and variables for advection and set the initial conditions for the chosen problem.

method_compute_timestep()

The timestep() function computes the advective timestep (CFL) constraint. The CFL constraint says that information cannot propagate further than one zone per timestep.

We use the driver.cfl parameter to control what fraction of the CFL step we actually take.

25.4 advection_rk package

The pyro method-of-lines advection solver. This uses a piecewise linear reconstruction in space together with a Runge-Kutta integration for time.

25.4.1 Subpackages

25.4.2 Submodules

25.4.3 advection_rk.fluxes module

`advection_rk.fluxes.`**`fluxes`**`(`*my_data*, *rp*, *dt*`)`

Construct the fluxes through the interfaces for the linear advection equation:

$$a_t + ua_x + va_y = 0$$

We use a second-order (piecewise linear) Godunov method to construct the interface states, using Runge-Kutta integration. These are one-dimensional predictions to the interface, relying on the coupling in transverse directions through the intermediate stages of the Runge-Kutta integrator.

In the pure advection case, there is no Riemann problem we need to solve – we just simply do upwinding. So there is only one 'state' at each interface, and the zone the information comes from depends on the sign of the velocity.

Our convection is that the fluxes are going to be defined on the left edge of the computational zones:

```
 |             |             |             |
 |             |             |             |
-+------+------+------+------+------+------+--
 |     i-1     |      i      |     i+1     |

        a_l,i  a_r,i   a_l,i+1
```

a_r,i and a_l,i+1 are computed using the information in zone i,j.

Parameters

my_data [CellCenterData2d object] The data object containing the grid and advective scalar that we are advecting.

rp [RuntimeParameters object] The runtime parameters for the simulation

dt [float] The timestep we are advancing through.

scalar_name [str] The name of the variable contained in my_data that we are advecting

Returns

out [ndarray, ndarray] The fluxes on the x- and y-interfaces

25.4.4 advection_rk.simulation module

`class` `advection_rk.simulation.`**`Simulation`**`(`*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*`)`

Bases: *`advection.simulation.Simulation`*

`evolve``()`

Evolve the linear advection equation through one timestep. We only consider the "density" variable in the CellCenterData2d object that is part of the Simulation.

`method_compute_timestep``()`

Compute the advective timestep (CFL) constraint. We use the driver.cfl parameter to control what fraction of the CFL step we actually take.

substep (*myd*)
take a single substep in the RK timestepping starting with the conservative state defined as part of myd

25.5 advection_weno package

The pyro advection solver. This implements a finite difference Lax-Friedrichs flux split method with WENO reconstruction based on Shu's review from 1998 (https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19980007543.pdf) although the notation more follows Gerolymos et al (https://doi.org/10.1016/j.jcp.2009.07.039).

Most of the code is taken from advection_rk and toy-conslaw.

The general flow of the solver when invoked through pyro.py is:

- create grid
- initial conditions
- main loop
  - fill ghost cells
  - compute dt
  - compute fluxes
  - conservative update
  - output

25.5.1 Subpackages

25.5.2 Submodules

25.5.3 advection_weno.fluxes module

advection_weno.fluxes.**fluxes** (*my_data*, *rp*, *dt*)
Construct the fluxes through the interfaces for the linear advection equation

$$a_t + ua_x + va_y = 0$$

We use a high-order flux split WENO method to construct the interface fluxes. No Riemann problems are solved. The Lax-Friedrichs flux split will probably make it diffusive; the lack of a transverse solver also will not help.

Parameters

my_data [CellCenterData2d object] The data object containing the grid and advective scalar that we are advecting.

rp [RuntimeParameters object] The runtime parameters for the simulation

dt [float] The timestep we are advancing through.

scalar_name [str] The name of the variable contained in my_data that we are advecting

Returns

out [ndarray, ndarray] The fluxes on the x- and y-interfaces

advection_weno.fluxes.**fvs** (*q*, *order*, *u*, *alpha*)
Perform Flux-Vector-Split (LF) finite differencing using WENO in 1d.

Parameters

q [np array] input data with at least order+1 ghost zones

order [int] WENO order (k)

u [float] Advection velocity in this direction

alpha [float] Maximum characteristic speed

Returns

f [np array] flux

25.5.4 advection_weno.simulation module

class advection_weno.simulation.**Simulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: *advection.simulation.Simulation*

evolve()

Evolve the linear advection equation through one timestep. We only consider the "density" variable in the CellCenterData2d object that is part of the Simulation.

method_compute_timestep()

Compute the advective timestep (CFL) constraint. We use the driver.cfl parameter to control what fraction of the CFL step we actually take.

substep(*myd*)

take a single substep in the RK timestepping starting with the conservative state defined as part of myd

25.6 compare module

compare.**compare**(*data1*, *data2*, *rtol=1e-12*)

given two CellCenterData2d objects, compare the data, zone-by-zone and output any errors

Parameters

data1, data2 [CellCenterData2d object] Two data grids to compare

rtol [float] relative tolerance to use to compare grids

25.7 compressible package

The pyro compressible hydrodynamics solver. This implements the second-order (piecewise-linear), unsplit method of Colella 1990.

25.7.1 Subpackages

compressible.problems package

Submodules

compressible.problems.acoustic_pulse module

`compressible.problems.acoustic_pulse.`**`finalize`**`()`
print out any information to the user at the end of the run

`compressible.problems.acoustic_pulse.`**`init_data`**`(`*myd*, *rp*`)`
initialize the acoustic_pulse problem. This comes from McCourquodale & Coella 2011

compressible.problems.advect module

`compressible.problems.advect.`**`finalize`**`()`
print out any information to the user at the end of the run

`compressible.problems.advect.`**`init_data`**`(`*my_data*, *rp*`)`
initialize a smooth advection problem for testing convergence

compressible.problems.bubble module

`compressible.problems.bubble.`**`finalize`**`()`
print out any information to the user at the end of the run

`compressible.problems.bubble.`**`init_data`**`(`*my_data*, *rp*`)`
initialize the bubble problem

compressible.problems.hse module

`compressible.problems.hse.`**`finalize`**`()`
print out any information to the user at the end of the run

`compressible.problems.hse.`**`init_data`**`(`*my_data*, *rp*`)`
initialize the HSE problem

compressible.problems.kh module

`compressible.problems.kh.`**`finalize`**`()`
print out any information to the user at the end of the run

`compressible.problems.kh.`**`init_data`**`(`*my_data*, *rp*`)`
initialize the Kelvin-Helmholtz problem

compressible.problems.logo module

`compressible.problems.logo.`**`finalize`**`()`
print out any information to the user at the end of the run

`compressible.problems.logo.`**`init_data`**`(`*my_data*, *rp*`)`
initialize the logo problem

compressible.problems.quad module

`compressible.problems.quad.finalize()`
print out any information to the user at the end of the run

`compressible.problems.quad.init_data(my_data, rp)`
initialize the quadrant problem

compressible.problems.ramp module

`compressible.problems.ramp.finalize()`
print out any information to the user at the end of the run

`compressible.problems.ramp.init_data(my_data, rp)`
initialize the double Mach reflection problem

compressible.problems.rt module

`compressible.problems.rt.finalize()`
print out any information to the user at the end of the run

`compressible.problems.rt.init_data(my_data, rp)`
initialize the rt problem

compressible.problems.rt2 module

A RT problem with two distinct modes: short wave length on the left and long wavelenght on the right. This allows one to see how the growth rate depends on wavenumber.

`compressible.problems.rt2.finalize()`
print out any information to the user at the end of the run

`compressible.problems.rt2.init_data(my_data, rp)`
initialize the rt problem

compressible.problems.sedov module

`compressible.problems.sedov.finalize()`
print out any information to the user at the end of the run

`compressible.problems.sedov.init_data(my_data, rp)`
initialize the sedov problem

compressible.problems.sod module

`compressible.problems.sod.finalize()`
print out any information to the user at the end of the run

`compressible.problems.sod.init_data(my_data, rp)`
initialize the sod problem

compressible.problems.test module

`compressible.problems.test.`**`finalize`**`()`
print out any information to the user at the end of the run

`compressible.problems.test.`**`init_data`**`(`*my_data*, *rp*`)`
an init routine for unit testing

25.7.2 Submodules

25.7.3 compressible.BC module

compressible-specific boundary conditions. Here, in particular, we implement an HSE BC in the vertical direction.

Note: the pyro BC routines operate on a single variable at a time, so some work will necessarily be repeated.

Also note: we may come in here with the aux_data (source terms), so we'll do a special case for them

`compressible.BC.`**`inflow_post_bc`**`(`*var*, *g*`)`

`compressible.BC.`**`inflow_pre_bc`**`(`*var*, *g*`)`

`compressible.BC.`**`user`**`(`*bc_name*, *bc_edge*, *variable*, *ccdata*`)`
A hydrostatic boundary. This integrates the equation of HSE into the ghost cells to get the pressure and density under the assumption that the specific internal energy is constant.

Upon exit, the ghost cells for the input variable will be set

Parameters

bc_name [{'hse'}] The descriptive name for the boundary condition – this allows for pyro to have multiple types of user-supplied boundary conditions. For this module, it needs to be 'hse'.

bc_edge [{'ylb', 'yrb'}] The boundary to update: ylb = lower y boundary; yrb = upper y boundary.

variable [{'density', 'x-momentum', 'y-momentum', 'energy'}] The variable whose ghost cells we are filling

ccdata [CellCenterData2d object] The data object

25.7.4 compressible.derives module

`compressible.derives.`**`derive_primitives`**`(`*myd*, *varnames*`)`
derive desired primitive variables from conserved state

25.7.5 compressible.eos module

This is a gamma-law equation of state: p = rho e (gamma - 1), where gamma is the constant ratio of specific heats.

`compressible.eos.`**`dens`**`(`*gamma*, *pres*, *eint*`)`
Given the pressure and the specific internal energy, return the density

Parameters

gamma [float] The ratio of specific heats

pres [float] The pressure

eint [float] The specific internal energy

Returns

out [float] The density

`compressible.eos.`**`pres`**(*gamma*, *dens*, *eint*)

Given the density and the specific internal energy, return the pressure

Parameters

gamma [float] The ratio of specific heats

dens [float] The density

eint [float] The specific internal energy

Returns

out [float] The pressure

`compressible.eos.`**`rhoe`**(*gamma*, *pres*)

Given the pressure, return (rho * e)

Parameters

gamma [float] The ratio of specific heats

pres [float] The pressure

Returns

out [float] The internal energy density, rho e

25.7.6 compressible.interface module

`compressible.interface.`**`artificial_viscosity`**

Compute the artifical viscosity. Here, we compute edge-centered approximations to the divergence of the velocity. This follows directly Colella Woodward (1984) Eq. 4.5

data locations:

```
j+3/2--+---------+---------+---------+
       |         |         |         |
  j+1  +         |         |         |
       |         |         |         |
j+1/2--+---------+---------+---------+
       |         |         |         |
     j +         X         |         |
       |         |         |         |
j-1/2--+---------+----Y----+---------+
       |         |         |         |
   j-1 +         |         |         |
       |         |         |         |
j-3/2--+---------+---------+---------+
       |    |    |    |    |    |    |
           i-1        i        i+1
     i-3/2     i-1/2     i+1/2     i+3/2
```

`X` is the location of `avisco_x[i,j]` `Y` is the location of `avisco_y[i,j]`

Parameters

ng [int] The number of ghost cells

dx, dy [float] Cell spacings

cvisc [float] viscosity parameter

u, v [ndarray] x- and y-velocities

Returns

out [ndarray, ndarray] Artificial viscosity in the x- and y-directions

`compressible.interface.`**`consFlux`**

Calculate the conservative flux.

Parameters

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

gamma [float] Adiabatic index

idens, ixmom, iymom, iener, irhoX [int] The indices of the density, x-momentum, y-momentum, internal energy density and species partial densities in the conserved state vector.

nspec [int] The number of species

U_state [ndarray] Conserved state vector.

Returns

out [ndarray] Conserved flux

`compressible.interface.`**`riemann_cgf`**

Solve riemann shock tube problem for a general equation of state using the method of Colella, Glaz, and Ferguson. See Almgren et al. 2010 (the CASTRO paper) for details.

The Riemann problem for the Euler's equation produces 4 regions, separated by the three characteristics (u - cs, u, u + cs):

```
u - cs    t    u      u + cs
  \       ^   .       /
   \  *L  |   . *R   /
    \     |  .     /
     \    |  .    /
 L    \   | .   /    R
       \  | .  /
        \ |. /
         \|./
----------+----------------> x
```

We care about the solution on the axis. The basic idea is to use estimates of the wave speeds to figure out which region we are in, and: use jump conditions to evaluate the state there.

Only density jumps across the u characteristic. All primitive variables jump across the other two. Special attention is needed if a rarefaction spans the axis.

Parameters

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

ng [int] The number of ghost cells

nspec [int] The number of species

idens, ixmom, iymom, iener, irhoX [int] The indices of the density, x-momentum, y-momentum, internal energy density and species partial densities in the conserved state vector.

lower_solid, upper_solid [int] Are we at lower or upper solid boundaries?

gamma [float] Adiabatic index

U_l, U_r [ndarray] Conserved state on the left and right cell edges.

