Clawpack Documentation

Release 5.2.2

The Clawpack Development Team

November 30, 2014

I Full Table of Contents 3

Part I

Full Table of Contents

CHAPTER

ONE

OVERVIEW AND GETTING STARTED

1.1 About this software

Clawpack stands for "Conservation Laws Package" and was initially developed for linear and nonlinear hyperbolic systems of conservation laws, with a focus on implementing high-resolution Godunov type methods using limiters in a general framework applicable to many applications. These finite volume methods require a "Riemann solver" to resolve the jump discontinuity at the interface between two grid cells into waves propagating into the neighboring cells. The formulation used in Clawpack allows easy extension to the solution of hyperbolic problems that are not in conservation form.

See *wp_algorithms* for a brief description of the finite volume methods used in Clawpack and *riemann* for a description of the subroutine(s) needed to specify the hyperbolic equation being solved.

Adaptive mesh refinement is included, see *AMRClaw* (page 70), and routines specialized to depth-averaged geophysical flows can be found in *GeoClaw* (page 86).

The *Pyclaw* (page 113) software provides a more pythonic interface and parallelism that scales to tens of thousands of cores.

New users may wish to read *Which Clawpack solver should I use?* (page 15) before starting.

The "wave propagation" algorithms implemented in Clawpack are discribed in detail in the book *Finite Volume Methods for Hyperbolic Problems* [LeVeque-FVMHP] (page 292). Virtually all of the figures in this book were generated using Clawpack (version 4.3). See *Examples from the book FVMHP* (page 289) for a list of available examples with pointers to the codes and resulting plots.

See the *Bibliography* (page 289) for some pointers to papers describing Clawpack and the algorithms used in more detail.

1.1.1 License

Clawpack is distributed under the terms of the Berkeley Software Distribution (BSD) license.

See *License* (page 13) for details.

1.1.2 Authors

Many people have contributed to the development of Clawpack since its inception in 1994.

Major algorithmic and software design contributions have been made by the following individuals:

- Randall J. LeVeque, University of Washington, @rjleveque (https://github.com/rjleveque/).
- Marsha Berger, Courant Institute, NYU, @mjberger (https://github.com/mjberger/).

- Jan Olav Langseth, Norwegian Defence Research Establishment.
- David George, USGS Cascades Volcano Observatory, @dlgeorge (https://github.com/dlgeorge/).
- David Ketcheson, KAUST @ketch (https://github.com/ketch/).
- Kyle Mandli, UT-Austin, @mandli (https://github.com/mandli/).

Other major contributors include:

- Aron Ahmadia, @ahmadia (https://github.com/ahmadia/).
- Amal Alghamdi, @amal-ghamdi (https://github.com/amal-ghamdi/).
- Peter Blossey.
- Donna Calhoun, @donnaboise (https://github.com/donnaboise/).
- Ondřej Čertík, @certik (https://github.com/certik/).
- Grady Lemoine, @gradylemoine (https://github.com/gradylemoine/).
- Sorin Mitran.
- Matteo Parsani, @mparsani (https://github.com/mparsani/).
- Andy Terrel, @aterrel (https://github.com/aterrel/).

Numerous students and other users have also contributed towards this software, by finding bugs, suggesting improvements, and exploring its use on new applications. Thank you!

1.1.3 Citing this work

If you use Clawpack in publications, please cite the following:

```
@misc{clawpack,
    title={Clawpack software},
    author={{Clawpack Development Team}},
    url={http://www.clawpack.org},
    note={Version x.y},
    year={2014}}
```

Please fill in the version number that you used, and its year.

Please also cite at least one of the following regarding the algorithms used in Clawpack (click the links for bibtex citations):

- Classic algorithms in 1d and 2d: [LeVeque97] (page 292), [LeVeque-FVMHP] (page 292)
- 3d classic algorithms: [LangsethLeVeque00] (page 292)
- AMR: [BergerLeVeque98] (page 291)
- f-wave algorithms: [BaleLevMitRoss02] (page 291)
- GeoClaw: [BergerGeorgeLeVequeMandli11] (page 291), [LeVequeGeorgeBerger] (page 292)
- High-order method-of-lines algorithms (SharpClaw): [KetParLev13] (page 293)
- PyClaw: [KetchesonMandliEtAl] (page 293)

1.1.4 Funding

Development of this software has been supported in part by

- NSF Grants DMS-8657319, DMS-9204329, DMS-9303404, DMS-9505021, DMS-96226645, DMS-9803442, DMS-0106511, CMS-0245206, DMS-0609661, DMS-0914942, DMS-1216732.
- DOE Grants DE-FG06-93ER25181, DE-FG03-96ER25292, DE-FG02-88ER25053, DE-FG02-92ER25139, DE-FG03-00ER2592, DE-FC02-01ER25474
- AFOSR grant F49620-94-0132,
- NIH grant 5R01AR53652-2,
- ONR grant N00014-09-1-0649
- The Norwegian Research Council (NFR) through the program no. 101039/420.
- The Scientific Computing Division at the National Center for Atmospheric Research (NCAR).
- The Boeing Professorship and the Founders Term Professorship in the Department of Applied Mathematics, University of Washington.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these agencies.

1.2 Recent changes — release notes

1.2.1 Clawpack 5.x

- Clawpack 5.0 has significant changes from past versions. See *clawpack5*.
- For changes in PyClaw, see the PyClaw changelog (https://github.com/clawpack/pyclaw/blob/master/CHANGES.md).
- *release_5_0_0* – January 8, 2014
- *release_5_1_0* – March 2, 2014
- *release_5_2_0* – July 18, 2014
- *release_5_2_1* – October 2, 2014
- *release_5_2_2* – October 28, 2014
- *changes_to_master*

1.2.2 Clawpack 4.x

For documentation and recent changes to the Clawpack 4.x version, please see Clawpack 4.x documentation (http://depts.washington.edu/clawpack/users-4.x/index.html)

1.3 Installation instructions

Contents

Please register (http://depts.washington.edu/clawpack/register/index.html) if you have not already done so. This is purely optional, but is useful in helping us track the extent of usage.

Note that any of these installations also includes *visclaw* for plotting.

See also:

- *Troubleshooting: Installation* (page 13)
- *Which Clawpack solver should I use?* (page 15)

1.3.1 Install only PyClaw (serial)

If you wish to install just PyClaw, everything is handled by pip:

```
pip install clawpack
```

Next go to *Running an example* (page 11).

1.3.2 Install all Clawpack packages

First, download a tar file of the latest release:

- https://github.com/clawpack/clawpack/releases/download/v5.2.2/clawpack-5.2.2.tar.gz
- *Previous versions of Clawpack* (page 24)
- *Recent changes — release notes* (page 7)

See *Clawpack components* (page 16) for a list of what's included in this tar file.

Save this tar file in the directory where you want the top level of the clawpack tree to reside. Then untar using the command:

```
tar -xzvf clawpack-5.2.2.tar.gz
```

Then move into the top level directory:

```
cd clawpack-5.2.2
```

Next install the Python components of Clawpack:

```
python setup.py install
```

This will compile a lot of Fortran code using *f2py* and will produce a lot of output, so you might want to redirect the output, e.g.

```
python setup.py install > install_output.txt
```

If you get compilation errors in this step, you can still use the Classic, AMRClaw, and GeoClaw; see *Install other packages without compiling PyClaw* (page 9).

If you only plan to use PyClaw, jump to *Running an example* (page 11). If you plan to use Classic, AMRClaw, or GeoClaw continue with *Set environment variables* (page 9).

Set environment variables

To use the Fortran versions of Clawpack you will need to set the environment variable *CLAW* to point to the top level of clawpack tree (there is no need to perform this step if you will only use PyClaw). In the bash shell these can be set via:

```
export CLAW=/full/path/to/top/level
```

You also need to set *FC* to point to the desired Fortran compiler, e.g.:

```
export FC=gfortran   # or other preferred Fortran compiler
```

Consider putting the two commands above in a file that is executed every time you open a new shell or terminal window. On Linux machines with the bash shell this is generally the file *.bashrc* in your home directory. On a Mac it may be called *.bash_profile*.

If your environment variable *CLAW* is properly set, the command

```
ls $CLAW
```

should list the top level directory, and report for example:

```
README.md       riemann/        pyclaw/
amrclaw/        setup.py        clawutil/
geoclaw/        visclaw/        classic/
```

Next go to *Running an example* (page 11).

1.3.3 Install other packages without compiling PyClaw

If you get errors in the compilation step when using *pip install* or *python setup.py install*, check *Troubleshooting: Installation* (page 13). If your problem is not addressed there, please let us know (claw-users@googlegroups.com) or raise an issue (https://github.com/clawpack/clawpack/issues). You can still use the Fortran codes (AMRClaw, GeoClaw, and Classic) by doing the following.

First, download a tarfile of the latest release as described in *Install all Clawpack packages* (page 8).

Next *Set environment variables* (page 9). You must then also set your PYTHONPATH manually:

```
export PYTHONPATH=$CLAW:$PYTHONPATH
```

Then you should be able to do:

```
cd $CLAW   # assuming this environment variable was properly set
python setup.py symlink-only
```

This will create some symbolic links in the *$CLAW/clawpack* subdirectory of your top level Clawpack directory, but does not compile code or put anything in your site-packages. In Python you should now be able to do the following, for example:

```
>>> from clawpack import visclaw
```

If not then either your *$PYTHONPATH* environment variable is not set properly or the required symbolic links were not created.

Next go to *Running an example* (page 11).

1.3.4 Install only PyClaw (for running in parallel)

First, install PyClaw as explained above. Then see the install instructions for *Running in parallel* (page 183).

Alternatively, you may use the following shell scripts (assembled by Damian San Roman) to install everything:

- Linux machine or Beowulf Cluster: https://gist.github.com/sanromd/9112666
- Mac OS X: https://gist.github.com/sanromd/10374134

1.3.5 Install the latest development version

The development version of Clawpack can be obtained by cloning https://github.com/clawpack. This is advised for those who want to help develop Clawpack or to have the most recent bleeding edge version. See *Installation instructions for developers* (page 277) for instructions.

1.3.6 Running Clawpack on a VM or in the Cloud

Virtual Machine. An alternative to installing the *Installation Prerequisites* (page 10) and Clawpack itself is to use the *Clawpack Virtual Machine* (page 16).

Cloud Computing.

- *Pyclaw* (page 113) can be installed and run in the cloud for free on http://wakari.io or http://cloud.sagemath.com; see *cloud*.
- All of Clawpack can be run on AWS using the *Amazon Web Services EC2 Clawpack AMI* (page 17).

1.3.7 Installation Prerequisites

Operating system:

- Linux
- Mac OS X (you need to have the Xcode developer tools (http://developer.apple.com/technologies/tools/xcode.html) installed in order to have "make" working)

Much of Clawpack will work under Windows using Cygwin, but this is not officially supported.

Fortran:

- gfortran (http://gcc.gnu.org/wiki/GFortran) or another F90 compiler

See *Fortran Compilers* (page 46) for more about which compilers work well with Clawpack. For Mac OSX, see hpc.sourceforge.net (http://hpc.sourceforge.net/) for some installation options.

Python:

- Python Version 2.7 or above (but **not** 3.0 or above, which is not backwards compatible)
- NumPy (http://www.numpy.org/) (for PyClaw/VisClaw)

- matplotlib (http://matplotlib.org/) (for PyClaw/VisClaw)

See *Python Hints* (page 52) for information on installing the required modules and to get started using Python if you are not familiar with it.

1.3.8 Testing your installation

PyClaw

If you downloaded Clawpack manually, you can test your PyClaw installation as follows (starting from your *clawpack* directory):

```
cd pyclaw
nosetests
```

This should return 'OK'.

Classic

As a first test of the Fortran code, try the following:

```
cd $CLAW/classic/tests
make tests
```

This will run several tests and compare a few numbers from the solution with archived results. The tests should run in a few seconds.

There are similar *tests* subdirectories of *$CLAW/amrclaw* and *$CLAW/geoclaw* to do quick tests of these codes.

See also *Testing your installation* (page 43).

1.4 Running an example

Many examples of Clawpack simulations can be seen in the *Galleries of all Clawpack applications* (page 25).

1.4.1 PyClaw

To run an example and plot the results using PyClaw, simply do the following (starting from your *clawpack* directory):

```
cd pyclaw/examples/euler_2d
python shock_bubble_interaction.py iplot=1
```

That's it. For next steps with PyClaw, see *PyClaw Basics* (page 113).

1.4.2 Classic

First ensure that you have *Set environment variables* (page 9). A simple 1-dimensional acoustics equations can be solved using the code in $CLAW/classic/examples/acoustics_1d_example1, as illustrated in *Gallery of Classic and AMRClaw applications* (page 25).

Move to this directory via:

```
cd $CLAW/classic/examples/acoustics_1d_example1
```

You can try the following test in this directory, or you may want to first make a copy of it (see the instructions in *Copying an existing example* (page 42)).

The Makefiles are set up to do dependency checking so that in many application directories you can simply type:

```
$ make .plots
```

and the Fortran code will be compiled, data files created, the code run, and the results plotted automatically, resulting in a set of webpages showing the results.

However, if this is your first attempt to run a code, it is useful to go through these steps one at a time, both to understand the steps and so that any problems with your installation can be properly identified.

You might want to start by examining the Makefile. This sets a number of variables, which at some point you might need to modify for other examples, see *Clawpack Makefiles* (page 54) for more about this. At the bottom of the Makefile is an *include* statement that points to a common Makefile that is used by most applications, and where all the details of the make process can be found.

To compile the code, type:

```
$ make .exe
```

If this gives an error, see *Trouble running "make .exe"* (page 14).

This should compile the example code (after first compiling the required library routines) and produce an executable named *xclaw* in this directory.

Before running the code, it is necessary to also create a set of data files that are read in by the Fortran code. This can be done via:

```
$ make .data
```

If this gives an error, see *Trouble running "make .data"* (page 14).

This uses the Python code in *setrun.py* to create data files that have the form **.data*.

Once the executable and the data files all exist, we can run the code. The recommended way to do this is to type:

```
$ make .output
```

If this gives an error, see *Trouble running "make .output"* (page 14).

Before running the code a subdirectory *_output* is created and the output of the code (often a large number of files) is directed to this subdirectory. This is convenient if you want to do several runs with different parameter values and keep the results organized. After the code has run you can rename the subdirectory, or you can modify the variable *OUTDIR* in the Makefile to direct results to a different directory. See *Clawpack Makefiles* (page 54) for more details. Copies of all the data files are also placed in the output directory for future reference.

Plotting the results. Once the code has run and the files listed above have been created, there are several options for plotting the results.

To try the Python tools, type:

```
$ make .plots
```

If this gives an error, see *Trouble running "make .plots"* (page 15).

If this works, it will create a subdirectory named *_plots* that contains a number of image files (the **.png* files) and a set of html files that can be used to view the results from a web browser. See *plotting_makeplots* for more details.

An alternative is to view the plots from an interactive Python session, as described in the section *Interactive plotting with Iplotclaw* (page 248).

If you wish to use Matlab instead, see *Plotting using Matlab* (page 268).

Other visualization packages could also be used to display the results, but you will need to figure out how to read in the data. See *fortfiles* for information about the format of the files produced by Clawpack.

1.5 License

Clawpack is distributed under the terms of the Berkeley Software Distribution (BSD) license.

See http://www.opensource.org/licenses/bsd-license.php for more details.

Copyright (c) 1994–2014, Randall J. LeVeque and others. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of the University of Washington nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

1.6 Troubleshooting

1.6.1 Troubleshooting: Installation

Setting the Fortran compiler to be used by f2py (pip)

When executing `pip install clawpack` or `python setup.py install`, if you get an error like:

```
error: f90 not supported by GnuFCompiler
```

then f2py is trying to use your f77 compiler. This may happen even if you also have an f90 compiler like gfortran installed. In this case, `pip install` will not work; you should download a tarball or clone the code from Github. Then, in order to see the compilers detected by f2py, run:

```
python setup.py config_fc --help-fcompiler
```

Then to install using a different compiler, do e.g.:

```
python setup.py config_fc --fcompiler=gfortran install
```

You may replace `gfortran` with the compiler you wish to use.

1.6.2 Trouble running "make .exe"

If the code does not compile, check the following:

- Make sure your environment variable *CLAW* is set properly:

  ```
  $ printenv CLAW
  ```

  to print the value. The Makefiles use this variable to find the common Makefile and library routines.

  If you get the error message:

  ```
  Makefile:154: /clawutil/src/Makefile.common: No such file or directory
  ```

  then *CLAW* is not set properly. It is looking for the file *$CLAW/clawutil/src/Makefile.common* and if *CLAW* is not set, the path will be missing.

- Make sure your environment variable *FC* is set properly. This should be set to the command used to invoke the Fortran compiler, e.g. *gfortran*.

  If you get an error like:

  ```
  make[1]: gfortran: No such file or directory
  ```

  then the gfortran compiler is not being found.

1.6.3 Trouble running "make .data"

If there are errors in the *setrun* function (usually defined in *setrun.py*) then the these may show up when you try to "make .data" since this function must be executed.

See *Specifying classic run-time parameters in setrun.py* (page 56) for information about the setrun function.

1.6.4 Trouble running "make .output"

If you want to re-run the code and you get:

```
$ make .output
make: '.output' is up to date.
```

then you can force it to run again by removing the file *.output*:

```
$ rm -f .output
$ make .output
```

This happens for example if you changed something that you know will affect the output but that isn't in the Makefile's set of dependencies.

You can also do

    $ make output

(with no dot before `output`) to run the code without checking dependencies. See *Clawpack Makefiles* (page 54) for more details and warnings.

1.6.5 Trouble running "make .plots"

The Python plotting routines require *NumPy* and *matplotlib*. See *Python Hints* (page 52) for information on installing these.

If there are errors in the *setplot* function (usually defined in *setplot.py*) then the these may show up when you try to "make .plots" since this function must be executed. See *Using setplot.py to specify the desired plots* (page 251).

You can also do

```
$ make plots
```

(with no dot before `plots`) to plot the output without checking dependencies. This will never run the code, it will only attempt to plot the output files found in *_output* directory (or wherever the *OUTDIR* variable in the *Makefile* points).

See *Clawpack Makefiles* (page 54) for more details and warnings.

1.7 Which Clawpack solver should I use?

Clawpack includes a number of related hyperbolic PDE solvers:

- Classic
- *AMRClaw* (page 70)
- *GeoClaw* (page 86)
- *Pyclaw* (page 113)

All of them are built on common algorithmic ideas, make use of the same set of Riemann solvers, and can be used with VisClaw for visualization. If you're not sure which solver to use, here you will find the main differences between them.

1.7.1 Installation and user interface

The AMRClaw, GeoClaw, and Classic solvers are Fortran-based packages and rely on Makefiles and environment variables. Problems are specified partially through Python scripts at run time (*setrun.py*) and partially through custom Fortran code at compile time (to set initial conditions, for instance).

With PyClaw, problems are specified entirely at run time through Python script files, or interactively (e.g., in IPython). Typically, the user does not need to write any Fortran code (though custom routines can be written in Fortran when necessary for performance reasons). PyClaw uses much of the same library of Fortran code, but that code is compiled during installation so that it can be imported dynamically within Python programs.

1.7.2 Algorithmic differences

All of the Clawpack solvers include the *classic* algorithms described in [LeVeque-FVMHP] (page 292); if you only require those, it's easiest to use Classic or *Pyclaw* (page 113). Most of the packages contain additional algorithms:

- **AMRClaw** includes block-structured adaptive mesh refinement that allows one to use a non-uniform grid that changes in time and uses smaller grid cells in regions with fine structure or where high accuracy is required.
- **GeoClaw** Includes the AMR capabilities of AMRClaw and also has a number of special routines and algorithms for handling geophysical problems, including special well-balanced, positivity-preserving shallow water solvers.
- **PyClaw** includes the *high-order WENO-RK algorithms of SharpClaw* (page 209), described in [KetParLev13] (page 293).

1.7.3 Parallel computing

- **AMRClaw**, **GeoClaw**, and **Classic** can be run in parallel using shared memory via OpenMP.
- **PyClaw** can be run in parallel on distributed-memory machines using MPI (through PETSc) and has been shown to scale to tens of thousands of cores.

1.8 Clawpack components

Clawpack is developed using the git (http://git-scm.com/) version control system and all the source code is openly available via the Clawpack GitHub Organization (https://github.com/organizations/clawpack).

The code is organized in several independent git repositories. One of these is the clawpack/clawpack (https://github.com/clawpack/clawpack) super-repository that is used to coordinate versions between other repositories. If you are interested in cloning the code directly from GitHub and/or helping develop Clawpack, see *Developers' Guide* (page 276).

If you download a tar file of Clawpack, as described in *Install all Clawpack packages* (page 8), then you will obtain a top level directory that has the following subdirectories:

- *classic* (Classic single-grid Fortran code)
- *amrclaw* (*AMRClaw* (page 70), AMR version of Fortran code)
- *riemann* (Riemann solvers, in Fortran, also used by PyClaw)
- *geoclaw* (GeoClaw for geophysical flows)
- *clawutil* (Utility functions, Makefile.common used in multiple repositories)
- *pyclaw* (Python version that includes SharpClaw and PETSc parallelization)
- *visclaw* (Python graphics and visualization tools)

These correspond to individual GitHub repositories.

1.8.1 Other repositories

Other repositories in the Clawpack GitHub Organization (https://github.com/organizations/clawpack) may be of interest to some users:

- *apps* contains additional applications, see *Clawpack Applications repository* (page 41).
- *doc* contains sphinx (http://sphinx.pocoo.org/) input files for the Clawpack documentation.
- *clawpack.github.com* contains the html files that appear on the web.
- *clawpack-4.x* contains the latest version of Clawpack 4.x, the legacy code.

1.9 Clawpack Virtual Machine

There is no Clawpack 5.0 VM yet.

See The Clawpack 4.6 VM documentation (http://depts.washington.edu/clawpack/users-4.x/vm.html) for now. This VM should work with Clawpack 5.0 if you install it following the *Installation instructions* (page 7).

1.10 Amazon Web Services EC2 Clawpack AMI

Warning: This has not been updated for Clawpack-5 yet, but it should still work if you start up an instance as described below and then install Clawpack-5 by following the *Installation instructions* (page 7).

To run Clawpack in the Cloud using Amazon Web Services Elastic Cloud Computing (EC2), first sign up for an account. Note that you can get 750 hours free micro instance usage (which may be sufficient for many things) in the free usage tier (http://aws.amazon.com/free/).

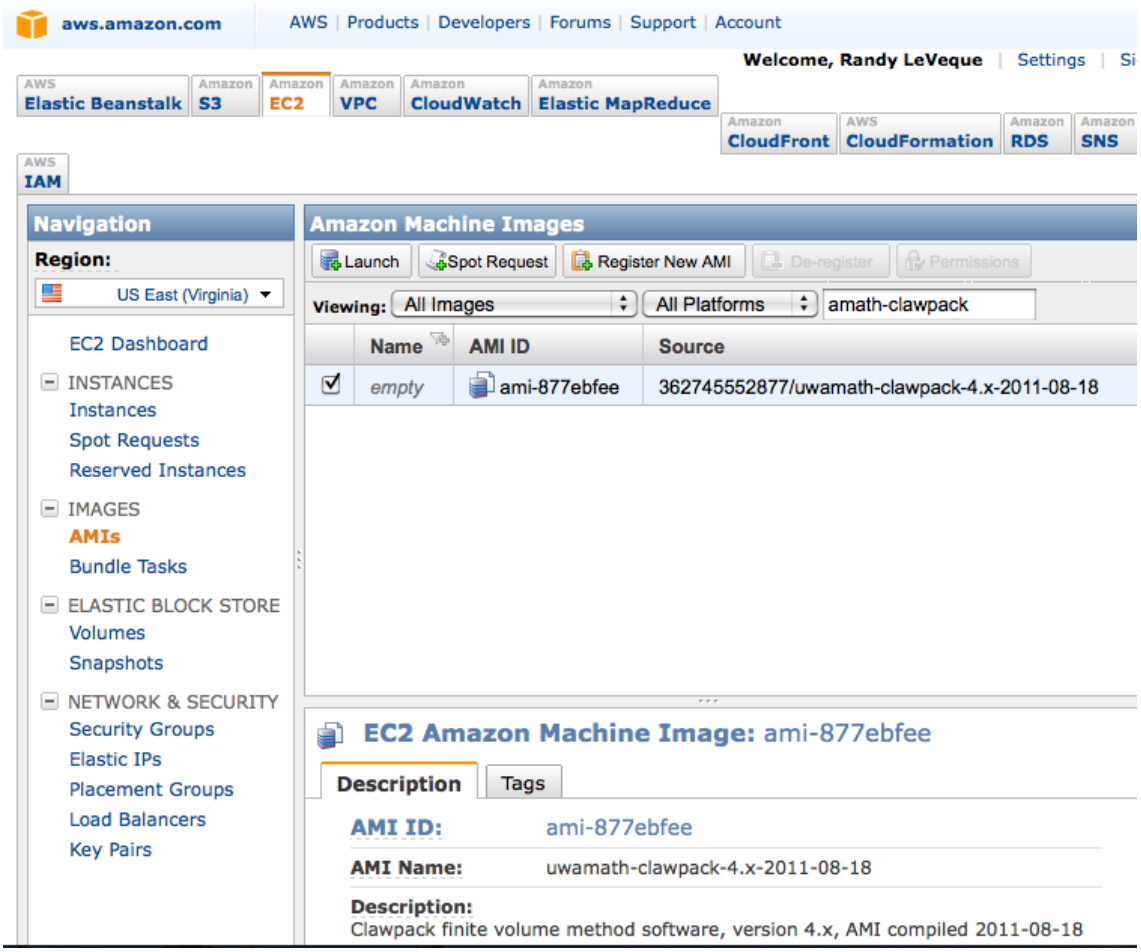
For general information and a guide to getting started:

- UW eScience information on AWS (http://escience.washington.edu/get-help-now/get-started-amazon-web-services).
- Getting started with EC2 (http://docs.amazonwebservices.com/AWSEC2/latest/GettingStartedGuide/), with tutorial to lead you through an example (a similar tutorial geared to Clawpack is included below).
- EC2 FAQ (http://aws.amazon.com/ec2/faqs).
- Pricing (http://aws.amazon.com/ec2/#pricing). Note: you are charged per hour for hours (or fraction thereof) that your instance is in *running* mode, regardless of whether the CPU is being used.

1.10.1 Finding the Clawpack AMI

Once you have an AWS account, sign in to the management console (https://console.aws.amazon.com/ec2/) and click on the EC2 tab, and then select Region US East (which has cheaper rates) and click on *AMIs* on the menu to the left.

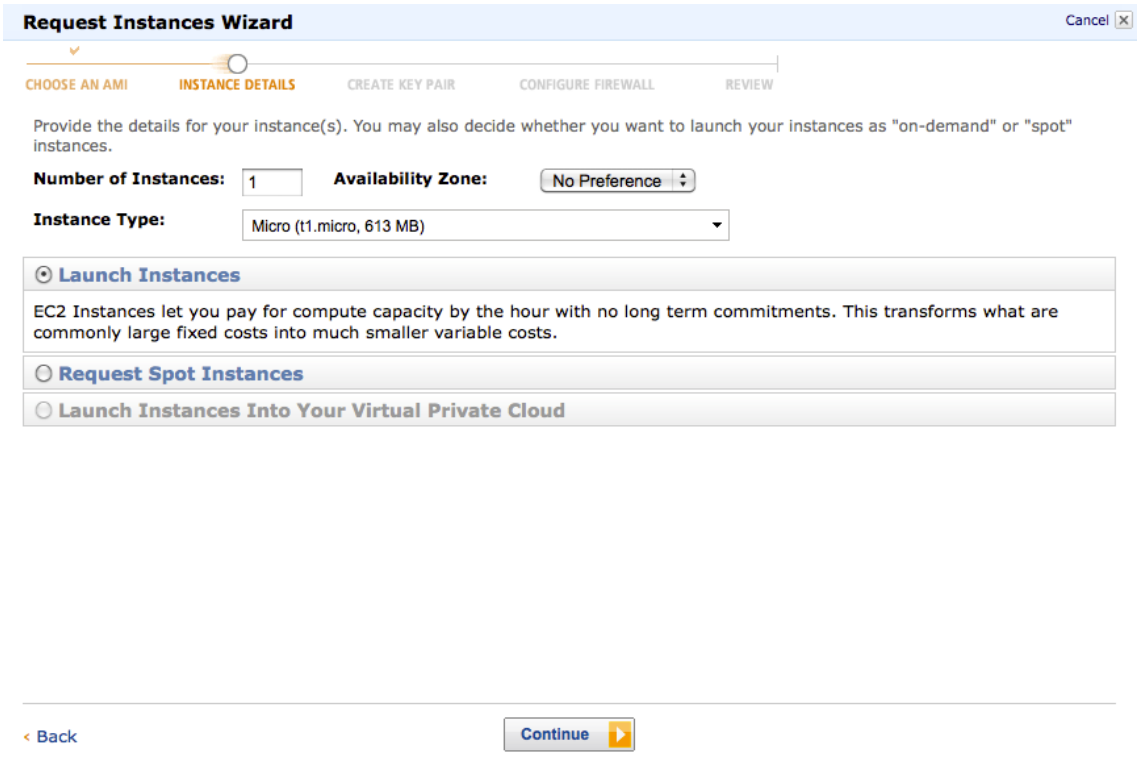
Change *Viewing:* to *All Images* and *All Platforms* and then *after* it has finished loading the database start typing *uwamath-clawpack* in the search bar. You should find at least one AMI, as shown in this screen shapshot:



1.10.2 Launching an instance

Select the Clawpack image and then click on the *Launch* button on this page to start launching an instance based on this AMI. This means a virtual machine will be started for you, initialized with this disk image (which is a Ubuntu linux distribution with Clawpack and its dependencies).

This should give a popup page that looks like this:



Here you can select what sort of instance you wish to start (larger instances cost more per hour).