Returns

out [ndarray] Conserved flux

`compressible.interface.`**`riemann_hllc`**

This is the HLLC Riemann solver. The implementation follows directly out of Toro's book. Note: this does not handle the transonic rarefaction.

Parameters

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

ng [int] The number of ghost cells

nspec [int] The number of species

idens, ixmom, iymom, iener, irhoX [int] The indices of the density, x-momentum, y-momentum, internal energy density and species partial densities in the conserved state vector.

lower_solid, upper_solid [int] Are we at lower or upper solid boundaries?

gamma [float] Adiabatic index

U_l, U_r [ndarray] Conserved state on the left and right cell edges.

Returns

out [ndarray] Conserved flux

`compressible.interface.`**`riemann_prim`**

this is like riemann_cgf, except that it works on a primitive variable input state and returns the primitive variable interface state

Solve riemann shock tube problem for a general equation of state using the method of Colella, Glaz, and Ferguson. See Almgren et al. 2010 (the CASTRO paper) for details.

The Riemann problem for the Euler's equation produces 4 regions, separated by the three characteristics $(u - cs, u, u + cs)$:

```
u - cs    t    u      u + cs
  \       ^   .       /
   \  *L  |   . *R   /
    \     |  .     /
     \    |  .    /
 L    \   | .   /    R
       \  | .  /
        \ |. /
         \|./
----------+----------------> x
```

We care about the solution on the axis. The basic idea is to use estimates of the wave speeds to figure out which region we are in, and: use jump conditions to evaluate the state there.

Only density jumps across the u characteristic. All primitive variables jump across the other two. Special attention is needed if a rarefaction spans the axis.

Parameters

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

ng [int] The number of ghost cells

nspec [int] The number of species

irho, iu, iv, ip, iX [int] The indices of the density, x-velocity, y-velocity, pressure and species fractions in the state vector.

lower_solid, upper_solid [int] Are we at lower or upper solid boundaries?

gamma [float] Adiabatic index

q_l, q_r [ndarray] Primitive state on the left and right cell edges.

Returns

out [ndarray] Primitive flux

`compressible.interface.`**`states`**

predict the cell-centered state to the edges in one-dimension using the reconstructed, limited slopes.

We follow the convection here that `V_l[i]` is the left state at the i-1/2 interface and `V_l[i+1]` is the left state at the i+1/2 interface.

We need the left and right eigenvectors and the eigenvalues for the system projected along the x-direction.

Taking our state vector as $Q = (\rho, u, v, p)^T$, the eigenvalues are $u - c$, u, $u + c$.

We look at the equations of hydrodynamics in a split fashion – i.e., we only consider one dimension at a time.

Considering advection in the x-direction, the Jacobian matrix for the primitive variable formulation of the Euler equations projected in the x-direction is:

```
    / u   r   0   0 \
    | 0   u   0  1/r |
A = | 0   0   u   0  |
    \ 0  rc^2 0   u  /
```

The right eigenvectors are:

```
     /  1  \         / 1 \        / 0 \        /  1  \
     |-c/r |         | 0 |        | 0 |        | c/r |
r1 = |  0  |   r2 =  | 0 |   r3 = | 1 |   r4 = |  0  |
     \ c^2 /         \ 0 /        \ 0 /        \ c^2 /
```

In particular, we see from r3 that the transverse velocity (v in this case) is simply advected at a speed u in the x-direction.

The left eigenvectors are:

```
l1 =     ( 0,   -r/(2c),   0, 1/(2c^2) )
l2 =     ( 1,      0,      0,  -1/c^2  )
l3 =     ( 0,      0,      1,     0    )
l4 =     ( 0,    r/(2c),   0, 1/(2c^2) )
```

The fluxes are going to be defined on the left edge of the computational zones:

```
 |             |             |             |
 |             |             |             |
-+------+------+------+------+------+------+--
 |     i-1     |      i      |     i+1     |
                         ^  ^          ^
          q_l,i q_r,i  q_l,i+1
```

q_r,i and q_l,i+1 are computed using the information in zone i,j.

Parameters

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

ng [int] The number of ghost cells

dx [float] The cell spacing

dt [float] The timestep

irho, iu, iv, ip, ix [int] Indices of the density, x-velocity, y-velocity, pressure and species in the state vector

nspec [int] The number of species

gamma [float] Adiabatic index

qv [ndarray] The primitive state vector

dqv [ndarray] Spatial derivitive of the state vector

Returns

out [ndarray, ndarray] State vector predicted to the left and right edges

25.7.7 compressible.simulation module

class compressible.simulation.**Simulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: *simulation_null.NullSimulation*

The main simulation class for the corner transport upwind compressible hydrodynamics solver

dovis()

Do runtime visualization.

evolve()

Evolve the equations of compressible hydrodynamics through a timestep dt.

initialize(*extra_vars=None*, *ng=4*)

Initialize the grid and variables for compressible flow and set the initial conditions for the chosen problem.

method_compute_timestep()

The timestep function computes the advective timestep (CFL) constraint. The CFL constraint says that information cannot propagate further than one zone per timestep.

We use the driver.cfl parameter to control what fraction of the CFL step we actually take.

write_extras(*f*)

Output simulation-specific data to the h5py file f

class compressible.simulation.**Variables**(*myd*)

Bases: object

a container class for easy access to the different compressible variable by an integer key

compressible.simulation.**cons_to_prim**(*U*, *gamma*, *ivars*, *myg*)

convert an input vector of conserved variables to primitive variables

compressible.simulation.**prim_to_cons**(*q*, *gamma*, *ivars*, *myg*)

convert an input vector of primitive variables to conserved variables

25.7.8 compressible.unsplit_fluxes module

Implementation of the Colella 2nd order unsplit Godunov scheme. This is a 2-dimensional implementation only. We assume that the grid is uniform, but it is relatively straightforward to relax this assumption.

There are several different options for this solver (they are all discussed in the Colella paper).

- limiter: 0 = no limiting; 1 = 2nd order MC limiter; 2 = 4th order MC limiter
- riemann: HLLC or CGF (for Colella, Glaz, and Freguson solver)
- use_flattening: set to 1 to use the multidimensional flattening at shocks
- delta, z0, z1: flattening parameters (we use Colella 1990 defaults)

The grid indices look like:

```
j+3/2--+---------+---------+---------+
       |         |         |         |
  j+1 _|         |         |         |
       |         |         |         |
       |         |         |         |
j+1/2--+---------XXXXXXXXXXX---------+
       |         X         X         |
    j _|         X         X         |
       |         X         X         |
       |         X         X         |
j-1/2--+---------XXXXXXXXXXX---------+
       |         |         |         |
  j-1 _|         |         |         |
       |         |         |         |
       |         |         |         |
j-3/2--+---------+---------+---------+
       |    |    |    |    |    |    |
          i-1        i        i+1
     i-3/2     i-1/2     i+1/2     i+3/2
```

We wish to solve

$$U_t + F^x_x + F^y_y = H$$

we want U_{i+1/2}^{n+1/2} – the interface values that are input to the Riemann problem through the faces for each zone.

Taylor expanding yields:

```
 n+1/2                        dU            dU
U           = U   + 0.5 dx  --   + 0.5 dt --
 i+1/2,j,L     i,j            dx            dt
```

(continues on next page)

(continued from previous page)

```
                    dU              dF^x    dF^y
   = U    + 0.5 dx  --   - 0.5 dt  ( ---- + ---- - H )
      i,j           dx                dx      dy


                     dU      dF^x             dF^y
   = U    + 0.5  ( dx -- - dt ---- ) - 0.5 dt ---- + 0.5 dt H
      i,j            dx       dx               dy


                         dt       dU            dF^y
   = U    + 0.5 dx ( 1 - -- A^x )  --   - 0.5 dt ---- + 0.5 dt H
      i,j                dx       dx              dy


                       dt          _             dF^y
   = U    + 0.5   ( 1 - -- A^x ) DU  - 0.5 dt ---- + 0.5 dt H
      i,j               dx                        dy

            +----------+-----------+  +----+----+   +---+---+
                       |                   |             |

           this is the monotonized   this is the   source term
           central difference term   transverse
                                     flux term
```

There are two components, the central difference in the normal to the interface, and the transverse flux difference. This is done for the left and right sides of all 4 interfaces in a zone, which are then used as input to the Riemann problem, yielding the 1/2 time interface values:

```
 n+1/2
U
 i+1/2,j
```

Then, the zone average values are updated in the usual finite-volume way:

```
 n+1    n      dt     x   n+1/2         x   n+1/2
U     = U    + -- { F (U         ) - F (U         ) }
 i,j    i,j   dx         i-1/2,j           i+1/2,j


               dt     y   n+1/2         y   n+1/2
             + -- { F (U         ) - F (U         ) }
               dy         i,j-1/2           i,j+1/2
```

Updating U_{i,j}:

- We want to find the state to the left and right (or top and bottom) of each interface, ex. U_{i+1/2,j,[lr]}^{n+1/2}, and use them to solve a Riemann problem across each of the four interfaces.
- U_{i+1/2,j,[lr]}^{n+1/2} is comprised of two parts, the computation of the monotonized central differences in the normal direction (eqs. 2.8, 2.10) and the computation of the transverse derivatives, which requires the solution of a Riemann problem in the transverse direction (eqs. 2.9, 2.14).
  - the monotonized central difference part is computed using the primitive variables.

– We compute the central difference part in both directions before doing the transverse flux differencing, since for the high-order transverse flux implementation, we use these as the input to the transverse Riemann problem.

`compressible.unsplit_fluxes.unsplit_fluxes`(*my_data*, *my_aux*, *rp*, *ivars*, *solid*, *tc*, *dt*)
unsplitFluxes returns the fluxes through the x and y interfaces by doing an unsplit reconstruction of the interface values and then solving the Riemann problem through all the interfaces at once

currently we assume a gamma-law EOS

The runtime parameter grav is assumed to be the gravitational acceleration in the y-direction

Parameters

my_data [CellCenterData2d object] The data object containing the grid and advective scalar that we are advecting.

rp [RuntimeParameters object] The runtime parameters for the simulation

vars [Variables object] The Variables object that tells us which indices refer to which variables

tc [TimerCollection object] The timers we are using to profile

dt [float] The timestep we are advancing through.

Returns

out [ndarray, ndarray] The fluxes on the x- and y-interfaces

25.8 compressible_fv4 package

This is a 4th order accurate compressible hydrodynamics solver, implementing the algorithm from McCorquodale & Colella (2011).

25.8.1 Subpackages

compressible_fv4.problems package

Submodules

compressible_fv4.problems.acoustic_pulse module

`compressible_fv4.problems.acoustic_pulse.finalize`()
print out any information to the user at the end of the run

`compressible_fv4.problems.acoustic_pulse.init_data`(*myd*, *rp*)
initialize the acoustic_pulse problem. This comes from McCourquodale & Coella 2011

25.8.2 Submodules

25.8.3 compressible_fv4.fluxes module

`compressible_fv4.fluxes.flux_cons`(*ivars*, *idir*, *gamma*, *q*)

`compressible_fv4.fluxes.fluxes`(*myd*, *rp*, *ivars*, *solid*, *tc*)

25.8.4 compressible_fv4.simulation module

class compressible_fv4.simulation.**Simulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.fv.FV2d'>*)

Bases: *compressible_rk.simulation.Simulation*

evolve()
Evolve the equations of compressible hydrodynamics through a timestep dt.

initialize(*ng=5*)
Initialize the grid and variables for compressible flow and set the initial conditions for the chosen problem.

preevolve()
Since we are 4th order accurate we need to make sure that we initialized with accurate zone-averages, so the preevolve for this solver assumes that the initialization was done to cell-centers and converts it to cell-averages.

substep(*myd*)
compute the advective source term for the given state

25.9 compressible_react package

The pyro compressible hydrodynamics solver with reactions. This implements the second-order (piecewise-linear), unsplit method of Colella 1990, and incorporates reactions via Strang splitting.

25.9.1 Subpackages

compressible_react.problems package

Submodules

compressible_react.problems.flame module

compressible_react.problems.flame.**finalize**()
print out any information to the user at the end of the run

compressible_react.problems.flame.**init_data**(*my_data*, *rp*)
initialize the sedov problem

compressible_react.problems.rt module

compressible_react.problems.rt.**finalize**()
print out any information to the user at the end of the run

compressible_react.problems.rt.**init_data**(*my_data*, *rp*)
initialize the rt problem

25.9.2 Submodules

25.9.3 compressible_react.simulation module

class compressible_react.simulation.**Simulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: *compressible.simulation.Simulation*

burn(*dt*)
react fuel to ash

diffuse(*dt*)
diffuse for dt

dovis()
Do runtime visualization.

evolve()
Evolve the equations of compressible hydrodynamics through a timestep dt.

initialize()
For the reacting compressible solver, our initialization of the data is the same as the compressible solver, but we supply additional variables.

25.10 compressible_rk package

A method-of-lines compressible hydrodynamics solver. Piecewise constant reconstruction is done in space and a Runge-Kutta time integration is used to advance the solutiion.

25.10.1 Subpackages

25.10.2 Submodules

25.10.3 compressible_rk.fluxes module

This is a 2nd-order PLM method for a method-of-lines integration (i.e., no characteristic tracing).

We wish to solve

$$U_t + F^x_x + F^y_y = H$$

we want U_{i+1/2} – the interface values that are input to the Riemann problem through the faces for each zone.

Taylor expanding *in space only* yields:

```
                         dU
U          = U   + 0.5 dx  --
 i+1/2,j,L    i,j            dx
```

compressible_rk.fluxes.**fluxes**(*my_data*, *rp*, *ivars*, *solid*, *tc*)
unsplitFluxes returns the fluxes through the x and y interfaces by doing an unsplit reconstruction of the interface values and then solving the Riemann problem through all the interfaces at once

currently we assume a gamma-law EOS

Parameters

my_data [CellCenterData2d object] The data object containing the grid and advective scalar that we are advecting.

rp [RuntimeParameters object] The runtime parameters for the simulation

vars [Variables object] The Variables object that tells us which indices refer to which variables

tc [TimerCollection object] The timers we are using to profile

Returns

out [ndarray, ndarray] The fluxes on the x- and y-interfaces

25.10.4 compressible_rk.simulation module

class compressible_rk.simulation.**Simulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: `compressible.simulation.Simulation`

The main simulation class for the method of lines compressible hydrodynamics solver

evolve()

Evolve the equations of compressible hydrodynamics through a timestep dt.

method_compute_timestep()

The timestep function computes the advective timestep (CFL) constraint. The CFL constraint says that information cannot propagate further than one zone per timestep.

We use the driver.cfl parameter to control what fraction of the CFL step we actually take.

substep(*myd*)

take a single substep in the RK timestepping starting with the conservative state defined as part of myd

25.11 compressible_sdc package

This is a 4th order accurate compressible hydrodynamics solver, implementing the spatial reconstruction from McCorquodale & Colella (2011) but using an SDC scheme for the time integration.

25.11.1 Subpackages

25.11.2 Submodules

25.11.3 compressible_sdc.simulation module

The routines that implement the 4th-order compressible scheme, using SDC time integration

class compressible_sdc.simulation.**Simulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.fv.FV2d'>*)

Bases: `compressible_fv4.simulation.Simulation`

Drive the 4th-order compressible solver with SDC time integration

evolve()

Evolve the equations of compressible hydrodynamics through a timestep dt.

sdc_integral (*m_start*, *m_end*, *As*)
Compute the integral over the sources from m to m+1 with a Simpson's rule

25.12 diffusion package

The pyro diffusion solver. This implements second-order implicit diffusion using Crank-Nicolson time-differencing. The resulting system is solved using multigrid.

The general flow is:

- compute the RHS given the current state
- set up the MG
- solve the system using MG for updated phi

The timestep is computed as:

```
CFL* 0.5*dt/dx**2
```

25.12.1 Subpackages

diffusion.problems package

Submodules

diffusion.problems.gaussian module

diffusion.problems.gaussian.**finalize**()
print out any information to the user at the end of the run

diffusion.problems.gaussian.**init_data**(*my_data*, *rp*)
initialize the Gaussian diffusion problem

diffusion.problems.gaussian.**phi_analytic**(*dist*, *t*, *t_0*, *k*, *phi_1*, *phi_2*)
the analytic solution to the Gaussian diffusion problem

diffusion.problems.test module

diffusion.problems.test.**finalize**()
print out any information to the user at the end of the run

diffusion.problems.test.**init_data**(*my_data*, *rp*)
an init routine for unit testing

25.12.2 Submodules

25.12.3 diffusion.simulation module

A simulation of diffusion

class diffusion.simulation.**Simulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: *simulation_null.NullSimulation*

A simulation of diffusion

dovis()

Do runtime visualization.

evolve()

Diffusion through dt using C-N implicit solve with multigrid

initialize()

Initialize the grid and variables for diffusion and set the initial conditions for the chosen problem.

method_compute_timestep()

The diffusion timestep() function computes the timestep using the explicit timestep constraint as the starting point. We then multiply by the CFL number to get the timestep. Since we are doing an implicit discretization, we do not require CFL < 1.

25.13 examples package

25.13.1 Subpackages

examples.multigrid package

Submodules

examples.multigrid.mg_test_general_alphabeta_only module

Test the general MG solver with a variable coeffcient Helmholtz problem. This ensures we didn't screw up the base functionality here.

Here we solve:

```
alpha phi + div . ( beta grad phi ) = f
```

with:

```
alpha = 1.0
beta = 2.0 + cos(2.0*pi*x)*cos(2.0*pi*y)

f = (-16.0*pi**2*cos(2*pi*x)*cos(2*pi*y) - 16.0*pi**2 + 1.0)*sin(2*pi*x)*sin(2*pi*y)
```

This has the exact solution:

```
phi = sin(2.0*pi*x)*sin(2.0*pi*y)
```

on [0,1] x [0,1]

We use Dirichlet BCs on phi. For beta, we do not have to impose the same BCs, since that may represent a different physical quantity. Here we take beta to have Neumann BCs. (Dirichlet BCs for beta will force it to 0 on the boundary, which is not correct here)

examples.multigrid.mg_test_general_alphabeta_only.**alpha**(*x*, *y*) examples.multigrid.mg_test_general_alphabeta_only.**beta**(*x*, *y*)

examples.multigrid.mg_test_general_alphabeta_only.**f**(*x*, *y*)

examples.multigrid.mg_test_general_alphabeta_only.**gamma_x**(*x*, *y*)

examples.multigrid.mg_test_general_alphabeta_only.**gamma_y**(*x*, *y*)

examples.multigrid.mg_test_general_alphabeta_only.**test_general_poisson_dirichlet**(*N*, *store_bench=*, *comp_bench=*, *make_plot=Fa*, *verbose=1*, *rtol=1e-12*)

test the general MG solver. The return value here is the error compared to the exact solution, UNLESS comp_bench=True, in which case the return value is the error compared to the stored benchmark

examples.multigrid.mg_test_general_alphabeta_only.**true**(*x*, *y*)

examples.multigrid.mg_test_general_beta_only module

Test the general MG solver with a variable coeffcient Poisson problem (in essence, we are making this solver act like the variable_coefficient_MG.py solver). This ensures we didn't screw up the base functionality here.

Here we solve:

```
div . ( beta grad phi ) = f
```

with:

```
beta = 2.0 + cos(2.0*pi*x)*cos(2.0*pi*y)

f = -16.0*pi**2*(cos(2*pi*x)*cos(2*pi*y) + 1)*sin(2*pi*x)*sin(2*pi*y)
```

This has the exact solution:

```
phi = sin(2.0*pi*x)*sin(2.0*pi*y)
```

on [0,1] x [0,1]

We use Dirichlet BCs on phi. For beta, we do not have to impose the same BCs, since that may represent a different physical quantity. Here we take beta to have Neumann BCs. (Dirichlet BCs for beta will force it to 0 on the boundary, which is not correct here)

examples.multigrid.mg_test_general_beta_only.**alpha**(*x*, *y*)

examples.multigrid.mg_test_general_beta_only.**beta**(*x*, *y*)

examples.multigrid.mg_test_general_beta_only.**f**(*x*, *y*)

examples.multigrid.mg_test_general_beta_only.**gamma_x**(*x*, *y*)

examples.multigrid.mg_test_general_beta_only.**gamma_y**(*x*, *y*) examples.multigrid.mg_test_general_beta_only.**test_general_poisson_dirichlet**(*N*, *store_bench=False*, *comp_bench=False*, *make_plot=False*, *verbose=1*, *rtol=1e-12*)

test the general MG solver. The return value here is the error compared to the exact solution, UNLESS comp_bench=True, in which case the return value is the error compared to the stored benchmark

examples.multigrid.mg_test_general_beta_only.**true**(*x*, *y*)

examples.multigrid.mg_test_general_constant module

Test the general MG solver with a CONSTANT coefficient problem – the same one from the multigrid class test. This ensures we didn't screw up the base functionality here.

We solve:

```
u_xx + u_yy = -2[(1-6x**2)y**2(1-y**2) + (1-6y**2)x**2(1-x**2)]
u = 0 on the boundary
```

this is the example from page 64 of the book *A Multigrid Tutorial, 2nd Ed.*

The analytic solution is u(x,y) = (x**2 - x**4)(y**4 - y**2)

examples.multigrid.mg_test_general_constant.**alpha**(*x*, *y*)

examples.multigrid.mg_test_general_constant.**beta**(*x*, *y*)

examples.multigrid.mg_test_general_constant.**f**(*x*, *y*)

examples.multigrid.mg_test_general_constant.**gamma_x**(*x*, *y*)

examples.multigrid.mg_test_general_constant.**gamma_y**(*x*, *y*)

examples.multigrid.mg_test_general_constant.**test_general_poisson_dirichlet**(*N*, *store_bench=False*, *comp_bench=False*, *make_plot=False*, *verbose=1*, *rtol=1e-12*)

test the general MG solver. The return value here is the error compared to the exact solution, UNLESS comp_bench=True, in which case the return value is the error compared to the stored benchmark

examples.multigrid.mg_test_general_constant.**true**(*x*, *y*)

examples.multigrid.mg_test_general_dirichlet module

Test the general MG solver with Dirichlet boundary conditions.

Here we solve:

```
alpha phi + div{beta grad phi} + gamma . grad phi = f
```

with:

```
alpha = 1.0
beta = cos(2*pi*x)*cos(2*pi*y) + 2.0
gamma_x = sin(2*pi*x)
gamma_y = sin(2*pi*y)

f = (-16.0*pi**2*cos(2*pi*x)*cos(2*pi*y) + 2.0*pi*cos(2*pi*x) +
      2.0*pi*cos(2*pi*y) - 16.0*pi**2 + 1.0)*sin(2*pi*x)*sin(2*pi*y)
```

This has the exact solution:

```
phi = sin(2.0*pi*x)*sin(2.0*pi*y)
```

on [0,1] x [0,1]

We use Dirichlet BCs on phi.

For the coefficients we do not have to impose the same BCs, since that may represent a different physical quantity. beta is the one that really matters since it must be brought to the edges. Here we take beta to have Neumann BCs. (Dirichlet BCs for beta will force it to 0 on the boundary, which is not correct here)

`examples.multigrid.mg_test_general_dirichlet.`**`alpha`**(*x*, *y*)

`examples.multigrid.mg_test_general_dirichlet.`**`beta`**(*x*, *y*)

`examples.multigrid.mg_test_general_dirichlet.`**`f`**(*x*, *y*)

`examples.multigrid.mg_test_general_dirichlet.`**`gamma_x`**(*x*, *y*)

`examples.multigrid.mg_test_general_dirichlet.`**`gamma_y`**(*x*, *y*)

`examples.multigrid.mg_test_general_dirichlet.`**`test_general_poisson_dirichlet`**(*N*, *store_bench=False*, *comp_bench=False*, *make_plot=False*, *verbose=1*, *rtol=1e-12*)

test the general MG solver. The return value here is the error compared to the exact solution, UNLESS comp_bench=True, in which case the return value is the error compared to the stored benchmark

`examples.multigrid.mg_test_general_dirichlet.`**`true`**(*x*, *y*)

examples.multigrid.mg_test_general_inhomogeneous module

Test the general MG solver with inhomogeneous Dirichlet boundary conditions.

Here we solve:

```
alpha phi + div{beta grad phi} + gamma . grad phi = f
```

with:

```
alpha = 10.0
beta = x*y + 1  (note: x*y alone doesn't work)
gamma_x = 1
gamma_y = 1

f =  -(pi/2)*(x + 1)*sin(pi*y/2)*cos(pi*x/2)
```

(continues on next page)

(continued from previous page)

```
    - (pi/2)*(y + 1)*sin(pi*x/2)*cos(pi*y/2) +
    (-pi**2*(x*y+1)/2 + 10)*cos(pi*x/2)*cos(pi*y/2)
```

This has the exact solution:

```
phi = cos(pi*x/2)*cos(pi*y/2)
```

on [0,1] x [0,1], with Dirichlet boundary conditions:

```
phi(x=0) = cos(pi*y/2)
phi(x=1) = 0
phi(y=0) = cos(pi*x/2)
phi(y=1) = 0
```

For the coefficients we do not have to impose the same BCs, since that may represent a different physical quantity. beta is the one that really matters since it must be brought to the edges. Here we take beta to have Neumann BCs. (Dirichlet BCs for beta will force it to 0 on the boundary, which is not correct here)

`examples.multigrid.mg_test_general_inhomogeneous.`**`alpha`**(*x*, *y*)

`examples.multigrid.mg_test_general_inhomogeneous.`**`beta`**(*x*, *y*)

`examples.multigrid.mg_test_general_inhomogeneous.`**`f`**(*x*, *y*)

`examples.multigrid.mg_test_general_inhomogeneous.`**`gamma_x`**(*x*, *y*)

`examples.multigrid.mg_test_general_inhomogeneous.`**`gamma_y`**(*x*, *y*)

`examples.multigrid.mg_test_general_inhomogeneous.`**`test_general_poisson_inhomogeneous`**(*N*, *store_ben* *comp_ber* *make_plo* *verbose=1*, *rtol=1e-12*)

test the general MG solver. The return value here is the error compared to the exact solution, UNLESS comp_bench=True, in which case the return value is the error compared to the stored benchmark

`examples.multigrid.mg_test_general_inhomogeneous.`**`true`**(*x*, *y*)

`examples.multigrid.mg_test_general_inhomogeneous.`**`xl_func`**(*y*)

`examples.multigrid.mg_test_general_inhomogeneous.`**`yl_func`**(*x*)

examples.multigrid.mg_test_simple module

an example of using the multigrid class to solve Laplace's equation. Here, we solve:

```
u_xx + u_yy = -2[(1-6x**2)y**2(1-y**2) + (1-6y**2)x**2(1-x**2)]
u = 0 on the boundary
```

this is the example from page 64 of the book *A Multigrid Tutorial, 2nd Ed.*

The analytic solution is u(x,y) = (x**2 - x**4)(y**4 - y**2)

`examples.multigrid.mg_test_simple.`**`f`**(*x*, *y*) examples.multigrid.mg_test_simple.**test_poisson_dirichlet**(*N*, *store_bench=False*, *comp_bench=False*, *make_plot=False*, *verbose=1*, *rtol=1e-12*)

examples.multigrid.mg_test_simple.**true**(*x*, *y*)

examples.multigrid.mg_test_vc_constant module

Test the variable coefficient MG solver with a CONSTANT coefficient problem – the same one from the multigrid class test. This ensures we didn't screw up the base functionality here.