Click *Continue* on the next few screens and eventually you get to one that looks like:

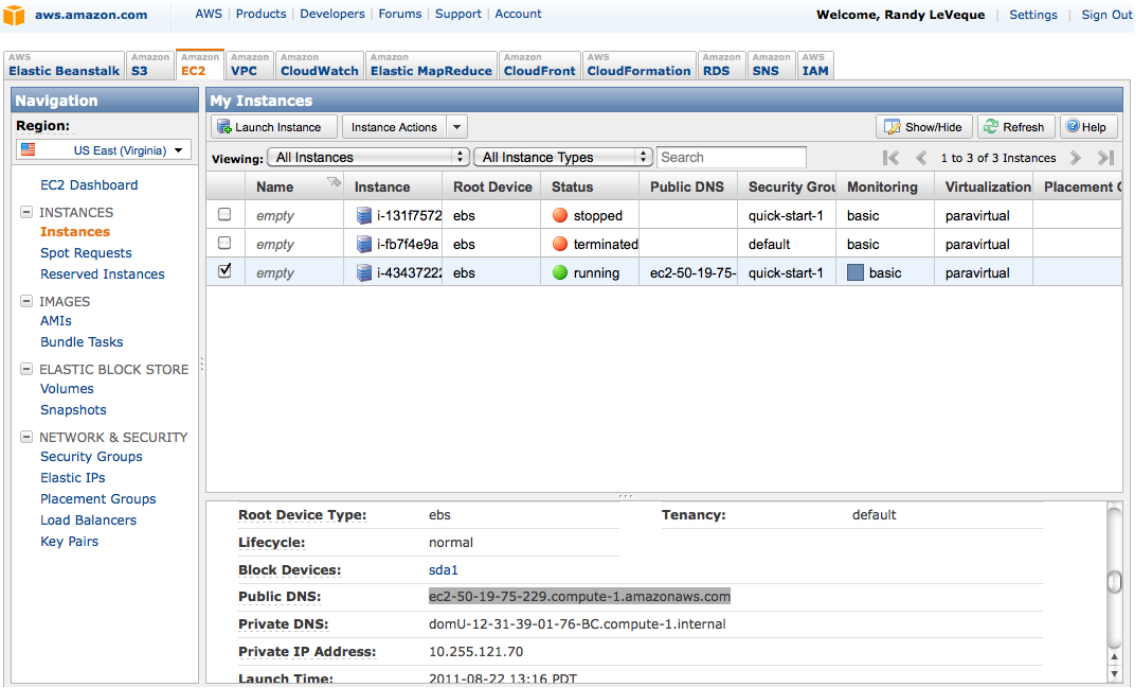
If you don't already have a key pair, create a new one and then select this key pair here.

Click *Continue* and you will get a screen to set Security Groups. Select the *quick-start-1* option. On the next screen click *Launch*.

1.10.3 Logging on to your instance

Click *Close* on the next page to go back to the Management Console. Click on *Instances* on the left menu and you should see a list of instance you have created, in your case only one. If the status is not yet *running* then wait until it is (click on the *Refresh* button if necessary).

Click on the instance and information about it should appear at the bottom of the screen. Scroll down until you find the *Public DNS* information, highlighted on the screenshot below:



Go into the directory where your key pair is stored, in a file with a name like *rjlkey.pem* and you should be able to *ssh* into your instance using the name of the public DNS, with format like:

```
$ ssh -i KEYPAIR-FILE  ubuntu@DNS
```

where KEYPAIR-FILE and DNS must be replaced by the appropriate things, e.g. for the above example:

```
$ ssh -i rjlkey.pem ubuntu@ec2-50-19-75-229.compute-1.amazonaws.com
```

Note:

- You must include *-i keypair-file*
- You must log in as user ubuntu.

1.10.4 Using Clawpack

Once you have logged into your instance, you are on Ubuntu Linux that has software needed for Clawpack pre-installed, including:

- gfortran

- Ipython, numpy, scipy, matplotlib
- make
- git
- netcdf
- apache web server

Other software is easily installed using *apt-get install*.

The current development version of Clawpack is installed in */claw/clawpack-4.x*. If you want to use this version, you might want to:

```
$ cd /claw/clawpack-4.x
$ git fetch origin   # bring over any recent changes
$ git merge origin/master  # merge them in
$ python python/make_libs.py  # compile libraries
```

The *$CLAW* variable is set to point to this version of Clawpack (in the *.bashrc* file).

Of course you could instead download a tar file of Clawpack and install following the instructions at *Installation instructions* (page 7). At any rate, see that section for instructions on what to do next if you are new to Clawpack.

Warning: If you want to use Clawpack-5, instead follow the *Installation instructions* (page 7).

1.10.5 Viewing plots of results

If you run Clawpack on your instance then you will probably want to view the results. There are at least three possible approaches (see *Visclaw Plotting options* (page 247) for general information about plotting in Clawpack):

- If you are on a computer that supports X windows and you add the *-X* flag to your *ssh* command, then you should be able to plot interactively (see *Interactive plotting with Iplotclaw* (page 248)). Response may be pretty slow, however.
- If you create plots using

  ```
  $ make .plots
  ```

  then you will have a directory (named *_plots* by default) that contains *.png* figures and *.html* files for viewing them. You can tar this directory up and transfer it to your local machine using *sftp*, and then view locally.

  Note that the plot files are often **much** smaller than the Fortran output files in *_output*, and so much quicker to transfer.
- You can view the plots directly using a web browser as explained in the next section.

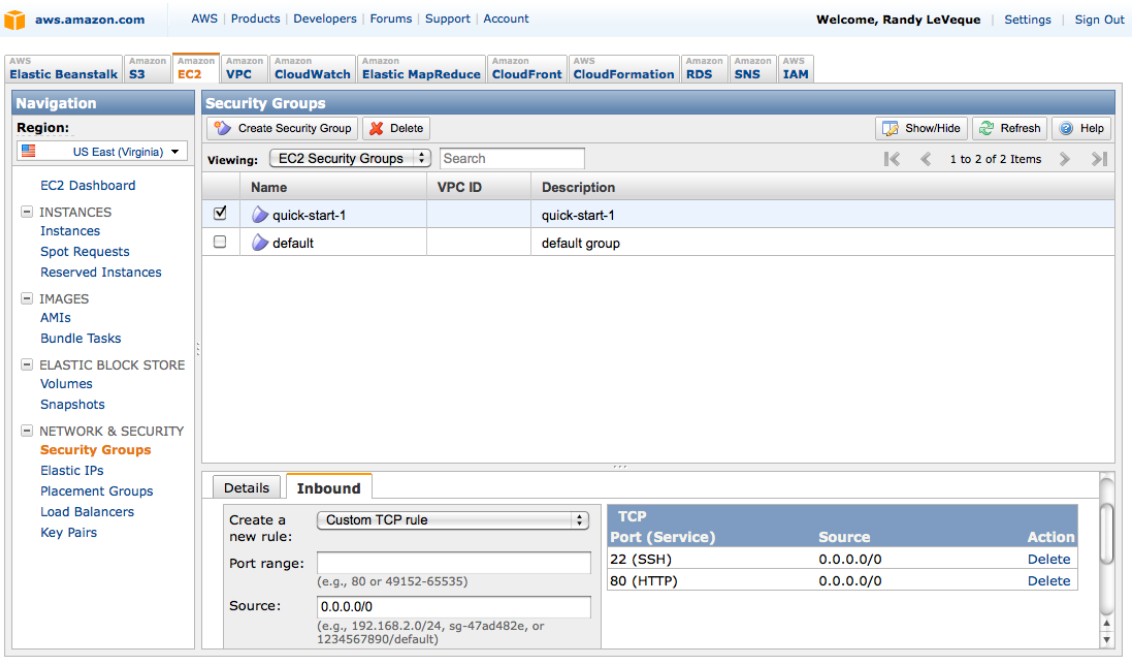
1.10.6 Viewing webpages directly from your instance

If you use

```
$ make .plots
```

to make a set of plot files and html files for viewing them, you can view them directly by opening a web browser to an appropriate path on your instance.

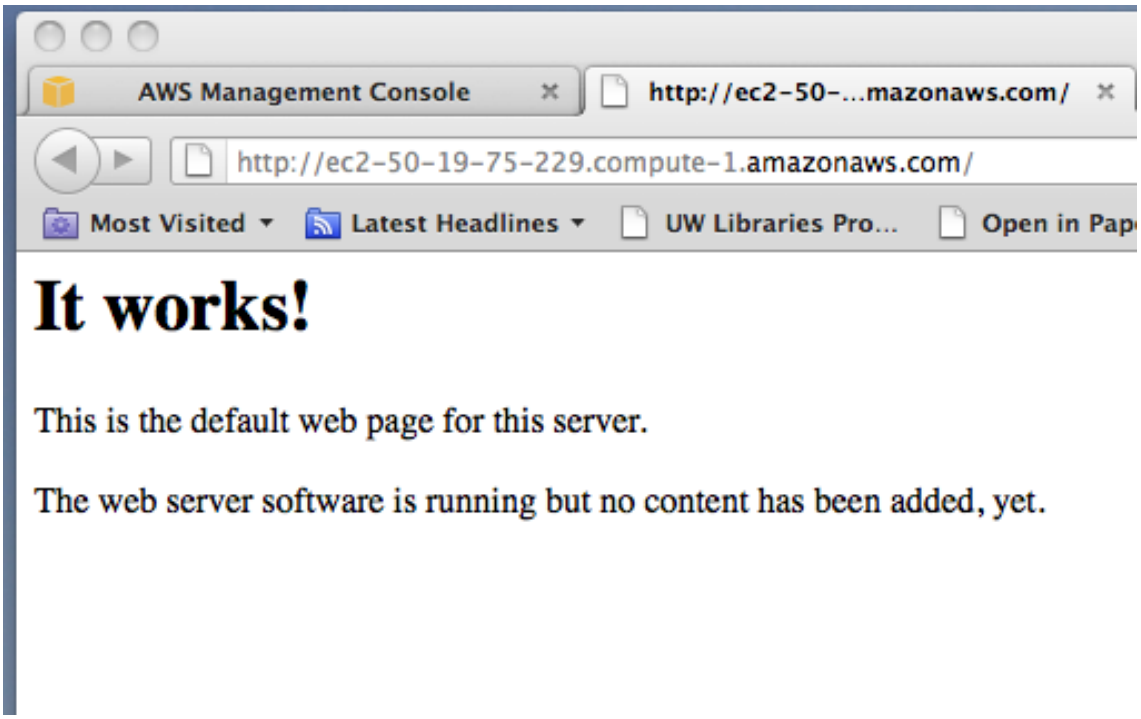
The apache webserver should already be running, but to allow people to view webpages you will need to adjust the security settings. Go back to the Management Console and click on *Security Groups* on the left menu. Select *quick-start-1* and then click on *Inbound*. You should see a list of ports that only lists 22 (SSH). You want to add port 80 (HTTP). Select HTTP from the drop-down menu that says *Custom TCP Rule* and type 80 for the *Port range*. Then click *Apply Rule Changes*. This should give something like the next screen shot:



Now you should be able to point your browser to *http://DNS* where *DNS* is replaced by the Public DNS name of your instance, the same as used for the *ssh* command. So for the example above, this would be

`'http://ec2-50-19-75-229.compute-1.amazonaws.com'`.

You should see this page:



The page being displayed can be found in */var/www/index.html* on your instance. Any files you want to be visible on the web should be in */var/www*, or it is sufficient to have a link from this directory to where they are located (created with the *ln -s* command in linux).

So, for example, if you do the following:

```
$ cd /var/www
$ ln -s /claw/clawpack-4.x/apps ./apps
```

Then you should be able to see the *apps* directory in your web browser, which would be at

```
http://ec2-50-19-75-229.compute-1.amazonaws.com/apps/
```

for the above example. You will have to replace the DNS with that of your instance.

If you want to expose all of your home directory to the web:

```
$ cd /var/www
$ ln -s /home/ubuntu ./home
```

1.10.7 Transferring files to/from your instance

You can use *scp* to transfer files between a running instance and the computer on which the ssh key is stored.

From your computer (not from the instance):

```
$ scp -i KEYPAIR-FILE FILE-TO-SEND ubuntu@DNS:REMOTE-DIRECTORY
```

where DNS is the public DNS of the instance and *REMOTE-DIRECTORY* is the path (relative to home directory) where you want the file to end up. You can leave off *:REMOTE-DIRECTORY* if you want it to end up in your home directory.

Going the other way, you can download a file from your instance to your own computer via:

```
$ scp -i KEYPAIR-FILE ubuntu@DNS:FILE-TO-GET .
```

to retrieve the file named *FILE-TO-GET* (which might include a path relative to the home directory) into the current directory.

1.10.8 Stopping your instance

Once you are done computing for the day, you will probably want to stop your instance so you won't be charged while it's sitting idle. You can do this by selecting the instance from the Management Console / Instances, and then select *Stop* from the *Instance Actions* menu.

You can restart it later and it will be in the same state you left it in. But note that it will probably have a new Public DNS!

1.10.9 Creating your own AMI

If you add additional software and want to save a disk image of your improved virtual machine (e.g. in order to launch additional images in the future to run multiple jobs at once), simply click on *Create Image (EBS AMI)* from the *Instance Actions* menu.

1.11 Previous versions of Clawpack

- Versions of Clawpack 5.0 can be found at https://github.com/clawpack/clawpack/releases/.
- The 4.6 version of Clawpack can be found here (http://depts.washington.edu/clawpack/download/downloadmenu.html).
- See *Developers' Guide* (page 276) for information on cloning from GitHub.

CHAPTER

TWO

EXAMPLES AND APPLICATIONS

2.1 Galleries of all Clawpack applications

See also *IPython notebook examples* (page 53).

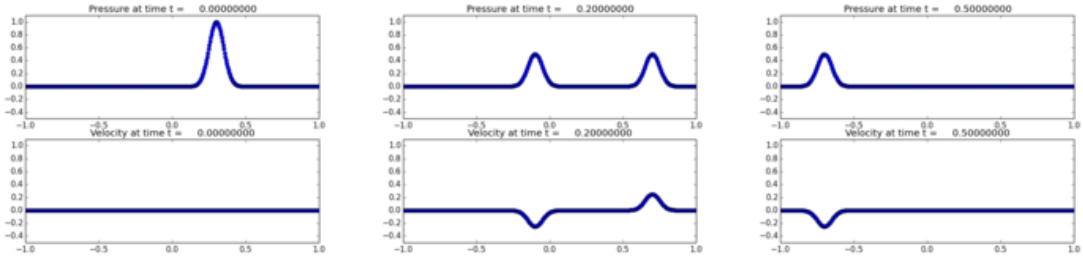
2.1.1 Gallery of Classic and AMRClaw applications

Contents

1-dimensional acoustics

Directory: '$CLAW/classic/examples/acoustics_1d_example1'

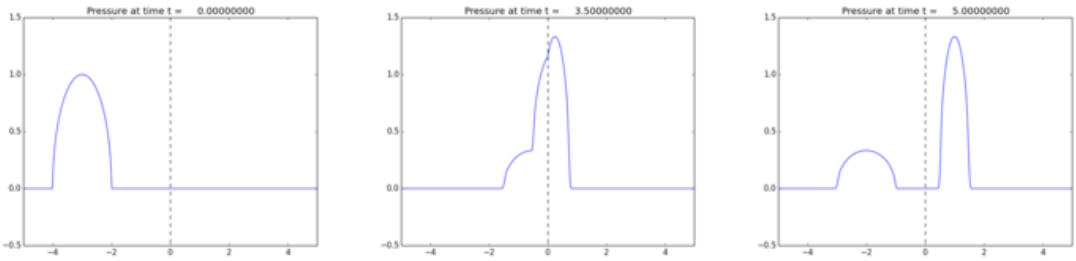
Acoustics equations with interface showing reflection and transmission.