We solve:

```
u_xx + u_yy = -2[(1-6x**2)y**2(1-y**2) + (1-6y**2)x**2(1-x**2)]
u = 0 on the boundary
```

this is the example from page 64 of the book *A Multigrid Tutorial, 2nd Ed.*

The analytic solution is u(x,y) = (x**2 - x**4)(y**4 - y**2)

examples.multigrid.mg_test_vc_constant.**alpha**(*x*, *y*)

examples.multigrid.mg_test_vc_constant.**f**(*x*, *y*)

examples.multigrid.mg_test_vc_constant.**test_vc_constant**(*N*)

examples.multigrid.mg_test_vc_constant.**true**(*x*, *y*)

examples.multigrid.mg_test_vc_dirichlet module

Test the variable-coefficient MG solver with Dirichlet boundary conditions.

Here we solve:

```
div . ( alpha grad phi ) = f
```

with:

```
alpha = 2.0 + cos(2.0*pi*x)*cos(2.0*pi*y)

f = -16.0*pi**2*(cos(2*pi*x)*cos(2*pi*y) + 1)*sin(2*pi*x)*sin(2*pi*y)
```

This has the exact solution:

```
phi = sin(2.0*pi*x)*sin(2.0*pi*y)
```

on [0,1] x [0,1]

We use Dirichlet BCs on phi. For alpha, we do not have to impose the same BCs, since that may represent a different physical quantity. Here we take alpha to have Neumann BCs. (Dirichlet BCs for alpha will force it to 0 on the boundary, which is not correct here)

examples.multigrid.mg_test_vc_dirichlet.**alpha**(*x*, *y*)

examples.multigrid.mg_test_vc_dirichlet.**f**(*x*, *y*) `examples.multigrid.mg_test_vc_dirichlet.`**`test_vc_poisson_dirichlet`**`(`*N, store_bench=False, comp_bench=False, make_plot=False, verbose=1, rtol=1e-12*`)`

test the variable-coefficient MG solver. The return value here is the error compared to the exact solution, UNLESS comp_bench=True, in which case the return value is the error compared to the stored benchmark

`examples.multigrid.mg_test_vc_dirichlet.`**`true`**`(`*x, y*`)`

examples.multigrid.mg_test_vc_periodic module

Test the variable-coefficient MG solver with periodic data.

Here we solve:

```
div . ( alpha grad phi ) = f
```

with:

```
alpha = 2.0 + cos(2.0*pi*x)*cos(2.0*pi*y)

f = -16.0*pi**2*(cos(2*pi*x)*cos(2*pi*y) + 1)*sin(2*pi*x)*sin(2*pi*y)
```

This has the exact solution:

```
phi = sin(2.0*pi*x)*sin(2.0*pi*y)
```

on [0,1] x [0,1]

We use Dirichlet BCs on phi. For alpha, we do not have to impose the same BCs, since that may represent a different physical quantity. Here we take alpha to have Neumann BCs. (Dirichlet BCs for alpha will force it to 0 on the boundary, which is not correct here)

`examples.multigrid.mg_test_vc_periodic.`**`alpha`**`(`*x, y*`)`

`examples.multigrid.mg_test_vc_periodic.`**`f`**`(`*x, y*`)`

`examples.multigrid.mg_test_vc_periodic.`**`test_vc_poisson_periodic`**`(`*N, store_bench=False, comp_bench=False, make_plot=False, verbose=1, rtol=1e-12*`)`

test the variable-coefficient MG solver. The return value here is the error compared to the exact solution, UNLESS comp_bench=True, in which case the return value is the error compared to the stored benchmark

`examples.multigrid.mg_test_vc_periodic.`**`true`**`(`*x, y*`)`

examples.multigrid.mg_vis module

an example of using the multigrid class to solve Laplace's equation. Here, we solve:

```
u_xx + u_yy = -2[(1-6x**2)y**2(1-y**2) + (1-6y**2)x**2(1-x**2)]
u = 0 on the boundary
```

this is the example from page 64 of the book *A Multigrid Tutorial, 2nd Ed.*

The analytic solution is u(x,y) = (x**2 - x**4)(y**4 - y**2)

`examples.multigrid.mg_vis.`**`doit`**`(`*nx*, *ny*`)`

`examples.multigrid.mg_vis.`**`f`**`(`*x*, *y*`)`

`examples.multigrid.mg_vis.`**`true`**`(`*x*, *y*`)`

examples.multigrid.project_periodic module

test of a cell-centered, centered-difference approximate projection.

initialize the velocity field to be divergence free and then add to it the gradient of a scalar (whose normal component vanishes on the boundaries). The projection should recover the original divergence- free velocity field.

The test velocity field comes from Almgen, Bell, and Szymczak 1996.

This makes use of the multigrid solver with periodic boundary conditions.

One of the things that this test demonstrates is that the initial projection may not be able to completely remove the divergence free part, so subsequent projections may be necessary. In this example, we add a very strong gradient component.

The total number of projections to perform is given by nproj. Each projection uses the divergence of the velocity field from the previous iteration as its source term.

Note: the output file created stores the original field, the poluted field, and the recovered field.

`examples.multigrid.project_periodic.`**`doit`**`(`*nx*, *ny*`)`
manage the entire projection

examples.multigrid.prolong_restrict_demo module

`examples.multigrid.prolong_restrict_demo.`**`doit`**`()`

25.14 incompressible package

The pyro solver for incompressible flow. This implements as second-order approximate projection method. The general flow is:

- create the limited slopes of u and v (in both directions)
- get the advective velocities through a piecewise linear Godunov method
- enforce the divergence constraint on the velocities through a projection (the MAC projection)
- recompute the interface states using the new advective velocity
- update U in time to get the provisional velocity field
- project the final velocity to enforce the divergence constraint.

The projections are done using multigrid

25.14.1 Subpackages

incompressible.problems package

Submodules

incompressible.problems.converge module

Initialize a smooth incompressible convergence test. Here, the velocities are initialized as

$$u(x, y) = 1 - 2\cos(2\pi x)\sin(2\pi y)$$
$$v(x, y) = 1 + 2\sin(2\pi x)\cos(2\pi y)$$

and the exact solution at some later time t is then

$$u(x, y, t) = 1 - 2\cos(2\pi(x - t))\sin(2\pi(y - t))$$
$$v(x, y, t) = 1 + 2\sin(2\pi(x - t))\cos(2\pi(y - t))$$
$$p(x, y, t) = -\cos(4\pi(x - t)) - \cos(4\pi(y - t))$$

The numerical solution can be compared to the exact solution to measure the convergence rate of the algorithm.

`incompressible.problems.converge.`**`finalize`**`()`
print out any information to the user at the end of the run

`incompressible.problems.converge.`**`init_data`**`(`*my_data*, *rp*`)`
initialize the incompressible converge problem

incompressible.problems.shear module

Initialize the doubly periodic shear layer (see, for example, Martin and Colella, 2000, JCP, 163, 271). This is run in a unit square domain, with periodic boundary conditions on all sides. Here, the initial velocity is:

```
              / tanh(rho_s (y-0.25))   if y <= 0.5
u(x,y,t=0) = <
              \ tanh(rho_s (0.75-y))   if y > 0.5

v(x,y,t=0) = delta_s sin(2 pi x)
```

`incompressible.problems.shear.`**`finalize`**`()`
print out any information to the user at the end of the run

`incompressible.problems.shear.`**`init_data`**`(`*my_data*, *rp*`)`
initialize the incompressible shear problem

25.14.2 Submodules

25.14.3 incompressible.incomp_interface module

`incompressible.incomp_interface.`**`get_interface_states`**
Compute the unsplit predictions of u and v on both the x- and y-interfaces. This includes the transverse terms.

Parameters

ng [int] The number of ghost cells

dx, dy [float] The cell spacings

dt [float] The timestep

u, v [ndarray] x-velocity and y-velocity

ldelta_ux, ldelta_uy: ndarray Limited slopes of the x-velocity in the x and y directions

ldelta_vx, ldelta_vy: ndarray Limited slopes of the y-velocity in the x and y directions

gradp_x, gradp_y [ndarray] Pressure gradients in the x and y directions

Returns

out [ndarray, ndarray, ndarray, ndarray, ndarray, ndarray, ndarray, ndarray] unsplit predictions of u and v on both the x- and y-interfaces

`incompressible.incomp_interface.`**`mac_vels`**

Calculate the MAC velocities in the x and y directions.

Parameters

ng [int] The number of ghost cells

dx, dy [float] The cell spacings

dt [float] The timestep

u, v [ndarray] x-velocity and y-velocity

ldelta_ux, ldelta_uy: ndarray Limited slopes of the x-velocity in the x and y directions

ldelta_vx, ldelta_vy: ndarray Limited slopes of the y-velocity in the x and y directions

gradp_x, gradp_y [ndarray] Pressure gradients in the x and y directions

Returns

out [ndarray, ndarray] MAC velocities in the x and y directions

`incompressible.incomp_interface.`**`riemann`**

Solve the Burger's Riemann problem given the input left and right states and return the state on the interface.

This uses the expressions from Almgren, Bell, and Szymczak 1996.

Parameters

ng [int] The number of ghost cells

q_l, q_r [ndarray] left and right states

Returns

out [ndarray] Interface state

`incompressible.incomp_interface.`**`riemann_and_upwind`**

First solve the Riemann problem given q_l and q_r to give the velocity on the interface and: use this velocity to upwind to determine the state (q_l, q_r, or a mix) on the interface).

This differs from upwind, above, in that we don't take in a velocity to upwind with).

Parameters

ng [int] The number of ghost cells

q_l, q_r [ndarray] left and right states

Returns

out [ndarray] Upwinded state

`incompressible.incomp_interface.`**`states`**

This is similar to `mac_vels`, but it predicts the interface states of both u and v on both interfaces, using the MAC velocities to do the upwinding.

Parameters

ng [int] The number of ghost cells

dx, dy [float] The cell spacings

dt [float] The timestep

u, v [ndarray] x-velocity and y-velocity

ldelta_ux, ldelta_uy: ndarray Limited slopes of the x-velocity in the x and y directions

ldelta_vx, ldelta_vy: ndarray Limited slopes of the y-velocity in the x and y directions

gradp_x, gradp_y [ndarray] Pressure gradients in the x and y directions

u_MAC, v_MAC [ndarray] MAC velocities in the x and y directions

Returns

out [ndarray, ndarray, ndarray, ndarray] x and y velocities predicted to the interfaces

`incompressible.incomp_interface.`**`upwind`**

upwind the left and right states based on the specified input velocity, s. The resulting interface state is q_int

Parameters

ng [int] The number of ghost cells

q_l, q_r [ndarray] left and right states

s [ndarray] velocity

Returns

out [ndarray] Upwinded state

25.14.4 incompressible.simulation module

`class` `incompressible.simulation.`**`Simulation`**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: *`simulation_null.NullSimulation`*

`dovis`()

Do runtime visualization

`evolve`()

Evolve the incompressible equations through one timestep.

`initialize`()

Initialize the grid and variables for incompressible flow and set the initial conditions for the chosen problem.

`method_compute_timestep`()

The timestep() function computes the advective timestep (CFL) constraint. The CFL constraint says that information cannot propagate further than one zone per timestep.

We use the driver.cfl parameter to control what fraction of the CFL step we actually take.

preevolve()

preevolve is called before we being the timestepping loop. For the incompressible solver, this does an initial projection on the velocity field and then goes through the full evolution to get the value of phi. The fluid state (u, v) is then reset to values before this evolve.

25.15 lm_atm package

The pyro solver for low Mach number atmospheric flow. This implements as second-order approximate projection method. The general flow is:

- create the limited slopes of rho, u and v (in both directions)
- get the advective velocities through a piecewise linear Godunov method
- enforce the divergence constraint on the velocities through a projection (the MAC projection)
- predict rho to edges and do the conservative update
- recompute the interface states using the new advective velocity
- update U in time to get the provisional velocity field
- project the final velocity to enforce the divergence constraint.

The projections are done using multigrid

25.15.1 Subpackages

lm_atm.problems package

Submodules

lm_atm.problems.bubble module

`lm_atm.problems.bubble.`**finalize**()

print out any information to the user at the end of the run

`lm_atm.problems.bubble.`**init_data**(*my_data*, *base*, *rp*)

initialize the bubble problem

25.15.2 Submodules

25.15.3 lm_atm.LM_atm_interface module

`lm_atm.LM_atm_interface.`**get_interface_states**

Compute the unsplit predictions of u and v on both the x- and y-interfaces. This includes the transverse terms.

Note that the `gradp_x`, `gradp_y` should have any coefficients already included (e.g. β_0/ρ)

Parameters

ng [int] The number of ghost cells

dx, dy [float] The cell spacings

dt [float] The timestep

u, v [ndarray] x-velocity and y-velocity

ldelta_ux, ldelta_uy: ndarray Limited slopes of the x-velocity in the x and y directions

ldelta_vx, ldelta_vy: ndarray Limited slopes of the y-velocity in the x and y directions

gradp_x, gradp_y [ndarray] Pressure gradients in the x and y directions

source [ndarray] Source terms

Returns

out [ndarray, ndarray, ndarray, ndarray, ndarray, ndarray, ndarray, ndarray] unsplit predictions of u and v on both the x- and y-interfaces

`lm_atm.LM_atm_interface.`**`is_asymmetric`**

Is the left half of s asymmetric to the right half?

Parameters

ng [int] The number of ghost cells

nodal: bool Is the data nodal?

s [ndarray] The array to be compared

Returns

out [int] Is it asymmetric? (1 = yes, 0 = no)

`lm_atm.LM_atm_interface.`**`is_asymmetric_pair`**

Are sl and sr asymmetric about an axis parallel with the y-axis in the center of domain the x-direction?

Parameters

ng [int] The number of ghost cells

nodal: bool Is the data nodal?

sl, sr [ndarray] The two arrays to be compared

Returns

out [int] Are they asymmetric? (1 = yes, 0 = no)

`lm_atm.LM_atm_interface.`**`is_symmetric`**

Is the left half of s the mirror image of the right half?

Parameters

ng [int] The number of ghost cells

nodal: bool Is the data nodal?

s [ndarray] The array to be compared

Returns

out [int] Is it symmetric? (1 = yes, 0 = no)

`lm_atm.LM_atm_interface.`**`is_symmetric_pair`**

Are sl and sr symmetric about an axis parallel with the y-axis in the center of domain the x-direction?

Parameters

ng [int] The number of ghost cells

nodal: bool Is the data nodal?

sl, sr [ndarray] The two arrays to be compared

Returns

out [int] Are they symmetric? (1 = yes, 0 = no)

`lm_atm.LM_atm_interface.`**`mac_vels`**

Calculate the MAC velocities in the x and y directions.

Parameters

ng [int] The number of ghost cells

dx, dy [float] The cell spacings

dt [float] The timestep

u, v [ndarray] x-velocity and y-velocity

ldelta_ux, ldelta_uy: ndarray Limited slopes of the x-velocity in the x and y directions

ldelta_vx, ldelta_vy: ndarray Limited slopes of the y-velocity in the x and y directions

gradp_x, gradp_y [ndarray] Pressure gradients in the x and y directions

source [ndarray] Source terms

Returns

out [ndarray, ndarray] MAC velocities in the x and y directions

`lm_atm.LM_atm_interface.`**`rho_states`**

This predicts rho to the interfaces. We use the MAC velocities to do the upwinding

Parameters

ng [int] The number of ghost cells

dx, dy [float] The cell spacings

rho [ndarray] density

u_MAC, v_MAC [ndarray] MAC velocities in the x and y directions

ldelta_rx, ldelta_ry: ndarray Limited slopes of the density in the x and y directions

Returns

out [ndarray, ndarray] rho predicted to the interfaces

`lm_atm.LM_atm_interface.`**`riemann`**

Solve the Burger's Riemann problem given the input left and right states and return the state on the interface.

This uses the expressions from Almgren, Bell, and Szymczak 1996.

Parameters

ng [int] The number of ghost cells

q_l, q_r [ndarray] left and right states

Returns

out [ndarray] Interface state

`lm_atm.LM_atm_interface.`**`riemann_and_upwind`**

First solve the Riemann problem given q_l and q_r to give the velocity on the interface and: use this velocity to upwind to determine the state (q_l, q_r, or a mix) on the interface).

This differs from upwind, above, in that we don't take in a velocity to upwind with).

Parameters

ng [int] The number of ghost cells

q_l, q_r [ndarray] left and right states

Returns

out [ndarray] Upwinded state

`lm_atm.LM_atm_interface.`**`states`**

This is similar to `mac_vels`, but it predicts the interface states of both u and v on both interfaces, using the MAC velocities to do the upwinding.

Parameters

ng [int] The number of ghost cells

dx, dy [float] The cell spacings

dt [float] The timestep

u, v [ndarray] x-velocity and y-velocity

ldelta_ux, ldelta_uy: ndarray Limited slopes of the x-velocity in the x and y directions

ldelta_vx, ldelta_vy: ndarray Limited slopes of the y-velocity in the x and y directions

source [ndarray] Source terms

gradp_x, gradp_y [ndarray] Pressure gradients in the x and y directions

u_MAC, v_MAC [ndarray] MAC velocities in the x and y directions

Returns

out [ndarray, ndarray, ndarray, ndarray] x and y velocities predicted to the interfaces

`lm_atm.LM_atm_interface.`**`upwind`**

upwind the left and right states based on the specified input velocity, s. The resulting interface state is q_int

Parameters

ng [int] The number of ghost cells

q_l, q_r [ndarray] left and right states

s [ndarray] velocity

Returns

q_int [ndarray] Upwinded state

25.15.4 lm_atm.simulation module

`class` `lm_atm.simulation.`**`Basestate`**`(`*ny*, *ng=0*`)`

Bases: `object`

`jp``(`*shift*, *buf=0*`)`

`v``(`*buf=0*`)`

`v2d``(`*buf=0*`)`

`v2dp``(`*shift*, *buf=0*`)`

`class` `lm_atm.simulation.`**`Simulation`**`(`*solver_name*, *problem_name*, *rp*, *timers=None*`)`

Bases: *`simulation_null.NullSimulation`*

dovis ()
Do runtime visualization

evolve ()
Evolve the low Mach system through one timestep.

initialize ()
Initialize the grid and variables for low Mach atmospheric flow and set the initial conditions for the chosen problem.

make_prime (*a*, *a0*)

method_compute_timestep ()
The timestep() function computes the advective timestep (CFL) constraint. The CFL constraint says that information cannot propagate further than one zone per timestep.

We use the driver.cfl parameter to control what fraction of the CFL step we actually take.

preevolve ()
preevolve is called before we being the timestepping loop. For the low Mach solver, this does an initial projection on the velocity field and then goes through the full evolution to get the value of phi. The fluid state (rho, u, v) is then reset to values before this evolve.

read_extras (*f*)
read in any simulation-specific data from an h5py file object f

write_extras (*f*)
Output simulation-specific data to the h5py file f

25.16 mesh package

This is the general mesh module for pyro. It implements everything necessary to work with finite-volume data.

25.16.1 Submodules

25.16.2 mesh.array_indexer module

An array class that has methods supporting the type of stencil operations we see in finite-difference methods, like i+1, i-1, etc.

class `mesh.array_indexer.`**ArrayIndexer**
Bases: `numpy.ndarray`

a class that wraps the data region of a single array (d) and allows us to easily do array operations like d[i+1,j] using the ip() method.

copy ()
make a copy of the array, defined on the same grid

fill_ghost (*n=0*, *bc=None*)
Fill the boundary conditions. This operates on a single component, n. We do periodic, reflect-even, reflect-odd, and outflow

We need a BC object to tell us what BC type on each boundary.

ip (*shift*, *buf=0*, *n=0*, *s=1*)
return a view of the data shifted by shift in the x direction. By default the view is the same size as the valid

region, but the buf can specify how many ghost cells on each side to include. The component is n and s is the stride

ip_jp (*ishift*, *jshift*, *buf=0*, *n=0*, *s=1*)
return a view of the data shifted by ishift in the x direction and jshift in the y direction. By default the view is the same size as the valid region, but the buf can specify how many ghost cells on each side to include. The component is n and s is the stride

is_asymmetric (*nodal=False*, *tol=1e-14*)
return True is the data is left-right asymmetric (to the tolerance tol)—e.g, the sign flips. For node-centered data, set nodal=True

is_symmetric (*nodal=False*, *tol=1e-14*, *asymmetric=False*)
return True is the data is left-right symmetric (to the tolerance tol) For node-centered data, set nodal=True

jp (*shift*, *buf=0*, *n=0*, *s=1*)
return a view of the data shifted by shift in the y direction. By default the view is the same size as the valid region, but the buf can specify how many ghost cells on each side to include. The component is n and s is the stride

lap (*n=0*, *buf=0*)
return the 5-point Laplacian

norm (*n=0*)
find the norm of the quantity (index n) defined on the same grid, in the domain's valid region

pretty_print (*n=0*, *fmt=None*, *show_ghost=True*)
Print out a small dataset to the screen with the ghost cells a different color, to make things stand out

v (*buf=0*, *n=0*, *s=1*)
return a view of the valid data region for component n, with stride s, and a buffer of ghost cells given by buf

25.16.3 mesh.boundary module

Methods to manage boundary conditions

class mesh.boundary.**BC** (*xlb='outflow'*, *xrb='outflow'*, *ylb='outflow'*, *yrb='outflow'*, *xl_func=None*, *xr_func=None*, *yl_func=None*, *yr_func=None*, *grid=None*, *odd_reflect_dir=''*)
Bases: object

Boundary condition container – hold the BCs on each boundary for a single variable.

For Neumann and Dirichlet BCs, a function callback can be stored for inhomogeous BCs. This function should provide the value on the physical boundary (not cell center). This is evaluated on the relevant edge when the __init__ routine is called. For this reason, you need to pass in a grid object. Note: this only ensures that the first ghost cells is consistent with the BC value.

class mesh.boundary.**BCProp** (*xl_prop*, *xr_prop*, *yl_prop*, *yr_prop*)
Bases: object

A simple container to hold properties of the boundary conditions.

mesh.boundary.**bc_is_solid** (*bc*)
return a container class indicating which boundaries are solid walls

mesh.boundary.**define_bc** (*bc_type*, *function*, *is_solid=False*)
use this to extend the types of boundary conditions supported on a solver-by-solver basis. Here we pass in the reference to a function that can be called with the data that needs to be filled. is_solid indicates whether it should be interpreted as a solid wall (no flux through the BC)"

25.16.4 mesh.fv module

This implements support for 4th-order accurate finite-volume data by adding support for converting between cell averages and centers.

class mesh.fv.**FV2d**(*grid*, *dtype=<class 'numpy.float64'>*)

Bases: *mesh.patch.CellCenterData2d*

this is a finite-volume grid. We expect the data to represent cell-averages, and do operations to 4th order. This assumes dx = dy

from_centers(*name*)

treat the stored data as if it lives at cell-centers and convert it to an average

to_centers(*name*)

convert variable name from an average to cell-centers

25.16.5 mesh.integration module

A generic Runge-Kutta type integrator for integrating CellCenterData2d. We support a generic Butcher tableau for explicit the Runge-Kutta update:

```
0    |
c_2  | a_21
c_3  | a_31 a_32
:    |   :         .
:    |   :           .
c_s  | a_s1 a_s2 ... a_s,s-1
----+---------------------------
     | b_1  b_2  ... b_{s-1}  b_s
```

the update is:

```
y_{n+1} = y_n + dt sum_{i=1}^s {b_i k_i}
```

and the s increment is:

```
k_s = f(t + c_s dt, y_n + dt (a_s1 k1 + a_s2 k2 + ... + a_s,s-1 k_{s-1})
```

class mesh.integration.**RKIntegrator**(*t*, *dt*, *method='RK4'*)

Bases: object

the integration class for CellCenterData2d, supporting RK integration

compute_final_update()

this constructs the final t + dt update, overwriting the inital data

get_stage_start(*istage*)

get the starting conditions (a CellCenterData2d object) for stage istage

nstages()

return the number of stages

set_start(*start*)

store the starting conditions (should be a CellCenterData2d object)

store_increment(*istage*, *k_stage*)

store the increment for stage istage – this should not have a dt weighting

25.16.6 mesh.patch module

The patch module defines the classes necessary to describe finite-volume data and the grid that it lives on.

Typical usage:

- create the grid:

```
grid = Grid2d(nx, ny)
```

- create the data that lives on that grid:

```
data = CellCenterData2d(grid)

bc = BC(xlb="reflect", xrb="reflect",
        ylb="outflow", yrb="outflow")
data.register_var("density", bc)
...

data.create()
```

- initialize some data:

```
dens = data.get_var("density")
dens[:, :] = ...
```

- fill the ghost cells:

```
data.fill_BC("density")
```

class mesh.patch.**CellCenterData2d**(*grid*, *dtype=<class 'numpy.float64'>*)
Bases: `object`

A class to define cell-centered data that lives on a grid. A CellCenterData2d object is built in a multi-step process before it can be used.