README ... Plots

Directory: '$CLAW/classic/examples/acoustics_1d_heterogeneous'

Acoustics equations with interface showing reflection and transmission.

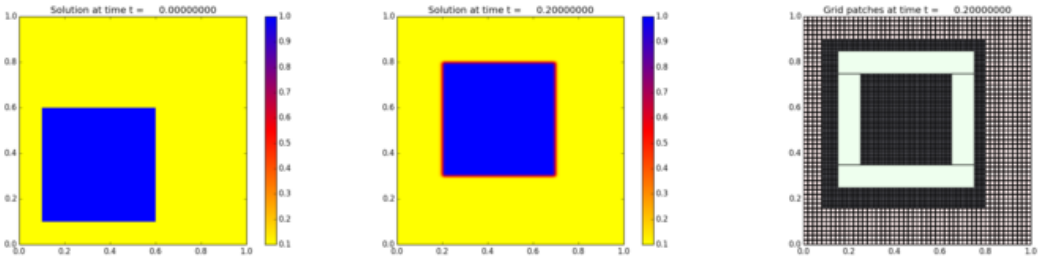


README ... Plots

2-dimensional advection

Directory: ‘$CLAW/amrclaw/examples/advection_2d_square‘

Advecting square with periodic boundary conditions.

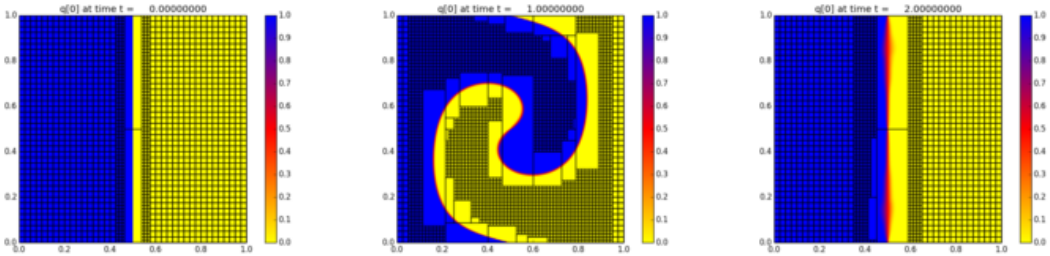


README ... Plots

2-dimensional variable-coefficient advection

Directory: ‘$CLAW/amrclaw/examples/advection_2d_swirl‘

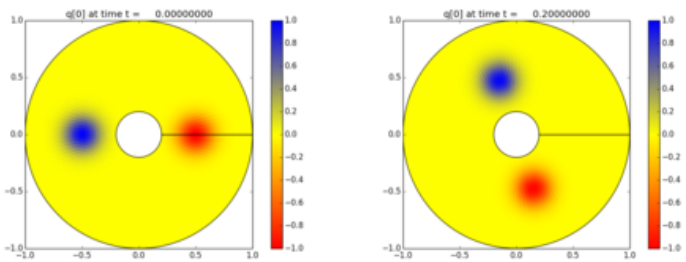
Advection with a swirling flow field with AMR.



README ... Plots

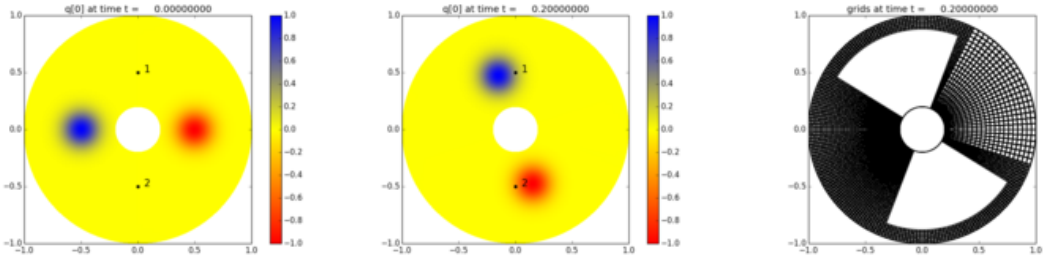
Directory: ‘$CLAW/classic/examples/advection_2d_annulus‘

Advection in an annular region.



README ... Plots **Directory: ‘$CLAW/amrclaw/examples/advection_2d_annulus‘**

Advection in an annular region with AMR.

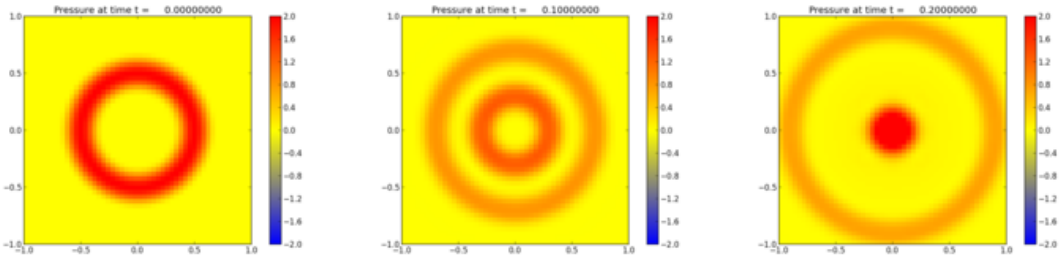


README ... Plots

2-dimensional acoustics

Directory: '$CLAW/classic/examples/acoustics_2d_radial'

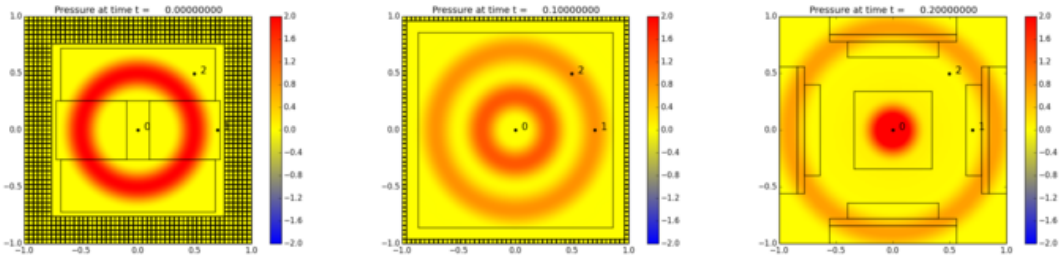
Expanding radial acoustic wave in a homogeneous medium.



README ... Plots

Directory: '$CLAW/amrclaw/examples/acoustics_2d_radial'

Expanding radial acoustic wave in a homogeneous medium with AMR.

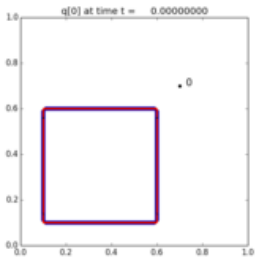


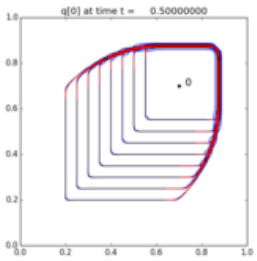
README ... Plots

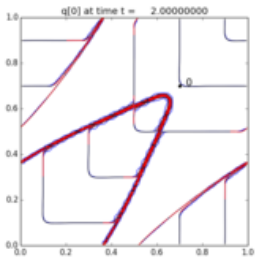
2-dimensional Burgers' equation

Directory: '$CLAW/amrclaw/examples/burgers_2d_square'

Burgers' equation $q_t + 0.5(q^2)_x + 0.5(q^2)_y = 0$ with square initial pulse and periodic boundary conditions.





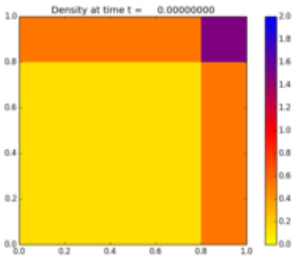


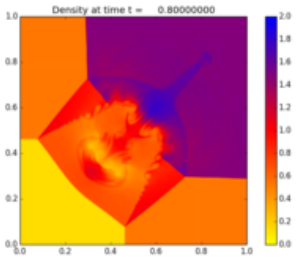
README ... Plots

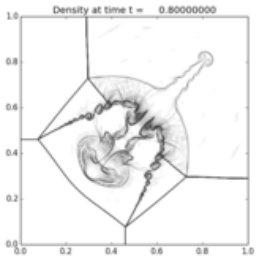
2-dimensional Euler equations

Directory: '$CLAW/amrclaw/examples/euler_2d_quadrants'

Euler equations with piecewise constant data in quadrants.







README ... Plots

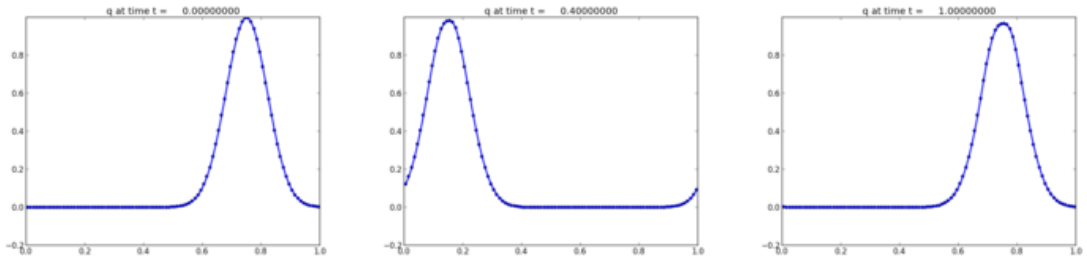
2.1.2 Gallery of all PyClaw applications

Contents

1-dimensional advection

Advecting Gaussian with periodic boundary.

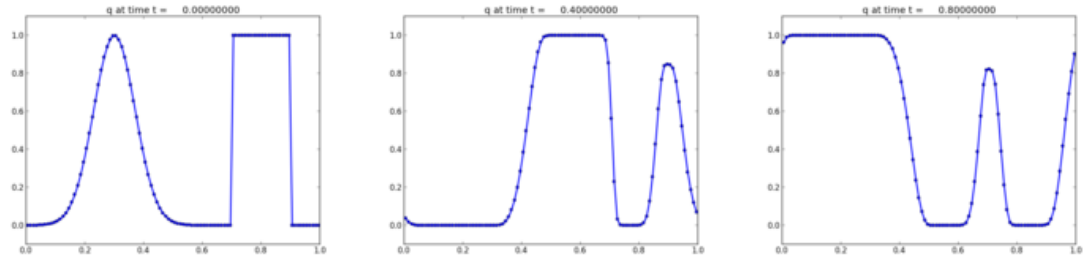
Source code ... Plots



1-dimensional variable-velocity advection

Advecting Gaussian and square wave with periodic boundary.

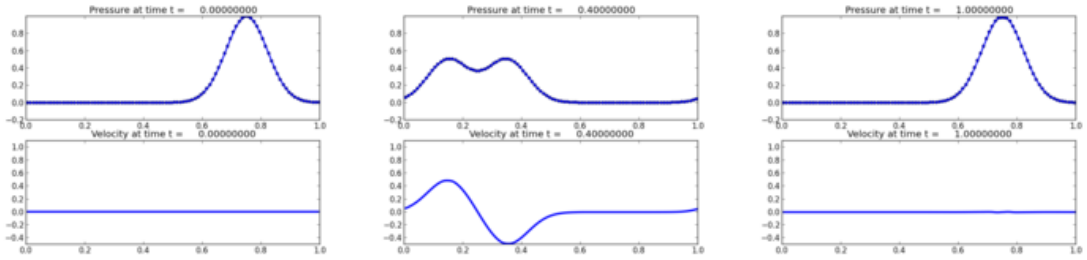
Source code ... Plots



1-dimensional acoustics

Acoustics equations with wall boundary at left and extrap at right.

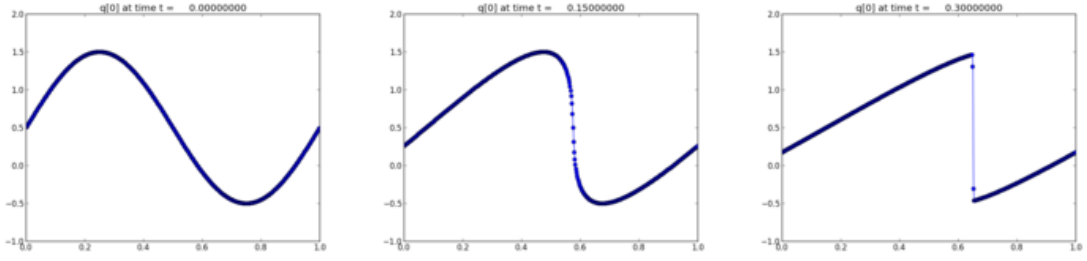
Source code ... Plots



1-dimensional Burgers' equation

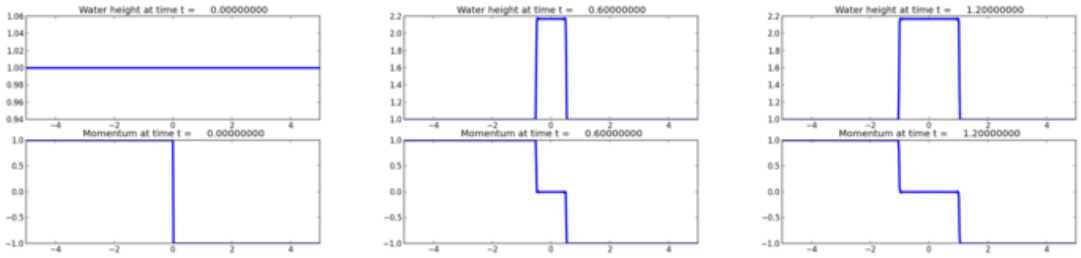
Burgers' equation with sinusoidal initial data, steepening to N-wave.

Source code ... Plots



1-dimensional shallow water equation

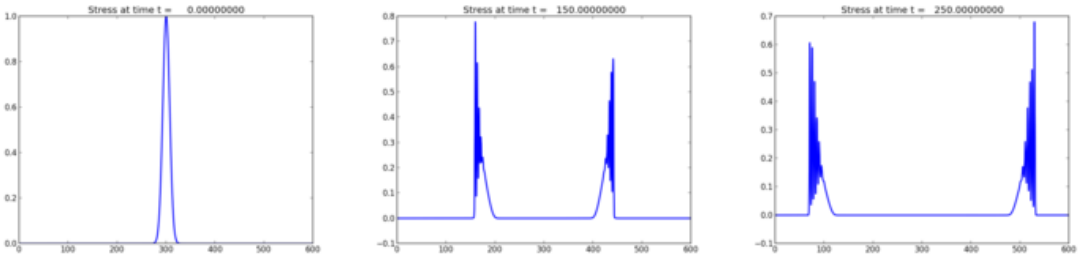
Shallow water shock tube.



Source code ... Plots

1-dimensional nonlinear elasticity

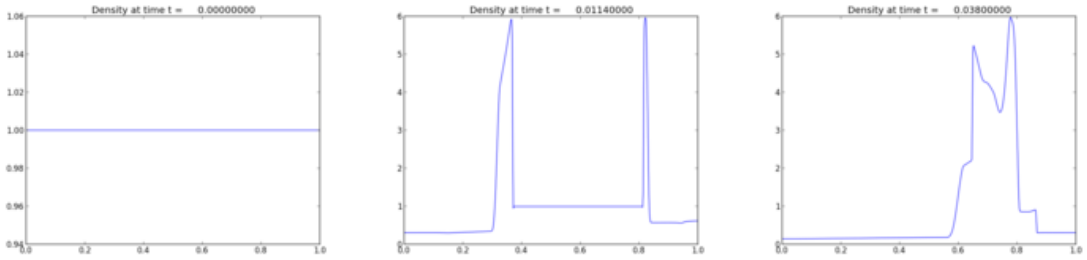
Evolution of two trains of solitary waves from an initial gaussian.