- Create the object. We pass in a grid object to describe where the data lives:

```
my_data = patch.CellCenterData2d(myGrid)
```

- Register any variables that we expect to live on this patch. Here BC describes the boundary conditions for that variable:

```
my_data.register_var('density', BC)
my_data.register_var('x-momentum', BC)
...
```

- Register any auxillary data – these are any parameters that are needed to interpret the data outside of the simulation (for example, the gamma for the equation of state):

```
my_data.set_aux(keyword, value)
```

- Finish the initialization of the patch:

```
my_data.create()
```

This last step actually allocates the storage for the state variables. Once this is done, the patch is considered to be locked. New variables cannot be added.

add_derived(*func*)
Register a function to compute derived variable

Parameters

func [function] A function to call to derive the variable. This function should take two arguments, a CellCenterData2d object and a string variable name (or list of variables)

create()
Called after all the variables are registered and allocates the storage for the state data.

fill_BC(*name*)
Fill the boundary conditions. This operates on a single state variable at a time, to allow for maximum flexibility.

We do periodic, reflect-even, reflect-odd, and outflow

Each variable name has a corresponding BC stored in the CellCenterData2d object – we refer to this to figure out the action to take at each boundary.

Parameters

name [str] The name of the variable for which to fill the BCs.

fill_BC_all()
Fill boundary conditions on all variables.

get_aux(*keyword*)
Get the auxillary data associated with keyword

Parameters

keyword [str] The name of the auxillary data to access

Returns

out [variable type] The value corresponding to the keyword

get_var(*name*)
Return a data array for the variable described by name. Stored variables will be checked first, and then any derived variables will be checked.

For a stored variable, changes made to this are automatically reflected in the CellCenterData2d object.

Parameters

name [str] The name of the variable to access

Returns

out [ndarray] The array of data corresponding to the variable name

get_var_by_index(*n*)
Return a data array for the variable with index n in the data array. Any changes made to this are automatically reflected in the CellCenterData2d object.

Parameters

n [int] The index of the variable to access

Returns

out [ndarray] The array of data corresponding to the index

get_vars()
Return the entire data array. Any changes made to this are automatically reflected in the CellCenterData2d object.

Returns

out [ndarray] The array of data

max (*name*, *ng=0*)
return the maximum of the variable name in the domain's valid region

min (*name*, *ng=0*)
return the minimum of the variable name in the domain's valid region

pretty_print (*var*, *fmt=None*)
print out the contents of the data array with pretty formatting indicating where ghost cells are.

prolong (*varname*)
Prolong the data in the current (coarse) grid to a finer (factor of 2 finer) grid. Return an array with the resulting data (and same number of ghostcells). Only the data for the variable varname will be operated upon.

We will reconstruct the data in the zone from the zone-averaged variables using the same limited slopes as in the advection routine. Getting a good multidimensional reconstruction polynomial is hard – we want it to be bilinear and monotonic – we settle for having each slope be independently monotonic:

```
              (x)          (y)
f(x,y) = m     x/dx + m     y/dy + <f>
```

where the m's are the limited differences in each direction. When averaged over the parent cell, this reproduces <f>.

Each zone's reconstrution will be averaged over 4 children:

```
+-----------+     +-----+-----+
|           |     |     |     |
|           |     |  3  |  4  |
|    <f>    | --> +-----+-----+
|           |     |     |     |
|           |     |  1  |  2  |
+-----------+     +-----+-----+
```

We will fill each of the finer resolution zones by filling all the 1's together, using a stride 2 into the fine array. Then the 2's and . . . , this allows us to operate in a vector fashion. All operations will use the same slopes for their respective parents.

register_var (*name*, *bc*)
Register a variable with CellCenterData2d object.

Parameters

name [str] The variable name

bc [BC object] The boundary conditions that describe the actions to take for this variable at the physical domain boundaries.

restrict (*varname*, *N=2*)
Restrict the variable varname to a coarser grid (factor of 2 coarser) and return an array with the resulting data (and same number of ghostcells)

set_aux (*keyword*, *value*)
Set any auxillary (scalar) data. This data is simply carried along with the CellCenterData2d object

Parameters

keyword [str] The name of the datum

value [any time] The value to associate with the keyword

write (*filename*)
create an output file in HDF5 format and write out our data and grid.

write_data (*f*)
write the data out to an hdf5 file – here, f is an h5py File pbject

zero (*name*)
Zero out the data array associated with variable name.

Parameters

name [str] The name of the variable to zero

class mesh.patch.**FaceCenterData2d** (*grid*, *idir*, *dtype=<class 'numpy.float64'>*)
Bases: *mesh.patch.CellCenterData2d*

A class to define face-centered data that lives on a grid. Data can be face-centered in x or y. This is built in the same multistep process as a CellCenterData2d object

add_derived (*func*)
Register a function to compute derived variable

Parameters

func [function] A function to call to derive the variable. This function should take two arguments, a CellCenterData2d object and a string variable name (or list of variables)

create ()
Called after all the variables are registered and allocates the storage for the state data. For face-centered data, we have one more zone in the face-centered direction.

fill_BC (*name*)
Fill the boundary conditions. This operates on a single state variable at a time, to allow for maximum flexibility.

We do periodic, reflect-even, reflect-odd, and outflow

Each variable name has a corresponding BC stored in the CellCenterData2d object – we refer to this to figure out the action to take at each boundary.

Parameters

name [str] The name of the variable for which to fill the BCs.

prolong (*varname*)
Prolong the data in the current (coarse) grid to a finer (factor of 2 finer) grid. Return an array with the resulting data (and same number of ghostcells). Only the data for the variable varname will be operated upon.

We will reconstruct the data in the zone from the zone-averaged variables using the same limited slopes as in the advection routine. Getting a good multidimensional reconstruction polynomial is hard – we want it to be bilinear and monotonic – we settle for having each slope be independently monotonic:

```
              (x)          (y)
f(x,y) = m     x/dx + m     y/dy + <f>
```

where the m's are the limited differences in each direction. When averaged over the parent cell, this reproduces <f>.

Each zone's reconstrution will be averaged over 4 children:

```
+-----------+     +-----+-----+
|           |     |     |     |
|           |     |  3  |  4  |
|    <f>    | --> +-----+-----+
|           |     |     |     |
|           |     |  1  |  2  |
+-----------+     +-----+-----+
```

We will fill each of the finer resolution zones by filling all the 1's together, using a stride 2 into the fine array. Then the 2's and . . . , this allows us to operate in a vector fashion. All operations will use the same slopes for their respective parents.

restrict (*varname*, *N=2*)
Restrict the variable varname to a coarser grid (factor of 2 coarser) and return an array with the resulting data (and same number of ghostcells)

write_data (*f*)
write the data out to an hdf5 file – here, f is an h5py File pbject

class mesh.patch.**Grid2d** (*nx*, *ny*, *ng=1*, *xmin=0.0*, *xmax=1.0*, *ymin=0.0*, *ymax=1.0*)
Bases: `object`

the 2-d grid class. The grid object will contain the coordinate information (at various centerings).

A basic (1-d) representation of the layout is:

```
|     |      |     X     |     |      |     |     X     |      |     |
+--*--+- // -+--*--X--*--+--*--+- // -+--*--+--*--X--*--+- // -+--*--+
   0          ng-1    ng   ng+1         ... ng+nx-1 ng+nx      2ng+nx-1

                     ilo                      ihi

|<- ng guardcells->|<---- nx interior zones ----->|<- ng guardcells->|
```

The '*' marks the data locations.

coarse_like (*N*)
return a new grid object coarsened by a factor n, but with all the other properties the same

fine_like (*N*)
return a new grid object finer by a factor n, but with all the other properties the same

scratch_array (*nvar=1*)
return a standard numpy array dimensioned to have the size and number of ghostcells as the parent grid

mesh.patch.**cell_center_data_clone** (*old*)
Create a new CellCenterData2d object that is a copy of an existing one

Parameters

old [CellCenterData2d object] The CellCenterData2d object we wish to copy

mesh.patch.**do_demo** ()
show examples of the patch methods / classes

25.16.7 mesh.reconstruction module

Support for computing limited differences needed in reconstruction of slopes in constructing interface states.

`mesh.reconstruction.flatten(myg, q, idir, ivars, rp)`
compute the 1-d flattening coefficients

`mesh.reconstruction.flatten_multid(myg, q, xi_x, xi_y, ivars)`
compute the multidimensional flattening coefficient

`mesh.reconstruction.limit(data, myg, idir, limiter)`
a single driver that calls the different limiters based on the value of the limiter input variable.

`mesh.reconstruction.limit2(a, myg, idir)`
2nd order monotonized central difference limiter

`mesh.reconstruction.limit4(a, myg, idir)`
4th order monotonized central difference limiter

`mesh.reconstruction.nolimit(a, myg, idir)`
just a centered difference without any limiting

`mesh.reconstruction.well_balance(q, myg, limiter, iv, grav)`
subtract off the hydrostatic pressure before limiting. Note, this only considers the y direction.

`mesh.reconstruction.weno(q, order)`
Perform WENO reconstruction

Parameters

q [np array] input data with 3 ghost zones

order [int] WENO order (k)

Returns

q_plus, q_minus [np array] data reconstructed to the right / left respectively

`mesh.reconstruction.weno_upwind(q, order)`
Perform upwinded (left biased) WENO reconstruction

Parameters

q [np array] input data

order [int] WENO order (k)

Returns

q_plus [np array] data reconstructed to the right

25.17 multigrid package

This is the pyro multigrid solver. There are several versions.

MG implements a second-order discretization of a constant-coefficient Helmholtz equation:

$$(\alpha - \beta L)\phi = f$$

variable_coeff_MG implements a variable-coefficient Poisson equation

$$\nabla \cdot \eta \nabla \phi = f$$

general_MG implements a more general elliptic equation

$$\alpha \phi + \nabla \cdot \beta \nabla \phi + \gamma \cdot \nabla \phi = f$$

All use pure V-cycles to solve elliptic problems

25.17.1 Submodules

25.17.2 multigrid.MG module

The multigrid module provides a framework for solving elliptic problems. A multigrid object is just a list of grids, from the finest mesh down (by factors of two) to a single interior zone (each grid has the same number of guardcells).

The main multigrid class is setup to solve a constant-coefficient Helmholtz equation

$$(\alpha - \beta L)\phi = f$$

where L is the Laplacian and α and β are constants. If $\alpha = 0$ and $\beta = -1$, then this is the Poisson equation.

We support Dirichlet or Neumann BCs, or a periodic domain.

The general usage is as follows:

```
a = multigrid.CellCenterMG2d(nx, ny, verbose=1, alpha=alpha, beta=beta)
```

this creates the multigrid object a, with a finest grid of nx by ny zones and the default boundary condition types. α and β are the coefficients of the Helmholtz equation. Setting verbose = 1 causing debugging information to be output, so you can see the residual errors in each of the V-cycles.

Initialization is done as:

```
a.init_zeros()
```

this initializes the solution vector with zeros (this is not necessary if you just created the multigrid object, but it can be used to reset the solution between runs on the same object).

Next:

```
a.init_RHS(zeros((nx, ny), numpy.float64))
```

this initializes the RHS on the finest grid to 0 (Laplace's equation). Any RHS can be set by passing through an array of (nx, ny) values here.

Then to solve, you just do:

```
a.solve(rtol = 1.e-10)
```

where rtol is the desired tolerance (residual norm / source norm)

to access the final solution, use the get_solution method:

```
v = a.get_solution()
```

For convenience, the grid information on the solution level is available as attributes to the class,

a.ilo, a.ihi, a.jlo, a.jhi are the indices bounding the interior of the solution array (i.e. excluding the ghost cells).

a.x and a.y are the coordinate arrays a.dx and a.dy are the grid spacings

class multigrid.MG.**CellCenterMG2d**(*nx, ny, ng=1, xmin=0.0, xmax=1.0, ymin=0.0, ymax=1.0, xl_BC_type='dirichlet', xr_BC_type='dirichlet', yl_BC_type='dirichlet', yr_BC_type='dirichlet', xl_BC=None, xr_BC=None, yl_BC=None, yr_BC=None, alpha=0.0, beta=-1.0, nsmooth=10, nsmooth_bottom=50, verbose=0, aux_field=None, aux_bc=None, true_function=None, vis=0, vis_title=''*)

Bases: `object`

The main multigrid class for cell-centered data.

We require that nx = ny be a power of 2 and dx = dy, for simplicity

get_solution(*grid=None*)

Return the solution after doing the MG solve

If a grid object is passed in, then the solution is put on that grid – not the passed in grid must have the same dx and dy

Returns

out [ndarray]

get_solution_gradient(*grid=None*)

Return the gradient of the solution after doing the MG solve. The x- and y-components are returned in separate arrays.