Source code ... Plots

1-dimensional Euler equations

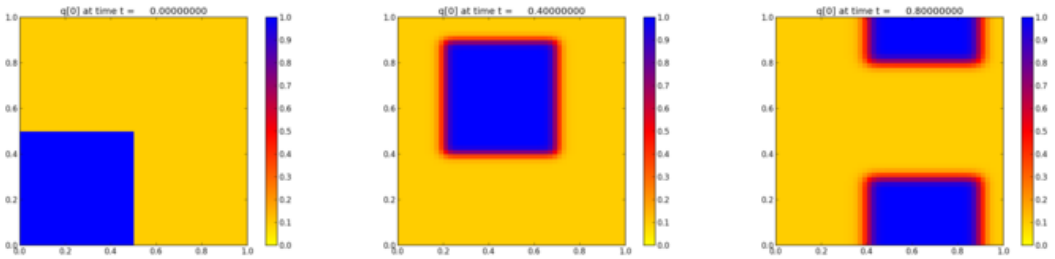
Woodward-Colella blast-wave interaction problem.



Source code ... Plots

2-dimensional advection

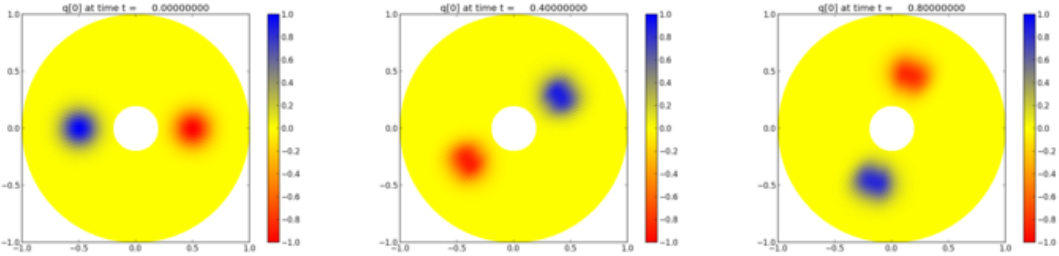
Advecting square with periodic boundary conditions.



Source code ... Plots

2-dimensional variable-coefficient advection

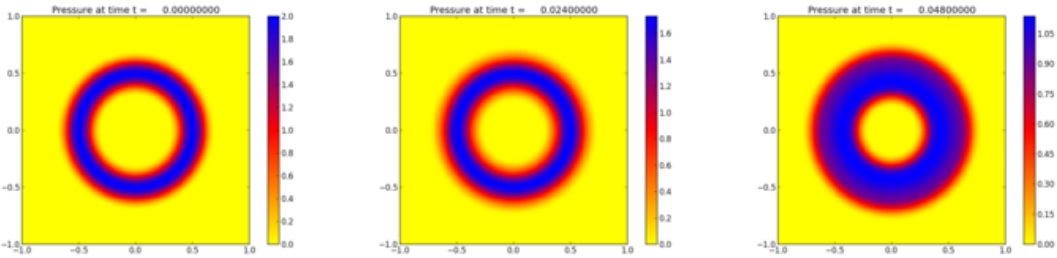
Advection in an annular region.



Source code ... Plots

2-dimensional acoustics

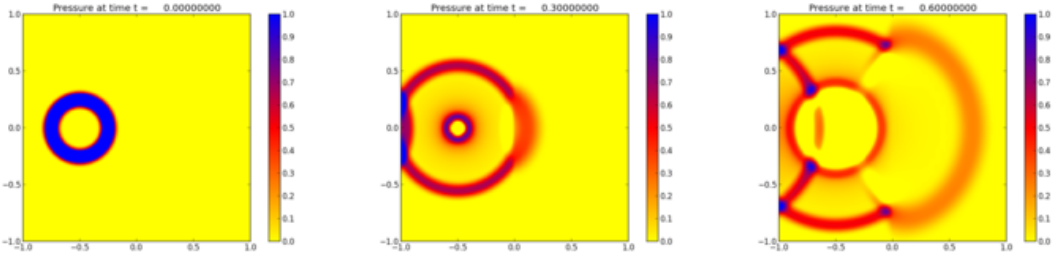
Expanding radial acoustic wave in a homogeneous medium.



Source code ... Plots

2-dimensional variable-coefficient acoustics

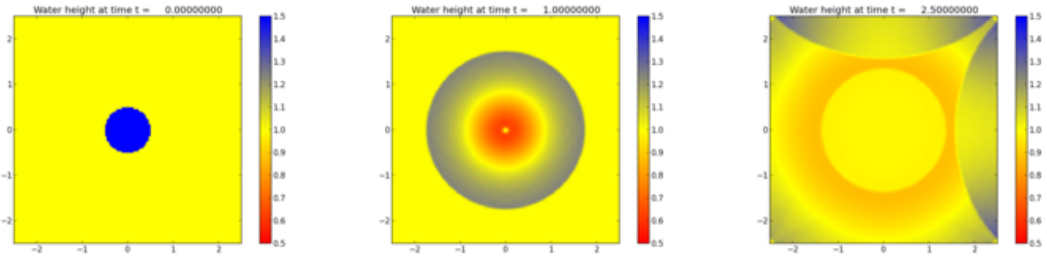
Expanding radial acoustic wave in a two-material medium with an interface.



Source code ... Plots

2-dimensional shallow water equations

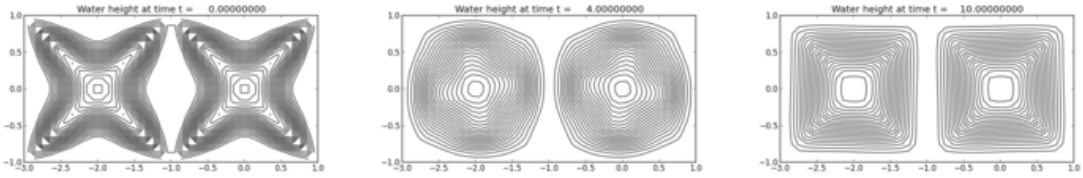
Radial dam-break.



Source code ... Plots

2-dimensional shallow water on the sphere

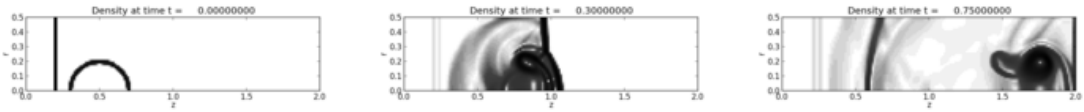
Wavenumber 4 Rossby-Haurwitz wave on a rotating sphere.



Source code ... Plots

2-dimensional Euler equations

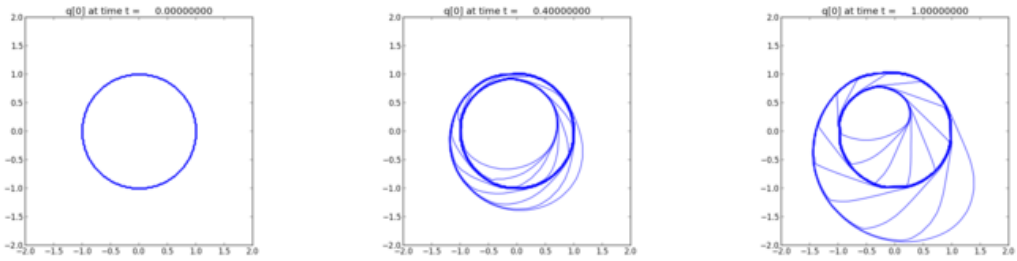
Shock-bubble interaction.



Source code ... Plots

2-dimensional KPP equation

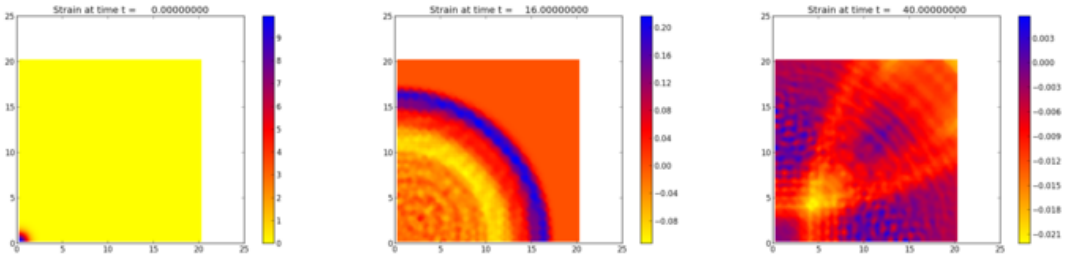
Non-convex flux example.



Source code ... Plots

2-dimensional p-system

Radial wave in a checkerboard-like medium.



Source code ... Plots

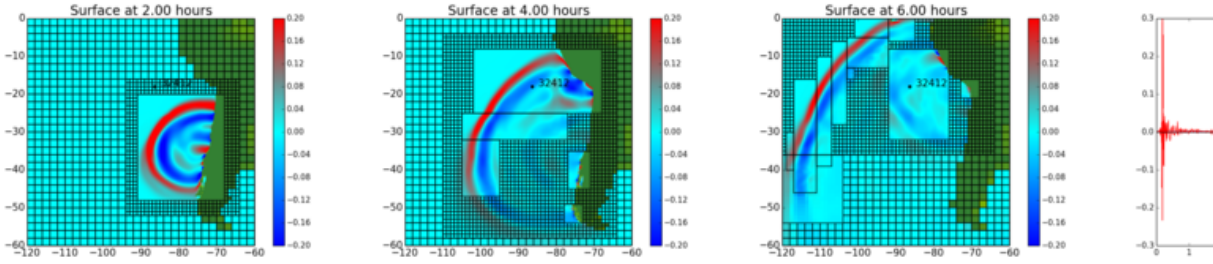
2.1.3 Gallery of GeoClaw applications

Contents

Chile 2010 tsunami

Directory: '$CLAW/geoclaw/examples/tsunami/chile2010'

Simple model of the 2010 tsunami arising offshore Maule, Chile.



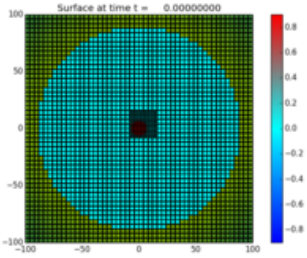
README ... Plots

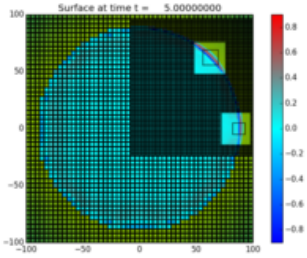
Radially-symmetric tsuanami in parabolic bowl

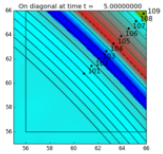
Directory: '$CLAW/geoclaw/examples/tsunami/bowl-radial'

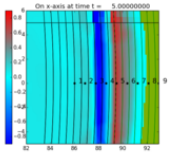
Sample code where solution should be radially symmetric.

README ... Plots







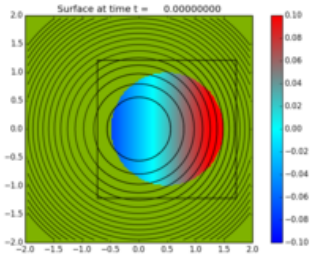


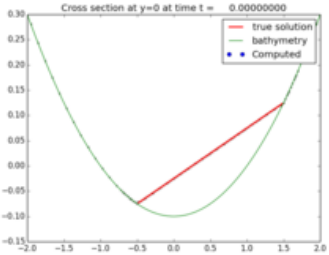
Sloshing water in parabolic bowl

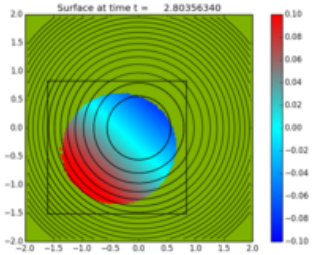
Directory: '$CLAW/geoclaw/examples/tsunami/bowl-slosh'

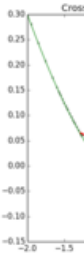
Sample code with analytic solution.

README ... Plots







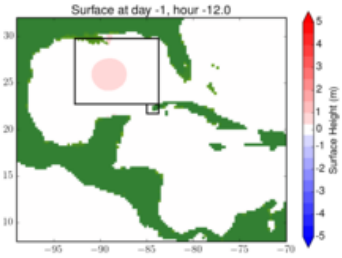


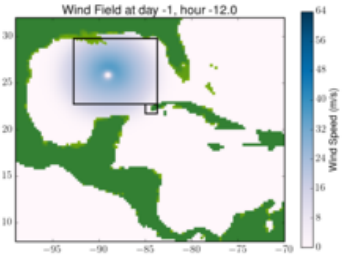
Hurricane Ike

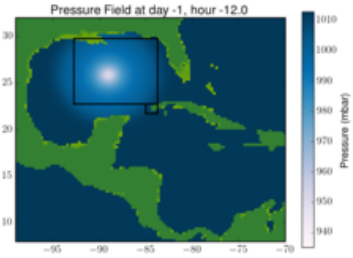
Directory: '$CLAW/geoclaw/examples/storm-surge/ike'

Storm surge simulation of Hurricane Ike (coarse grid)

README ... Plots







Multi-layer shallow water

Directory: '$CLAW/geoclaw/examples/multi-layer/plane_wave'

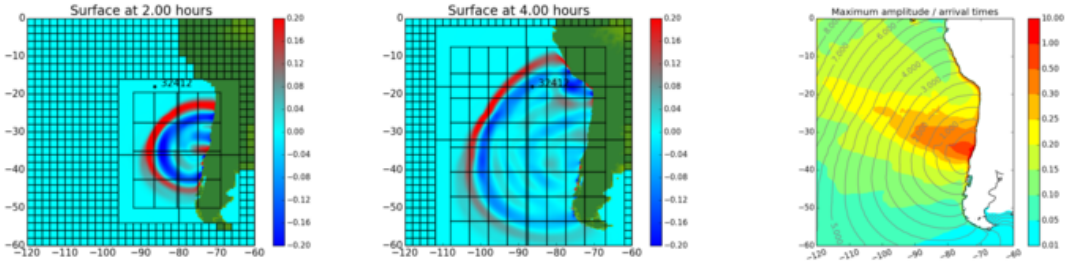
Plane wave hitting shelf with multi-layer equations