If a grid object is passed in, then the gradient is computed on that grid. Note: the passed-in grid must have the same dx, dy

Returns

out [ndarray, ndarray]

get_solution_object()

Return the full solution data object at the finest resolution after doing the MG solve

Returns

out [CellCenterData2d object]

grid_info(*level*, *indent=0*)

Report simple grid information

init_RHS(*data*)

Initialize the right hand side, f, of the Helmholtz equation $(\alpha - \beta L)\phi = f$

Parameters

data [ndarray] An array (of the same size as the finest MG level) with the values to initialize the solution to the elliptic problem.

init_solution(*data*)

Initialize the solution to the elliptic problem by passing in a value for all defined zones

Parameters

data [ndarray] An array (of the same size as the finest MG level) with the values to initialize the solution to the elliptic problem.

init_zeros()

Set the initial solution to zero

smooth(*level*, *nsmooth*)

Use red-black Gauss-Seidel iterations to smooth the solution at a given level. This is used at each stage of the V-cycle (up and down) in the MG solution, but it can also be called directly to solve the elliptic problem (although it will take many more iterations).

Parameters

level [int] The level in the MG hierarchy to smooth the solution

nsmooth [int] The number of r-b Gauss-Seidel smoothing iterations to perform

solve (*rtol=1e-11*)

The main driver for the multigrid solution of the Helmholtz equation. This controls the V-cycles, smoothing at each step of the way and uses simple smoothing at the coarsest level to perform the bottom solve.

Parameters

rtol [float] The relative tolerance (residual norm / source norm) to solve to. Note that if the source norm is 0 (e.g. the righthand side of our equation is 0), then we just use the norm of the residual.

v_cycle (*level*)

Perform a V-cycle for a single 2-level solve. This is applied recursively do V-cycle through the entire hierarchy.

25.17.3 multigrid.edge_coeffs module

class `multigrid.edge_coeffs.`**EdgeCoeffs** (*g*, *eta*, *empty=False*)

Bases: `object`

a simple container class to hold edge-centered coefficients and restrict them to coarse levels

restrict ()

restrict the edge values to a coarser grid. Return a new EdgeCoeffs object

25.17.4 multigrid.general_MG module

This multigrid solver is build from multigrid/MG.py and implements a more general solver for an equation of the form

$$\alpha\phi + \nabla \cdot \beta\nabla\phi + \gamma \cdot \nabla\phi = f$$

where alpha, beta, and gamma are defined on the same grid as phi. These should all come in as cell-centered quantities. The solver will put beta on edges. Note that gamma is a vector here, with x- and y-components.

A cell-centered discretization for phi is used throughout.

class `multigrid.general_MG.`**GeneralMG2d** (*nx*, *ny*, *xmin=0.0*, *xmax=1.0*, *ymin=0.0*, *ymax=1.0*, *xl_BC_type='dirichlet'*, *xr_BC_type='dirichlet'*, *yl_BC_type='dirichlet'*, *yr_BC_type='dirichlet'*, *xl_BC=None*, *xr_BC=None*, *yl_BC=None*, *yr_BC=None*, *nsmooth=10*, *nsmooth_bottom=50*, *verbose=0*, *coeffs=None*, *true_function=None*, *vis=0*, *vis_title=''*)

Bases: *multigrid.MG.CellCenterMG2d*

this is a multigrid solver that supports our general elliptic equation.

we need to accept a coefficient `CellCenterData2d` object with fields defined for `alpha`, `beta`, `gamma_x`, and `gamma_y` on the fine level.

We then restrict this data through the MG hierarchy (and average beta to the edges).

we need a `new compute_residual()` and `smooth()` function, that understands these coeffs.

smooth (*level*, *nsmooth*)

Use red-black Gauss-Seidel iterations to smooth the solution at a given level. This is used at each stage of the V-cycle (up and down) in the MG solution, but it can also be called directly to solve the elliptic problem (although it will take many more iterations).

Parameters

level [int] The level in the MG hierarchy to smooth the solution

nsmooth [int] The number of r-b Gauss-Seidel smoothing iterations to perform

25.17.5 multigrid.variable_coeff_MG module

This multigrid solver is build from multigrid/MG.py and implements a variable coefficient solver for an equation of the form

$$\nabla \cdot \eta \nabla \phi = f$$

where η is defined on the same grid as ϕ.

A cell-centered discretization is used throughout.

class multigrid.variable_coeff_MG.**VarCoeffCCMG2d**(*nx, ny, xmin=0.0, xmax=1.0, ymin=0.0, ymax=1.0, xl_BC_type='dirichlet', xr_BC_type='dirichlet', yl_BC_type='dirichlet', yr_BC_type='dirichlet', nsmooth=10, nsmooth_bottom=50, verbose=0, coeffs=None, coeffs_bc=None, true_function=None, vis=0, vis_title=''*)

Bases: *multigrid.MG.CellCenterMG2d*

this is a multigrid solver that supports variable coefficients

we need to accept a coefficient array, coeffs, defined at each level. We can do this at the fine level and restrict it down the MG grids once.

we need a new `compute_residual()` and `smooth()` function, that understands coeffs.

smooth(*level, nsmooth*)

Use red-black Gauss-Seidel iterations to smooth the solution at a given level. This is used at each stage of the V-cycle (up and down) in the MG solution, but it can also be called directly to solve the elliptic problem (although it will take many more iterations).

Parameters

level [int] The level in the MG hierarchy to smooth the solution

nsmooth [int] The number of r-b Gauss-Seidel smoothing iterations to perform

25.18 particles package

Particles routines

25.18.1 Submodules

25.18.2 particles.particles module

Stores and manages particles and updates their positions based on the velocity on the grid.

class particles.particles.**Particle**(*x*, *y*, *u=0*, *v=0*)
Bases: object

Class to hold properties of a single (massless) particle.

This class could be extended (i.e. inherited from) to model e.g. massive/charged particles.

interpolate_velocity(*myg*, *u*, *v*)
Interpolate the x- and y-velocities defined on grid myg to the particle's position.

Parameters

myg [Grid2d] grid which the velocities are defined on

u [ArrayIndexer] x-velocity

v [ArrayIndexer] y_velocity

pos()
Return position vector.

update(*u*, *v*, *dt*)
Advect the particle and update its velocity.

velocity()
Return velocity vector.

class particles.particles.**Particles**(*sim_data*, *bc*, *n_particles*, *particle_generator='grid'*, *pos_array=None*, *init_array=None*)
Bases: object

Class to hold multiple particles.

array_generate_particles(*pos_array*, *init_array=None*)
Generate particles based on array of their positions. This is used when reading particle data from file.

Parameters

pos_array [float array] Array of particle positions.

init_array [float array] Array of initial particle positions.

enforce_particle_boundaries()
Enforce the particle boundaries

TODO: copying the dict and adding everything back again is messy - think of a better way to do this? Did it this way so don't have to remove items from a dictionary while iterating it for outflow boundaries.

get_init_positions()
Return initial positions of the particles as an array.

We defined the particles as a dictionary with their initial positions as the keys, so this just becomes a restructuring operation.

get_positions()
Return an array of current particle positions.

grid_generate_particles(*n_particles*)
Generate particles equally spaced across the grid. Currently has the same number of particles in the x and y directions (so dx != dy unless the domain is square) - may be better to scale this.

If necessary, shall increase/decrease n_particles in order to fill grid.

randomly_generate_particles(*n_particles*)
Randomly generate n_particles.

update_particles(*dt*, *u=None*, *v=None*)
Update the particles on the grid. This is based off the `AdvectWithUcc` function in AMReX, which used the midpoint method to advance particles using the cell-centered velocity.

We will explicitly pass in u and v if they cannot be accessed from the `sim_data` using `get_var("velocity")`.

Parameters

dt [float] timestep

u [ArrayIndexer object] x-velocity

v [ArrayIndexer object] y-velocity

write_particles(*f*)
Output the particles' positions (and initial positions) to an HDF5 file.

Parameters

f [h5py object] HDF5 file to write to

25.19 plot module

`plot.`**get_args**()

`plot.`**makeplot**(*plotfile_name*, *outfile*, *width*, *height*)
plot the data in a plotfile using the solver's vis() method

25.20 pyro module

class `pyro.`**Pyro**(*solver_name*)
Bases: `object`

The main driver to run pyro.

get_var(*v*)
Alias for cc_data's get_var routine, returns the cell-centered data given the variable name v.

initialize_problem(*problem_name*, *inputs_file=None*, *inputs_dict=None*, *other_commands=None*)
Initialize the specific problem

Parameters

problem_name [str] Name of the problem

inputs_file [str] Filename containing problem's runtime parameters

inputs_dict [dict] Dictionary containing extra runtime parameters

other_commands [str] Other command line parameter options

run_sim()
Evolve entire simulation

single_step()
Do a single step

class pyro.**PyroBenchmark**(*solver_name*, *comp_bench=False*, *reset_bench_on_fail=False*, *make_bench=False*)

Bases: *pyro.Pyro*

A subclass of Pyro for benchmarking. Inherits everything from pyro, but adds benchmarking routines.

compare_to_benchmark(*rtol*)
Are we comparing to a benchmark?

run_sim(*rtol*)
Evolve entire simulation and compare to benchmark at the end.

store_as_benchmark()
Are we storing a benchmark?

pyro.**parse_args**()
Parse the runtime parameters

25.21 simulation_null module

class simulation_null.**NullSimulation**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: object

compute_timestep()
a generic wrapper for computing the timestep that respects the driver parameters on timestepping

do_output()
is it time to output?

dovis()

evolve()

finalize()
Do any final clean-ups for the simulation and call the problem's finalize() method.

finished()
is the simulation finished based on time or the number of steps

initialize()

method_compute_timestep()
the method-specific timestep code

preevolve()
Do any necessary evolution before the main evolve loop. This is not needed for advection

read_extras(*f*)
read in any simulation-specific data from an h5py file object f

write(*filename*)
Output the state of the simulation to an HDF5 file for plotting

write_extras(*f*)
write out any extra simulation-specific stuff

simulation_null.**bc_setup**(*rp*)

simulation_null.**grid_setup**(*rp*, *ng=1*)

25.22 swe package

The pyro swe hydrodynamics solver. This implements the second-order (piecewise-linear), unsplit method of Colella 1990.

25.22.1 Subpackages

swe.problems package

Submodules

swe.problems.acoustic_pulse module

swe.problems.acoustic_pulse.**finalize**()
: print out any information to the user at the end of the run

swe.problems.acoustic_pulse.**init_data**(*myd*, *rp*)
: initialize the acoustic_pulse problem. This comes from McCourquodale & Coella 2011

swe.problems.advect module

swe.problems.advect.**finalize**()
: print out any information to the user at the end of the run

swe.problems.advect.**init_data**(*my_data*, *rp*)
: initialize a smooth advection problem for testing convergence

swe.problems.dam module

swe.problems.dam.**finalize**()
: print out any information to the user at the end of the run

swe.problems.dam.**init_data**(*my_data*, *rp*)
: initialize the dam problem

swe.problems.kh module

swe.problems.kh.**finalize**()
: print out any information to the user at the end of the run

swe.problems.kh.**init_data**(*my_data*, *rp*)
: initialize the Kelvin-Helmholtz problem

swe.problems.logo module

swe.problems.logo.**finalize**()
: print out any information to the user at the end of the run

swe.problems.logo.**init_data**(*my_data*, *rp*)
: initialize the sedov problem

swe.problems.quad module

swe.problems.quad.**finalize**()
print out any information to the user at the end of the run

swe.problems.quad.**init_data**(*my_data*, *rp*)
initialize the quadrant problem

swe.problems.test module

swe.problems.test.**finalize**()
print out any information to the user at the end of the run

swe.problems.test.**init_data**(*my_data*, *rp*)
an init routine for unit testing

25.22.2 Submodules

25.22.3 swe.derives module

swe.derives.**derive_primitives**(*myd*, *varnames*)
derive desired primitive variables from conserved state

25.22.4 swe.interface module

swe.interface.**consFlux**
Calculate the conserved flux for the shallow water equations. In the x-direction, this is given by:

```
    /      hu       \
F = | hu^2 + gh^2/2 |
    \      huv      /
```

Parameters

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

g [float] Graviational acceleration

ih, ixmom, iymom, ihX [int] The indices of the height, x-momentum, y-momentum, height*species fraction in the conserved state vector.

nspec [int] The number of species

U_state [ndarray] Conserved state vector.

Returns

out [ndarray] Conserved flux

swe.interface.**riemann_hllc**
this is the HLLC Riemann solver. The implementation follows directly out of Toro's book. Note: this does not handle the transonic rarefaction.

Parameters

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

ng [int] The number of ghost cells

ih, ixmom, iymom, ihX [int] The indices of the height, x-momentum, y-momentum and height*species fractions in the conserved state vector.

nspec [int] The number of species

lower_solid, upper_solid [int] Are we at lower or upper solid boundaries?

g [float] Gravitational acceleration

U_l, U_r [ndarray] Conserved state on the left and right cell edges.

Returns

out [ndarray] Conserved flux

`swe.interface.`**`riemann_roe`**

This is the Roe Riemann solver with entropy fix. The implementation follows Toro's SWE book and the clawpack 2d SWE Roe solver.

Parameters

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

ng [int] The number of ghost cells

ih, ixmom, iymom, ihX [int] The indices of the height, x-momentum, y-momentum and height*species fractions in the conserved state vector.

nspec [int] The number of species

lower_solid, upper_solid [int] Are we at lower or upper solid boundaries?

g [float] Gravitational acceleration

U_l, U_r [ndarray] Conserved state on the left and right cell edges.

Returns

out [ndarray] Conserved flux

`swe.interface.`**`states`**

predict the cell-centered state to the edges in one-dimension using the reconstructed, limited slopes.