README ... Plots

fgmax examples

Directory: ‘$CLAW/apps/tsunami/chile2010_fgmax‘

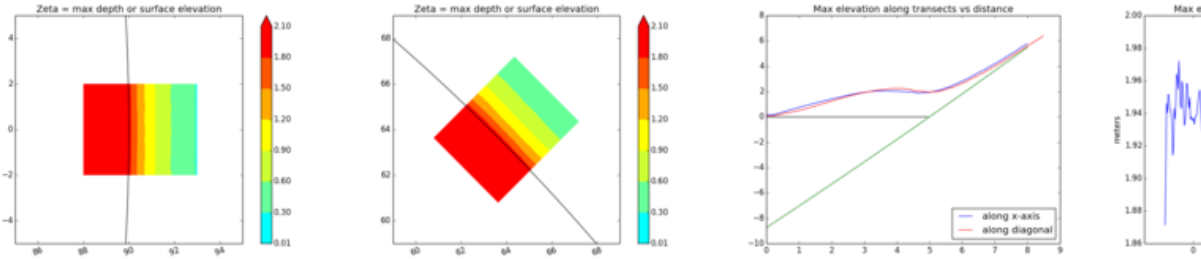
Chile 2010 wave heights and arrival times



README ... Plots

Directory: ‘$CLAW/apps/tsunami/bowl_radial_fgmax‘

Radial bowl comparing maximum amplitudes near x-axis and on diagonal



README ... Plots

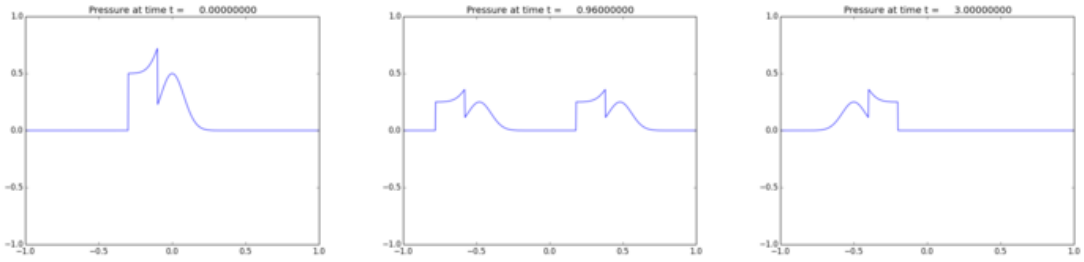
2.1.4 Gallery of *Examples from the book FVMHP* applications

Contents

Chapter 3: Linear Hyperbolic Equations

Directory: '$CLAW/apps/fvmbook/chap3/acousimple'

1D Acoustics simple waves

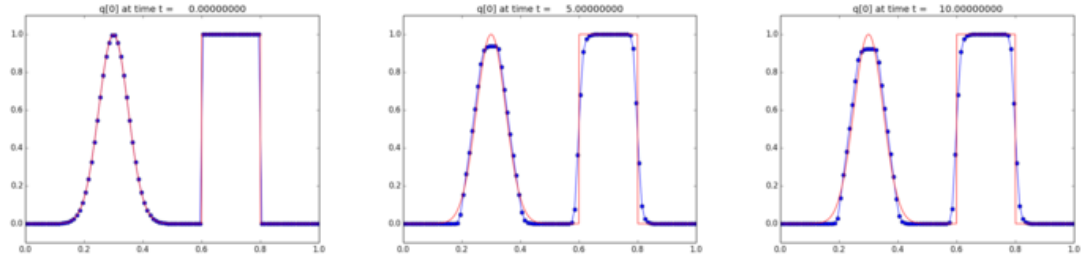


README ... Plots

Chapter 6: High-Resolution Methods

Directory: '$CLAW/apps/fvmbook/chap6/compareadv'

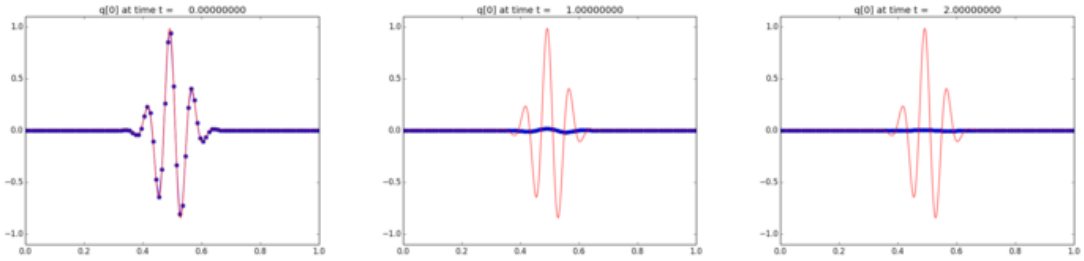
comparison of methods



README ... Plots

Directory: '$CLAW/apps/fvmbook/chap6/wavepacket'

wave packet with 1st order Godunov shown here

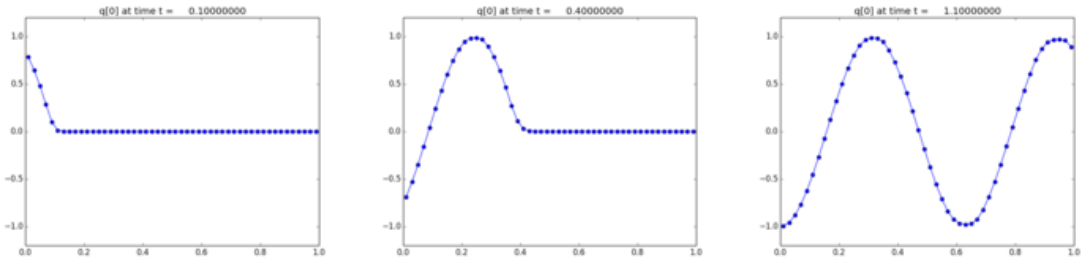


README ... Plots

Chapter 7: Boundary Conditions and Ghost Cells

Directory: '$CLAW/apps/fvmbook/chap7/advinflow'

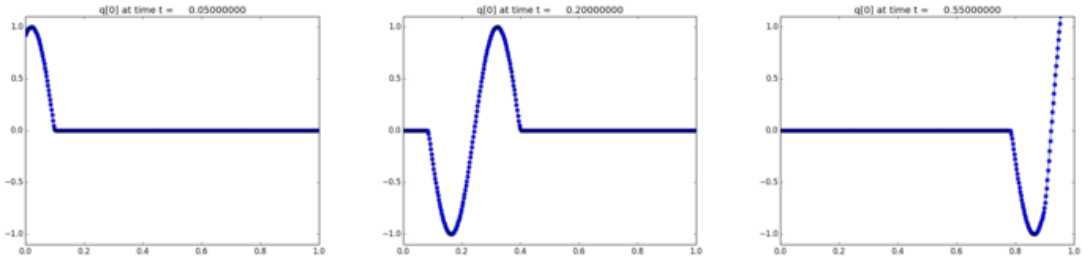
1D Advection with inflow boundary conditions at left and outflow BCs at right



README ... Plots

Directory: '$CLAW/apps/fvmbook/chap7/acouinflow'

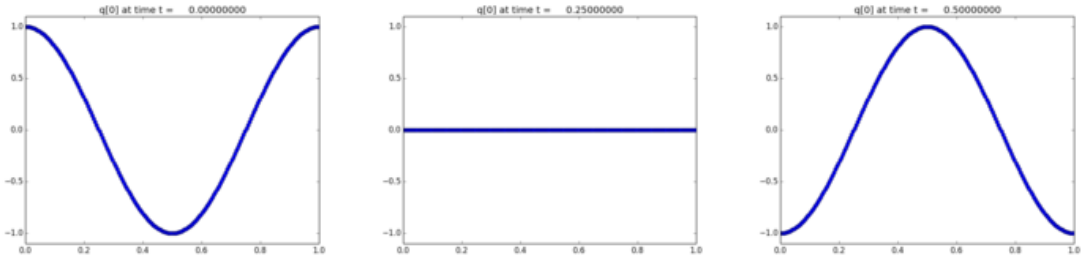
1D Acoustics with inflow boundary conditions at left and reflecting BCs at right



README ... Plots

Directory: '$CLAW/apps/fvmbook/chap7/standing'

1D Acoustics with a standing wave solution

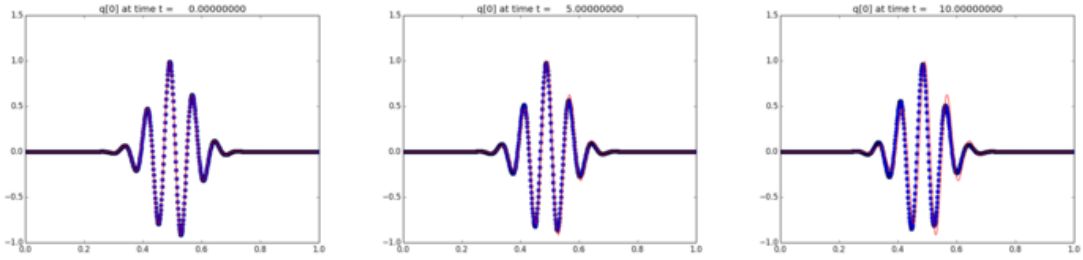


README ... Plots

Chapter 10: Other Approaches to High Resolution

Directory: '$CLAW/apps/fvmbook/chap10/tvb'

1D Advection with a TVB method

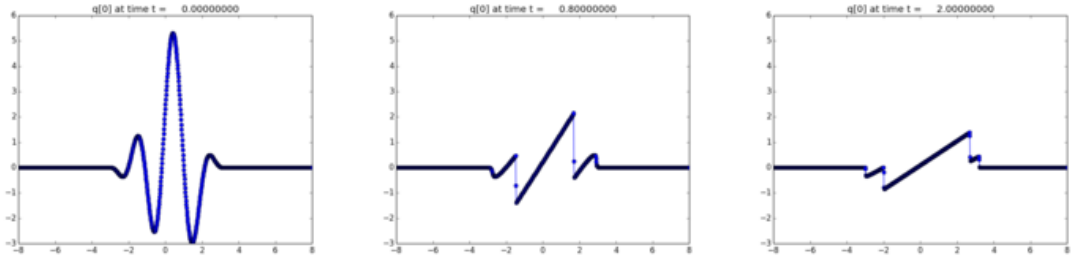


README ... Plots

Chapter 11: Nonlinear Scalar Conservation Laws

Directory: '$CLAW/apps/fvmbook/chap11/burgers'

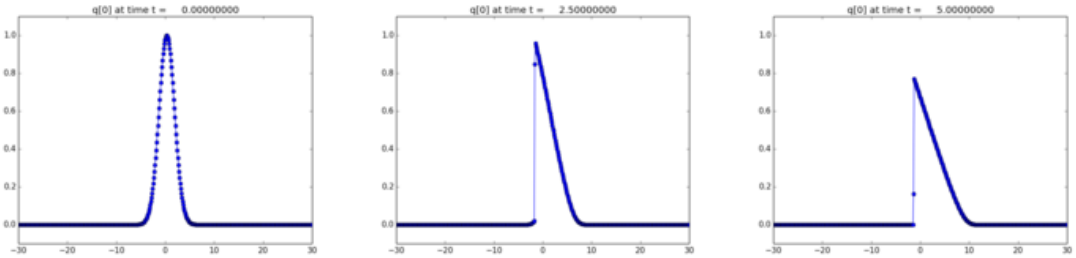
Burgers' equation



README ... Plots

Directory: '$CLAW/apps/fvmbook/chap11/congestion'

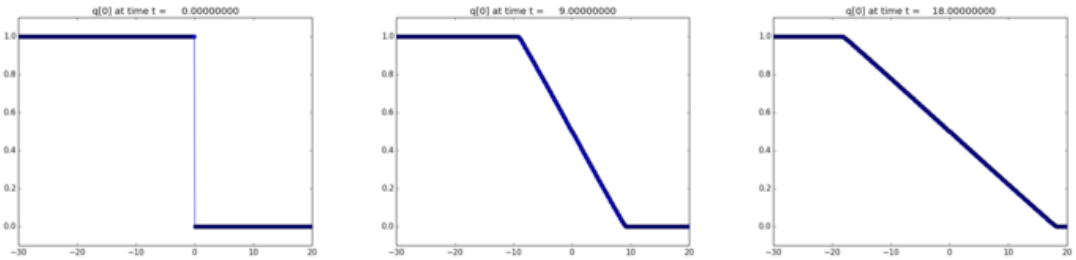
Traffic flow equation with density bulge



README ... Plots

Directory: '$CLAW/apps/fvmbook/chap11/greenlight'

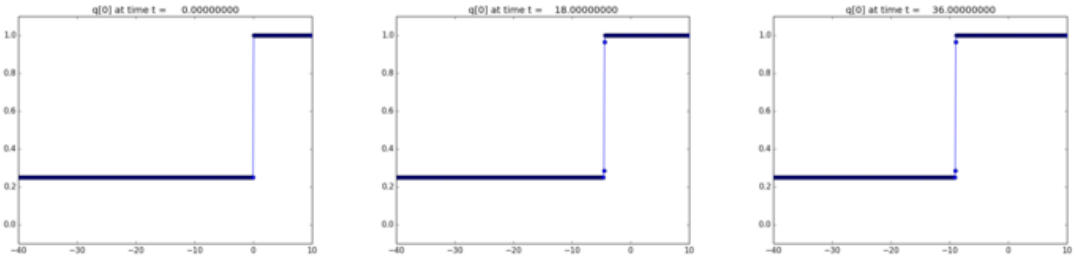
Traffic flow equation with expansion fan



README ... Plots

Directory: '$CLAW/apps/fvmbook/chap11/redlight'

Traffic flow equation with shock wave behind red light

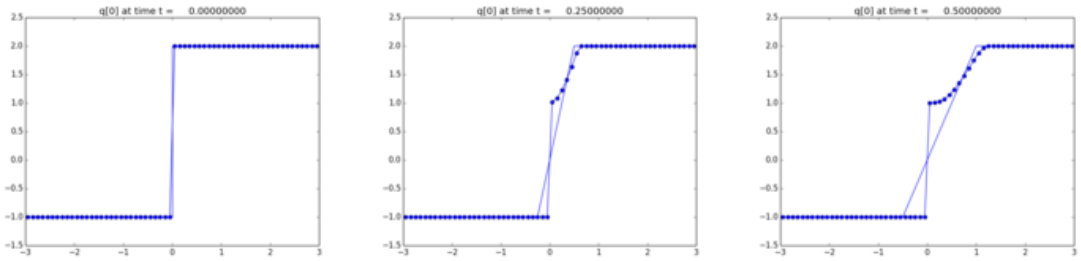


README ... Plots

Chapter 12: Finite Volume Methods for Nonlinear Scalar Conservation Laws

Directory: '$CLAW/apps/fvmbook/chap12/efix'

Burgers' equation without entropy fix

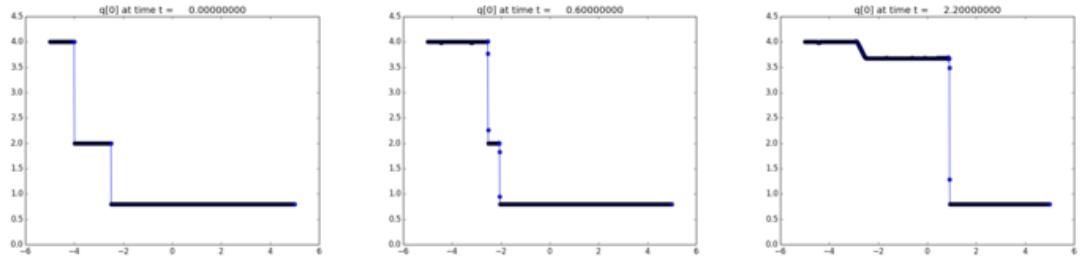


README ... Plots

Chapter 13: Nonlinear Systems of Conservation Laws.

Directory: ‘$CLAW/apps/fvmbook/chap13/collide‘

Colliding and merging shock waves in shallow water equations

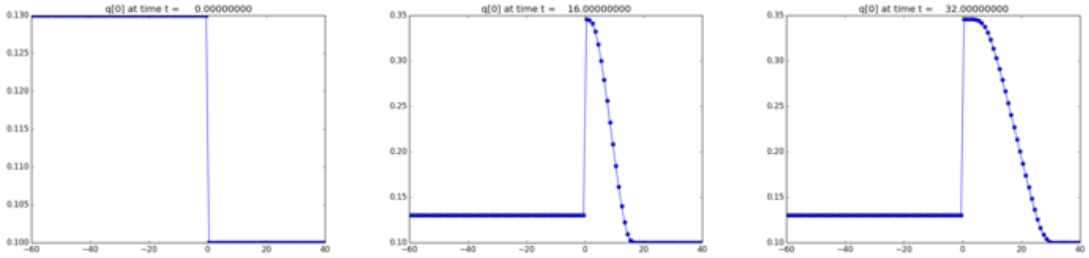


README ... Plots

Chapter 16: Some Nonclassical Hyperbolic Problems

Directory: ‘$CLAW/apps/fvmbook/chap16/vctraffic‘

Traffic equations with a spatially varying flux

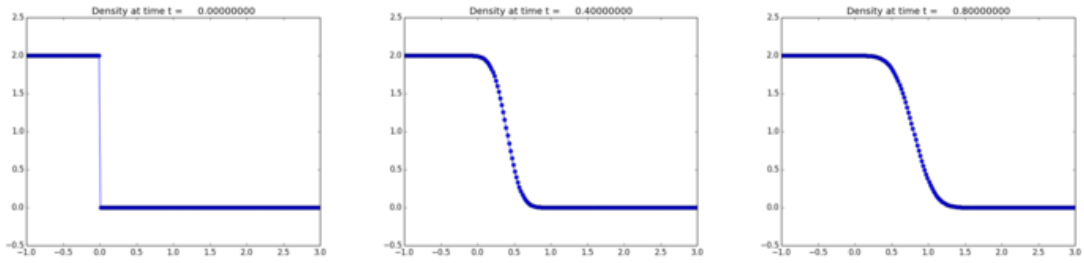


README ... Plots

Chapter 17: Source Terms and Balance Laws

Directory: ‘$CLAW/apps/fvmbook/chap17/advdiff‘

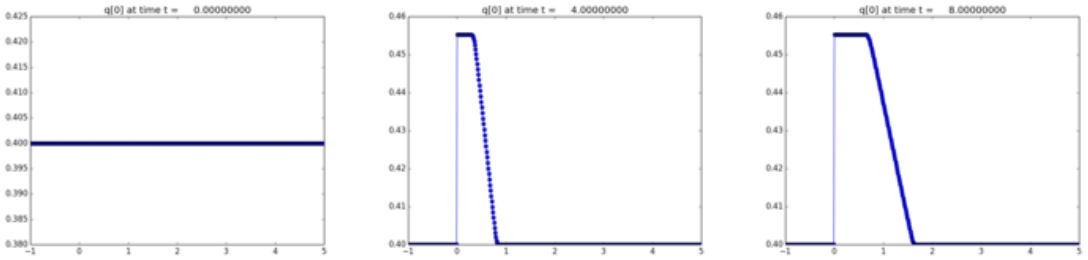
Advection-diffusion with implicit solver



README ... Plots

Directory: '$CLAW/apps/fvmbook/chap17/onramp'

Traffic flow with an on-ramp

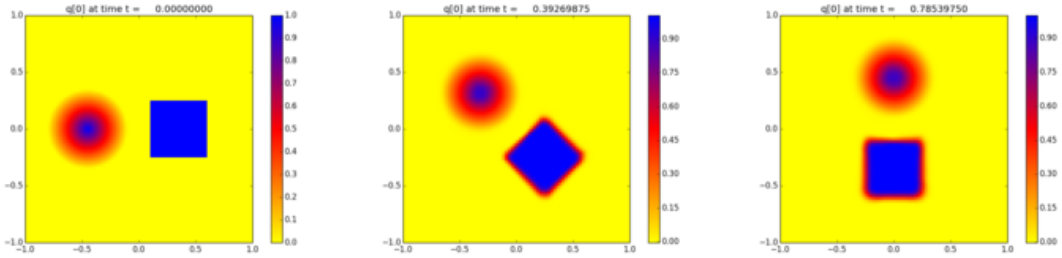


README ... Plots

Chapter 20: Multidimensional Scalar Equations

Directory: '$CLAW/apps/fvmbook/chap20/rotate'

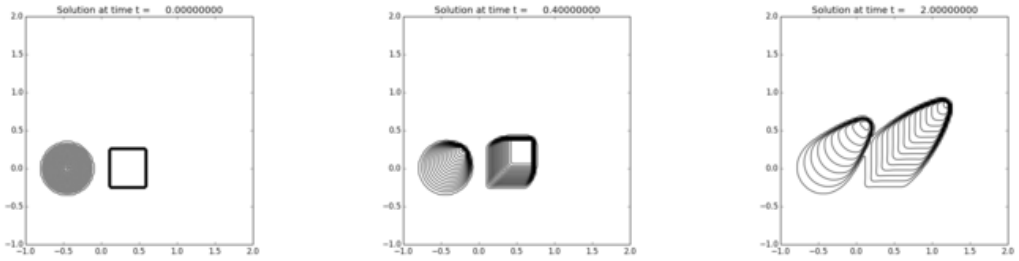
2D Advection with a rotation of square and Gaussian



README ... Plots

Directory: '$CLAW/apps/fvmbook/chap20/burgers'

2D Burgers' equation with square and Gaussian data



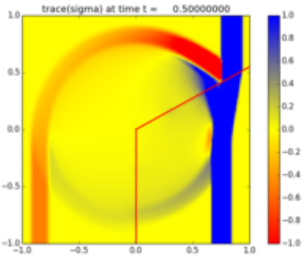
README ... Plots

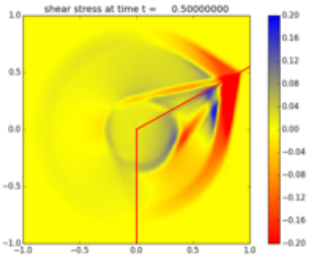
Chapter 22: Elastic Waves

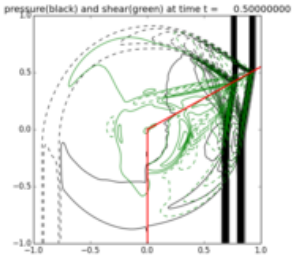
Directory: '$CLAW/apps/fvmbook/chap22/corner'

Elasticity equations in a region consisting of two materials with a corner

README ... Plots







2.2 Clawpack Applications repository

More complex examples and applications are archived in the Github *clawpack/apps* repository found at https://github.com/clawpack/apps.

In particular, the directory *apps/fvmbook* contains many *Examples from the book FVMHP* (page 289).

These examples are not included in the basic Clawpack installation. Users interested in obtaining this collection of applications can either clone the repository using git:

```
git clone git://github.com/clawpack/apps
```

or navigate to https://github.com/clawpack/apps and click on the "Download ZIP" button at the bottom of the page.

2.2.1 Examples included with Clawpack

Recall that a few examples of how to use different flavors of Clawpack are included in the directories

- *$CLAW/classic/examples*
- *$CLAW/amrclaw/examples*
- *$CLAW/geoclaw/examples*
- *$CLAW/pyclaw/examples*

These examples demonstrate some of the basic capabilities. The plots resulting from running these examples should agree with those seen in the *Galleries of all Clawpack applications* (page 25).

2.3 Examples from the book FVMHP

The book Finite Volume Methods for Hyperbolic Problems (http://faculty.washington.edu/rjl/book.html) contains many examples that link to Clawpack codes used to create the figures in the book. These codes are available for Clawpack 4.3 via the book webpage (http://faculty.washington.edu/rjl/book.html)

Some of have been converted to Clawpack 5.x form, with a *setrun.py* file for setting run time data and a *setplot.py* file for specifying plots with Python. See:

- *Specifying classic run-time parameters in setrun.py* (page 56)
- *Using setplot.py to specify the desired plots* (page 251)

2.3.1 Available examples

The examples converted so far can be found in the directory *apps/fvmbook* if you clone the *apps* repository. (See *Clawpack Applications repository* (page 41).)

You can also browse the examples in the *Gallery of fvmbook applications* (page 35).

2.4 Creating a new application directory

2.4.1 Copying an existing example

The simplest approach to implementing something new is to start with a Clawpack example and modify the code appropriately.

Rather than modifying one of the examples in place, it is best to copy it to a new directory. You might want to create a directory $CLAW/myclaw at the top level of Clawpack that can be used to put your own work, but Clawpack should work from any directory as long as the environment variable is set properly. (See *Set environment variables* (page 9).)

In unix/linux you can copy a directory recursively (with all subdirectories intact) using the *cp -r* command, e.g.

```
$ cp -r $CLAW/classic/examples/acoustics_1d_example1  path-to-newdir
```

2.5 Saving and sharing results

Clawpack now includes some tools to help facilitate archiving and sharing results that you have obtained with this software. These make it relatively easy to generate a set of webpages such as those seen when browsing the examples collected in *Galleries of all Clawpack applications* (page 25).

These webpages can easily be posted on your own website to be viewed by others if you wish, for example to share on-going work with collaborators or to supplement a journal article.

2.5.1 Making webpages of plots

The "make .plots" option available via the standard Makefiles will create a set of webpages illustrating the plots and allowing easy navigation between frames. These webpages also allow viewing all frames of a plot as an animation (via javascript within the browser).

See *Visclaw Plotting options* (page 247) for more details on how to specify what plots will appear on these webpages.

2.5.2 Sharing your results

To make it easy for others to view your code and the resulting plots, you can simply copy the example directory (containing the code and the _plots subdirectory) to your publicly visible web pages.

2.5.3 IPython notebooks

You should also consider creating an IPython notebook to explain your problem, illustrate your workflow, and present plots and animations all in one. See *IPython notebook examples* (page 53) for some examples.

2.6 Contributing examples and applications

The *Clawpack Applications repository* (page 41) contains a few examples of applications that can be solved using Clawpack. We hope to greatly expand this repository in the future.

Contributions from users are desired. We are still working out the best way to collect applications. One possibility is to use git submodules. For now you can issue a pull request to https://github.com/clawpack/apps.

Stay tuned for more details....

Of course you can always post your results using the techniques described in *Saving and sharing results* (page 42).

2.7 Testing your installation

2.7.1 PyClaw

Regression tests can be performed via:

```
cd $CLAW/pyclaw
nosetests
```

2.7.2 Fortran codes

Many repositories have a subdirectory named *tests* that contain a few regression tests that run very quickly and check that a few numbers determined from the solution agree with values stored in the example directories. For example, to run some quick tests of *amrclaw*:

```
cd $CLAW/amrclaw/tests
make tests
```

will run several tests and report the results.

More extensive tests can be performed by running all of the examples in the *examples* directory and comparing the resulting plots against those archived in the *Galleries of all Clawpack applications* (page 25). See also *Regression testing* (page 284).

2.8 Compiling the Sphinx documentation locally

For most users, the best way to view the documentation is online (http://clawpack.github.io/index.html).

The source files that create this documentation are included in the repository clawpack/doc (https://github.com/clawpack/doc) repository in *$CLAW/doc/doc*. See *Guide for updating this documentation* (page 282) if you want to create and view them locally.

CHAPTER

THREE

CLASSIC, AMRCLAW, AND GEOCLAW

3.1 Using the Fortran codes

3.1.1 Fortran version

Input parameters are generally specified in a Python script `setrun.py` and then:

```
$ make .data
```

creates the `*.data` files that the Fortran code requires.

Makefiles

Most example directories contain a `Makefile` that offers several options. Type:

```
$ make help
```

for a list. Often:

```
$ make .plots
```

is all you need to type to create the data files, compile the code, run it, and produce plots as png and html files.

Or, if you just want to run the code and produce output without making all the plots (and then do the plotting interactively, for example):

```
$ make .output
```

Note: There is a dot before `plots` and `output` in the above commands.

The directory where output and plots are stored is specified in the Makefile.

The Makefile in most directories includes a common Makefile found at *$CLAW/clawutil/src/Makefile.common* that does most of the work. If you get the error message:

```
Makefile:  /clawutil/src/Makefile.common: No such file or directory
```

then the environment variable *CLAW* is not set properly. See *Set environment variables* (page 9).

More tips

- The "make .output" command runs the code and stores the name of the output directory in the file `.output` and it is the modification time of this file that is checked relative to the dependencies. (Note: the unix command ls generally does not display files that start with a dot so this file may be invisible unless you use "ls -a".)

If you want to re-run the code and encounter:

```
$ make .output
$ make: '.output' is up to date.
```

you can remove the file `.output` to force the code to be run again.

- Similarly, remove the file `.plots` to force the plots to be recreated.
- If you change the compiler flags FFLAGS in the Makefile or as an environment variable, then you should make sure that all files used are recompiled with the new flags. The Makefiles as written do not catch this dependency and will not recompile all the .o files when the Makefile changes. To force recompilation, use:

  ```
  $ make new
  ```

  See *Fortran Compilers* (page 46) for more about compiler flags.

3.1.2 Fortran Compilers

This section is relevant to users who want to compile the fortran code in the classic, amrclaw, or geoclaw branches.

FC environment variable

Users should set the environment variable *FC* to point to the correct compiler, e.g. in bash via:

```
$ export FC=gfortran
```

Note that some versions of *make* will set *FC=f77* by default if no value is specified, and adding a line to the Makefile such as:

```
FC ?= gfortran
```

will not override this. The common Makefile in *$CLAW/clawutil/src/Makefile.common* now tests to see if *FC* is set to *f77* and if so resets it to *gfortran* since much of Clawpack is not *f77* compliant. However, it is best to set the *FC* environment variable yourself, e.g. in your *.bashrc* file.

FFLAGS environment variable

Compiler flags can be specified using the *FFLAGS* variable that can be set in an application Makefile. By default sample Makefiles now specify:

```
FFLAGS ?=
```

so that no flags are used unless the environment variable *FFLAGS* is set already. This line can be changed in the Makefile, but it is often easiest to set an environment variable for the flags you generally want to use.

Note: If you change the flags you generally have to recompile *all* the code, and this dependency is not handled automatically. So always do:

```
$ make new
```

before rerunning an example with *make .output* or *make .plots*.

gfortran compiler

Some useful flags:

- For debugging:

```
FFLAGS = -g -Wall -pedantic -fbounds-check -ffpe-trap=invalid,overflow,zero
```

- For optimizing:

```
FFLAGS = -O2
```

- For using OpenMP:

```
FFLAGS = -O2 -openmp
```

  In this case you should also set some environment variables. See *Using OpenMP* (page 50) for details.

  Note: Versions of gfortran before 4.6 are known to have OpenMP bugs.

Intel fortran compiler

Set the *FC* environment variable to *ifort*.

Some useful flags:

- For debugging:

```
FFLAGS = -g -C -CB -CU -fpe0 -ftrapuv -fp-model precise
```

- For optimizing:

```
FFLAGS = -O2
```

- For using OpenMP:

```
FFLAGS = -O2 -openmp
```

  In this case you should also set the environment variable *OMP_NUM_THREADS* to indicate how many threads to use.