We follow the convection here that `V_l[i]` is the left state at the i-1/2 interface and `V_l[i+1]` is the left state at the i+1/2 interface.

We need the left and right eigenvectors and the eigenvalues for the system projected along the x-direction

Taking our state vector as $Q = (\rho, u, v, p)^T$, the eigenvalues are $u - c$, u, $u + c$.

We look at the equations of hydrodynamics in a split fashion – i.e., we only consider one dimension at a time.

Considering advection in the x-direction, the Jacobian matrix for the primitive variable formulation of the Euler equations projected in the x-direction is:

```
    / u   0   0 \
    | g   u   0 |
A = \ 0   0   u /
```

The right eigenvectors are:

```
     /  h  \       /  0  \      /  h  \
r1 = | -c  |  r2 = |  0  | r3 = |  c  |
     \  0  /       \  1  /      \  0  /
```

The left eigenvectors are:

```
l1 =     ( 1/(2h),  -h/(2hc),  0 )
l2 =     ( 0,          0,  1 )
l3 =     ( -1/(2h), -h/(2hc),  0 )
```

The fluxes are going to be defined on the left edge of the computational zones:

```
 |             |             |             |
 |             |             |             |
-+------+------+------+------+------+------+--
 |     i-1     |      i      |     i+1     |
             ^ ^           ^
         q_l,i q_r,i  q_l,i+1
```

`q_r,i` and `q_l,i+1` are computed using the information in zone i,j.

Parameters

idir [int] Are we predicting to the edges in the x-direction (1) or y-direction (2)?

ng [int] The number of ghost cells

dx [float] The cell spacing

dt [float] The timestep

ih, iu, iv, ix [int] Indices of the height, x-velocity, y-velocity and species in the state vector

nspec [int] The number of species

g [float] Gravitational acceleration

qv [ndarray] The primitive state vector

dqv [ndarray] Spatial derivitive of the state vector

Returns

out [ndarray, ndarray] State vector predicted to the left and right edges

25.22.5 swe.simulation module

class `swe.simulation.`**`Simulation`**(*solver_name*, *problem_name*, *rp*, *timers=None*, *data_class=<class 'mesh.patch.CellCenterData2d'>*)

Bases: *simulation_null.NullSimulation*

The main simulation class for the corner transport upwind swe hydrodynamics solver

`dovis`()

Do runtime visualization.

`evolve`()

Evolve the equations of swe hydrodynamics through a timestep dt.

`initialize`(*extra_vars=None*, *ng=4*)

Initialize the grid and variables for swe flow and set the initial conditions for the chosen problem.

`method_compute_timestep`()

The timestep function computes the advective timestep (CFL) constraint. The CFL constraint says that information cannot propagate further than one zone per timestep.

We use the driver.cfl parameter to control what fraction of the CFL step we actually take.

class swe.simulation.**Variables**(*myd*)

Bases: `object`

a container class for easy access to the different swe variables by an integer key

swe.simulation.**cons_to_prim**(*U, g, ivars, myg*)

Convert an input vector of conserved variables $U = (h, hu, hv, hX)$ to primitive variables $q = (h, u, v, X)$.

swe.simulation.**prim_to_cons**(*q, g, ivars, myg*)

Convert an input vector of primitive variables $q = (h, u, v, X)$ to conserved variables $U = (h, hu, hv, hX)$

25.22.6 swe.unsplit_fluxes module

Implementation of the Colella 2nd order unsplit Godunov scheme. This is a 2-dimensional implementation only. We assume that the grid is uniform, but it is relatively straightforward to relax this assumption.

There are several different options for this solver (they are all discussed in the Colella paper).

- limiter: 0 = no limiting; 1 = 2nd order MC limiter; 2 = 4th order MC limiter
- riemann: HLLC or Roe-fix
- use_flattening: set to 1 to use the multidimensional flattening at shocks
- delta, z0, z1: flattening parameters (we use Colella 1990 defaults)

The grid indices look like:

```
j+3/2--+---------+---------+---------+
       |         |         |         |
  j+1 _|         |         |         |
       |         |         |         |
       |         |         |         |
j+1/2--+---------XXXXXXXXXXX---------+
       |         X         X         |
    j _|         X         X         |
       |         X         X         |
       |         X         X         |
j-1/2--+---------XXXXXXXXXXX---------+
       |         |         |         |
  j-1 _|         |         |         |
       |         |         |         |
       |         |         |         |
j-3/2--+---------+---------+---------+
       |    |    |    |    |    |    |
           i-1        i        i+1
     i-3/2     i-1/2     i+1/2     i+3/2
```

We wish to solve

$$U_t + F^x_x + F^y_y = H$$

we want U_{i+1/2}^{n+1/2} – the interface values that are input to the Riemann problem through the faces for each zone.

Taylor expanding yields:

```
 n+1/2                       dU           dU
U          = U   + 0.5 dx  --   + 0.5 dt  --
 i+1/2,j,L    i,j          dx             dt
```

(continues on next page)

(continued from previous page)

```
                     dU               dF^x    dF^y
   = U    + 0.5 dx  --   - 0.5 dt  ( ---- + ---- - H )
      i,j            dx                dx      dy


                      dU       dF^x             dF^y
   = U    + 0.5  ( dx -- - dt ---- ) - 0.5 dt ---- + 0.5 dt H
      i,j             dx       dx                dy


                          dt       dU            dF^y
   = U    + 0.5 dx ( 1 - -- A^x )  --   - 0.5 dt ---- + 0.5 dt H
      i,j                 dx        dx            dy


                        dt        _              dF^y
   = U    + 0.5   ( 1 - -- A^x ) DU  - 0.5 dt ---- + 0.5 dt H
      i,j               dx                        dy

            +----------+-----------+  +----+----+   +---+---+
                       |                   |             |

           this is the monotonized   this is the   source term
           central difference term   transverse
                                     flux term
```

There are two components, the central difference in the normal to the interface, and the transverse flux difference. This is done for the left and right sides of all 4 interfaces in a zone, which are then used as input to the Riemann problem, yielding the 1/2 time interface values:

```
 n+1/2
U
 i+1/2,j
```

Then, the zone average values are updated in the usual finite-volume way:

```
 n+1     n      dt     x   n+1/2         x   n+1/2
U     = U     + -- { F  (U        ) - F  (U        ) }
 i,j     i,j    dx         i-1/2,j           i+1/2,j


                dt     y   n+1/2         y   n+1/2
              + -- { F  (U        ) - F  (U        ) }
                dy         i,j-1/2           i,j+1/2
```

Updating U_{i,j}:

- We want to find the state to the left and right (or top and bottom) of each interface, ex. U_{i+1/2,j,[lr]}^{n+1/2}, and use them to solve a Riemann problem across each of the four interfaces.
- U_{i+1/2,j,[lr]}^{n+1/2} is comprised of two parts, the computation of the monotonized central differences in the normal direction (eqs. 2.8, 2.10) and the computation of the transverse derivatives, which requires the solution of a Riemann problem in the transverse direction (eqs. 2.9, 2.14).
  - the monotonized central difference part is computed using the primitive variables.

 – We compute the central difference part in both directions before doing the transverse flux differencing, since for the high-order transverse flux implementation, we use these as the input to the transverse Riemann problem.

`swe.unsplit_fluxes.`**`unsplit_fluxes`**(*my_data*, *my_aux*, *rp*, *ivars*, *solid*, *tc*, *dt*)

unsplitFluxes returns the fluxes through the x and y interfaces by doing an unsplit reconstruction of the interface values and then solving the Riemann problem through all the interfaces at once

The runtime parameter g is assumed to be the gravitational acceleration in the y-direction

Parameters

my_data [CellCenterData2d object] The data object containing the grid and advective scalar that we are advecting.

rp [RuntimeParameters object] The runtime parameters for the simulation

vars [Variables object] The Variables object that tells us which indices refer to which variables

tc [TimerCollection object] The timers we are using to profile

dt [float] The timestep we are advancing through.

Returns

out [ndarray, ndarray] The fluxes on the x- and y-interfaces

25.23 test module

`class` `test.`**`PyroTest`**(*solver*, *problem*, *inputs*, *options*)

Bases: `object`

`test.`**`do_tests`**(*build*, *out_file*, *do_standalone=True*, *do_main=True*, *reset_fails=False*, *store_all_benchmarks=False*, *single=None*, *solver=None*, *rtol=1e-12*)

25.24 util package

This module provides utility functions for pyro

25.24.1 Submodules

25.24.2 util.io module

This manages the reading of the HDF5 output files for pyro.

`util.io.`**`read`**(*filename*)

read an HDF5 file and recreate the simulation object that holds the data and state of the simulation.

`util.io.`**`read_bcs`**(*f*)

read in the boundary condition record from the HDF5 file

25.24.3 util.msg module

support output in highlighted colors

`util.msg.`**`bold`**`(`*string*`)`
Output a string in a bold weight

`util.msg.`**`fail`**`(`*string*`)`
Output a string to the terminal and abort if we are running non-interactively. The string is colored red to indicate a failure

`util.msg.`**`success`**`(`*string*`)`
Output a string to the terminal colored green to indicate success

`util.msg.`**`warning`**`(`*string*`)`
Output a string to the terminal colored orange to indicate a warning

25.24.4 util.plot_tools module

Some basic support routines for configuring the plots during runtime visualization

`util.plot_tools.`**`setup_axes`**`(`*myg*, *num*`)`
create a grid of axes whose layout depends on the aspect ratio of the domain

25.24.5 util.profile module

A very simple profiling class, to use to determine where most of the time is spent in a code. This supports nested timers and outputs a report at the end.

Warning: At present, no enforcement is done to ensure proper nesting.

`class` `util.profile.`**`Timer`**`(`*name*, *stack_count=0*`)`
Bases: `object`

A single timer – this simply stores the accumulated time for a single named region

`begin``()`
Start timing

`end``()`
Stop timing. This does not destroy the timer, it simply stops it from counting time.

`class` `util.profile.`**`TimerCollection`**
Bases: `object`

A timer collection—this manages the timers and has methods to start and stop them. Nesting of timers is tracked so we can pretty print the profiling information.

To define a timer:

```
tc = TimerCollection()
a = tc.timer('my timer')
```

This will add 'my timer' to the list of Timers managed by the TimerCollection. Subsequent calls to timer() will return the same Timer object.

To start the timer:

```
a.begin()
```

and to end it:

```
a.end()
```

For best results, the block of code timed should be large enough to offset the overhead of the timer class method calls.

tc.report() prints out a summary of the timing.

report()
Generate a timing summary report

timer(*name*)
Create a timer with the given name. If one with that name already exists, then we return that timer.

Parameters

name [str] Name of the timer

Returns

out [Timer object] A timer object corresponding to the name.

25.24.6 util.runparams module

basic syntax of the parameter file is:

```
# simple parameter file

[driver]
nsteps = 100         ; comment
max_time = 0.25

[riemann]
tol = 1.e-10
max_iter = 10

[io]
basename = myfile_
```

The recommended way to use this is for the code to have a master list of parameters and their defaults (e.g. _defaults), and then the user can override these defaults at runtime through an inputs file. These two files have the same format.

The calling sequence would then be:

```
rp = RuntimeParameters()
rp.load_params("_defaults")
rp.load_params("inputs")
```

The parser will determine what datatype the parameter is (string, integer, float), and store it in a RuntimeParameters object. If a parameter that already exists is encountered a second time (e.g., there is a default value in _defaults and the user specifies a new value in inputs), then the second instance replaces the first.

Runtime parameters can then be accessed via any module through the get_param method:

```
tol = rp.get_param('riemann.tol')
```

If the optional flag no_new=1 is set, then the load_params function will not define any new parameters, but only overwrite existing ones. This is useful for reading in an inputs file that overrides previously read default values.

class util.runparams.**RuntimeParameters**
Bases: object

command_line_params(*cmd_strings*)
finds dictionary pairs from a string that came from the commandline. Stores the parameters in only if they already exist.

we expect things in the string in the form: ["sec.opt=value", "sec.opt=value"]

with each opt an element in the list

Parameters

cmd_strings [list] The list of strings containing runtime parameter pairs

get_param(*key*)
returns the value of the runtime parameter corresponding to the input key

load_params(*pfile*, *no_new=0*)
Reads line from file and makes dictionary pairs from the data to store.

Parameters

file [str] The name of the file to parse

no_new [int, optional] If no_new = 1, then we don't add any new paramters to the dictionary of runtime parameters, but instead just override the values of existing ones.

print_all_params()
Print out all runtime parameters and their values

print_paramfile()
Create a file, inputs.auto, that has the structure of a pyro inputs file, with all known parameters and values

print_sphinx_tables(*outfile='params-sphinx.inc'*)
Output Sphinx-formatted tables for inclusion in the documentation. The table columns will be: param, default, description.

print_unused_params()
Print out the list of parameters that were defined by never used

write_params(*f*)
Write the runtime parameters to an HDF5 file. Here, f is the h5py file object

util.runparams.**is_float**(*string*)
is the given string a float?

util.runparams.**is_int**(*string*)
is the given string an interger?

CHAPTER 26

References

CHAPTER 27

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

a

c

d

e

i

l

m

p

s

t

u

Index

A

B

F

G

I

J

L

M

N

P

R

S

T