3.1.3 User files required for the Fortran code

The *Makefile* in an application directory shows the set of Fortran source code files that are being used. Most of these files are typically in one of the libraries, but a few subroutines must be provided by the user in order to specify the hyperbolic problem to be solved and the initial conditions. Other subroutines may also be provided that are application-specific. This page summarizes some of the most common user-modified routines.

The calling sequence for each subroutine differs with the number of space dimensions. The sample calling sequences shown below are for one space dimension.

The subroutines described below have default versions in the corresponding library and the *Makefile* can point to these if application-specific versions are not needed.

See the examples in the following directories for additional samples:

- *$CLAW/classic/examples*
- *$CLAW/amrclaw/examples*
- *$CLAW/geoclaw/examples*

You can also browse from the *Galleries of all Clawpack applications* (page 25) to the *README* file for an example and then to the source code for the application-specific codes.

Specifying the initial conditions

Calling sequence in 1d:

```
subroutine qinit(meqn,mbc,mx,xlower,dx,q,maux,aux)
```

See the *qinit_defaults* for other calling sequences and the proper declaration of input/output parameters.

Typically every application directory contains a file *qinit.f* or *qinit.f90* that sets the initial conditions, typically in a loop such as:

```
do i=1,mx
    xi = xlower + (i-0.5d0)*dx
    q(1,i) = xi**2
    enddo
```

This loop would set the value of q^1 in the i'th cell to x_i^2 where x_i is the cell center. For the finite volume methods used in Clawpack, the initial data should really be set to be the cell average of the data over each grid cell, determined by integrating the data for the PDE. If the initial data is given by a smooth function, then evaluating the function at the center of the grid cell generally agrees with the cell average to $\mathcal{O}(\Delta x^2)$ and is consistent with the second-order accurate high-resolution methods being used in Clawpack.

For a system of more than 1 equation, you must set *q(m,i)* for *m = 1, 2, ..., num_eqn*.

For adaptive mesh refinement codes, the *qinit* subroutine will be called for each grid patch at the initial time, so it is always necessary to compute the cell centers based on the information passed in.

Specifying the Riemann solver

The Riemann solver defines the hyperbolic equation that is being solved and does the bulk of the computational work – it is called at every cell interface every time step and returns the information about waves and speeds that is needed to update the solution.

See *riemann* for more details about the Riemann solvers.

All of the examples that come with Clawpack use Riemann solvers that are provided in the directory *$CLAW/riemann/src*, see the *Makefile* in one of the examples to determine what Riemann solver file(s) are being used (in two and three space dimensions, transverse Riemann solvers are also required).

The directory *$CLAW/riemann/src* contains Riemann solvers for many applications, including advection, acoustics, shallow water equations, Euler equations, traffic flow, Burgers' equation, etc.

Specifying boundary conditions

Boundary conditions are set by the library routines:

- *$CLAW/classic/src/Nd/bcN.f* for the classic code (N = 1, 2, 3).
- *$CLAW/amrclaw/src/Nd/bcNamr.f* for the amrclaw code (N = 2, 3).

Several standard choices of boundary condition procedures are provided in these routines – see *Boundary conditions* (page 65) for details.

For user-supplied boundary conditions that are not implemented in the library routines, the library routine can be copied to the application directory and changes made as described at *User-defined boundary conditions* (page 66). The *Makefile* should then be modified to point to the local version.

Specifying problem-specific data

Often an application problem has data or parameters that is most conveniently specified in a user-supplied routine named *setprob*. There is a library version that does nothing in case one is not specified in the application directory. As usual, the *Makefile* indicates what file is used.

The *setprob* subroutine takes no arguments. Data set in *setprob* is often passed in common blocks to other routines, such as *qinit* or the Riemann solver. This is appropriate only for data that does not change with time and does not vary in space (e.g. the gravitational constant g in the shallow water equations, or the density and bulk modulus for acoustics in a homogenous medium).

Note that named common blocks must have the same name in each routine where they are used. Check any Riemann solvers you use (including those from *$CLAW/riemann/src*) to see if they require some parameters to be passed in via a common block. If so, *setprob* is the place to set them.

For spatially-varying data, see *Specifying spatially-varying data using setaux* (page 49) below.

Often *setprob* is written so that it reads in data values from a file, often called *setprob.data*. This makes it easier to modify parameter values without recompiling the code. It is also possible to set these values in *setrun.py* so that this input data is specified in the same file as other input parameters. For a sample, see *$CLAW/classic/examples/acoustics_1d_heterogeneous*, for example.

Specifying spatially-varying data using *setaux*

Some problems require specifying spatially varying data, for example the density and bulk modulus for acoustics in a heterogenous medium might vary in space and in principle could be different in each grid cell. The best way to specify such data is by use of *auxiliary arrays* that are created whenever a grid patch for the solution is created and have the same number of cells with *num_aux* components in each cell. The value *num_aux* is specified in *setrun.py*, and the contents of the *aux* arrays are filled by a subroutine named *setaux*, which in one dimension has the calling sequence:

```
subroutine setaux(mbc,mx,xlower,dx,maux,aux)
```

See the *setaux_defaults* for other calling sequences and the proper declaration of input/output parameters.

If adaptive refinement is being used, then every time a new grid patch is created at any refinement level this subroutine will be called to fill in the corresponding *aux* arrays. For a sample, see *$CLAW/classic/examples/acoustics_1d_heterogeneous*, for example.

If the *aux* arrays need to be time-dependent, the easiest way to adjust them each time step is in the routine *b4step* described below.

Using *b4step* for work to be done before each time step

The routine *b4stepN* is called in N space dimensions (N=*1,2,3*) just before a time step is taken (and after ghost cells have been filled by the boundary conditions). The library version of this routine does nothing, but this can be modified to do something prior to every time step.

In one dimension the calling sequence is:

```
subroutine b4step1(mbc,mx,meqn,q,xlower,dx,t,dt,maux,aux)
```

See the *b4step_defaults* for other calling sequences and the proper declaration of input/output parameters.

For example, in *$CLAW/amrclaw/examples/advection_2d_swirl* the advection equation is solved with an advection velocity that varies in time as well as space. This is initialized for each grid patch in *setaux*, but is adjusted each time step in *b4step2*.

Using *src* for source terms

Problems of the form $q_t(x,t) + f(q(x,t))_x = \psi(q,x,t)$ can be solved using a fractional step approach, as described in Chapter 17 of [LeVeque-FVMHP] (page 292). The user can provide a subroutine named *srcN* in *N* space dimensions that takes a single time step on the equation $q_t = \psi$. In one dimension the calling sequence is:

```
subroutine src1(meqn,mbc,mx,xlower,dx,q,maux,aux,t,dt)
```

On output the *q* array should have been updated by using the input values as initial data for a single step of length *dt* starting at time *t*.

See the *src_defaults* for other calling sequences and the proper declaration of input/output parameters.

The library version of *srcN* does nothing. If you copy this to an application directory and modify for your equation, you must modify the *Makefile* to point to the local version. You must also set the *source_split* parameter in *setrun.py* (see *Specifying classic run-time parameters in setrun.py* (page 56)) to either *"godunov"* or *"strang"*. In the former case, the 1st order Godunov splitting is used (after each time step on the homogenous hyperbolic equation, a time step of the same length is taken on the source terms). In the latter case the 2nd order Strang splitting is used: the time step on the hyperbolic part is both preceeded and followed by a time step of half the length on the source terms.

For an example where source terms are used, see *$CLAW/classic/examples/acoustics_2d_radial/1drad* where a one-dimensional acoustic equation with a geometric source term is solved in order to provide a reference solution for the two-dimensional radially symmetric problem solved in *$CLAW/classic/examples/acoustics_2d_radial*.

Using *src1d* for source terms with AMRClaw

When the AMRClaw code is used for a problem in 2 or 3 dimensions with source terms, then a subroutine *srcN* must be provided as described above. In addition, for the AMR procedure to work properly it is also necessary to provide another subroutine *src1d* with calling sequence:

```
subroutine src1d(meqn,mbc,mx1d,q1d,maux,aux1d,t,dt)
```

See the *src1d_defaults* for other calling sequences and the proper declaration of input/output parameters.

This routine should be a simplified version of *src2* or *src3* that takes a one-dimensional set of data in *q1d* rather than a full 2- or 3-dimensional array of data. The input array *aux1d* has the corresponding set of auxiliary variables in case these are needed in stepping forward with the source terms.

If the source terms depend only on *q*, it should be easy to adapt src2 to create this routine, simply by looping over *i=1:mx1d* rather than over a multi-dimensional array.

This routine is used in computing adjustments around a fine grid patch that are needed in order to maintain global conservation after values in a coarser grid cell have been overwritten with the average of the more accurate fine grid values. Adjustment of the coarse grid values in the cells bordering this patch is then required to maintain conservation. This requires solving a set of Riemann problems between fine-grid and coarse-grid values around the edge of the patch and *src1d* is used in advancing coarse grid values to intermediate time steps.

The code may work fine without applying source terms in this context, so using the dummy library routine *src1d* might be successful even when source terms are present.

3.1.4 Using OpenMP

The Clawpack Fortran Classic 3d code, AMRClaw 2d and 3d code, and GeoClaw codes include OpenMP directives for making use of multicore shared memory machines.

Note: Versions of gfortran before 4.6 are known to have OpenMP bugs. You should use a recent version or a different compiler if you want to use OpenMP.

To invoke OpenMP you need to compile the entire code with appropriate compiler flags (see *Fortran Compilers* (page 46)). For example, with gfortran and the bash shell you could do:

```
export FFLAGS='-O2 -fopenmp'  # or hardwire FFLAGS in the Makefile
make new
```

in an application directory, which should recompile all of the library routines as well.

Then you may want to specify how many threads OpenMP should split the work between, e.g.

```
export OMP_NUM_THREADS=2
```

If you do not set this environment variable some default for your system will be used.

You may also need to increase the stack size if the code bombs for no apparent reason (and no useful error message):

```
export OMP_STACKSIZE=16M
```

and also:

```
ulimit -s unlimited
```

On a Mac this isn't allowed and the best you can do is

```
ulimit -s hard
```

To stop using OpenMP you could do:

```
export FFLAGS=-O2   # or hardwire FFLAGS in the Makefile
make new
```

Using OpenMP with AMR

The code in AMRClaw and GeoClaw is parallelized by splitting the list of patches that must be advanced in time between threads, and then each grid patch is handled by a single thread. For this reason good performance will be seen only when there are a sufficiently large number of patches at each level relative to the number of threads. For this reason it is recommended that the parameter *max1d* be set to 60 in the modules

- *$CLAW/amrclaw/src/2d/amr_module.f90*
- *$CLAW/amrclaw/src/3d/amr_module.f90*

when OpenMP is used. This limits the size of any patch to have at most *max1d* grid cells in each direction. If OpenMP is not used, a larger value of *max1d* might give somewhat better performance since there is less overhead associated with passing boundary values in ghost cells and other per-patch work. However, this is generally negligible and *max1d=60* is the default value set in the code. If you do change this value, remember to recompile everything via:

```
make new
```

Fixed grid output in GeoClaw

The original fixed grid output routines are not thread safe and so OpenMP should not be used if you want to produce output on fixed grids.

The newer fgmax routines that keep track of maxima on fixed grids should be thread safe, see *fgmax*.

3.1.5 Python Hints

Contents

Python is a powerful object-oriented interpreted scripting/programming language. Some version of Python is almost certainly on your computer already (on unix, linux, OSX type:

```
$ python --version
```

to find out which version, or just:

```
$ python
```

to start a python shell with a prompt that looks like:

```
>>>
```

You may prefer to use IPython (http://ipython.scipy.org/moin/), which is a nicer shell than the pure python shell, with things like command completion and history. See the Quick IPython Tutorial (http://ipython.scipy.org/doc/manual/html/interactive/tutorial.html).

Installation of required modules

To effectively use the pyclaw and Clawpack plotting routines that are written in Python, you will need version 2.7 or greater (but **not** 3.0 or above, which is not backwards compatible). some modules that are not included in the standard Python distribution. Python modules are loaded with the *import* statement in Python and a wide variety of specialized modules exist for various purposes since people use Python for many different purposes.

An alternative to installing the packages discussed below, you could also use the *Clawpack Virtual Machine* (page 16), which has all the required Python modules pre-installed.

For use with Clawpack, you will need the Numpy (http://www.numpy.org/) module (*Numerical Python*) that allows working with arrays in much the same way as in Matlab. This is distributed as part of SciPy (http://docs.scipy.org/doc/) (*Scientific Python*). See the Installing SciPy (http://www.scipy.org/Installing_SciPy) page for tips installing SciPy and NumPy on various platforms.

For plotting you will also need the matplotlib (http://matplotlib.org/) module which provides Matlab-like plotting commands for 1d and 2d plots (e.g. contour and pcolor plots). This is generally the hardest thing to get going properly. See the matplotlib Installation FAQ (http://matplotlib.sourceforge.net/faq/installing_faq.html).

Often the easiest way to get all the modules you need is to install the

Anaconda Python Distribution (http://docs.continuum.io/anaconda/) or the Enthought Python Distribution (http://www.enthought.com/products/epd.php), which is free for academic users. Versions are available for Windows, Mac OS X, and Redhat linux.

With some versions of Linux (e.g. debian and Ubuntu), you can easily install what's needed using `apt-get`:

```
$ apt-get install python-numpy
$ apt-get install python-scipy
$ apt-get install python-matplotlib
```

For OS X, you might also try the Scipy Superpack (http://stronginference.com/scipy-superpack/). See also these tips on installing matplotlib on OS X (http://matplotlib.sourceforge.net/faq/installing_faq.html#os-x-questions).

References and tutorials

Some useful links to get started learning Python:

- Enthought Python Distribution (http://www.enthought.com/products/epd.php)
- Dive Into Python (http://www.diveintopython.org/)
- Python tutorial (http://www.python.org/doc/tut/)
- NumPy User Guide (http://docs.scipy.org/doc/numpy/user/)
- NumPy for Matlab users (http://www.scipy.org/NumPy_for_Matlab_Users)
- SciPy Reference Guide (http://docs.scipy.org/doc/scipy/reference/)
- Matplotlib gallery (http://matplotlib.sourceforge.net/gallery.html)
- ‘LeVeque’s class notes

  Inline interpreted text or phrase reference start-string without end-string.
- <http://faculty.washington.edu/rjl/classes/am583s2013/notes/python.html>‘_
- Langtangen’s book (http://folk.uio.no/hpl/scripting/) and Introductory slides (http://heim.ifi.uio.no/ hpl/scripting/all-nosplit/)

Notebooks

See *IPython notebook examples* (page 53).

Sage

Sage (http://www.sagemath.org/) is an open source mathematics software collection with its own interface and notebook system. It is based on Python and the full distribution contains in particular the Python modules needed for Clawpack.

You can try Sage directly from the web without downloading, click on “Try Sage Online” link on the Sage (http://www.sagemath.org/) webpage.

We are working on incorporating Pyclaw into Sage as an optional package.

3.1.6 IPython notebook examples

The IPython notebook (http://ipython.org/notebook.html) is a very nice platform for illustrating Clawpack examples.

If you have used Clawpack with the IPython notebook, please send us a link or submit a pull request to the apps repository (http://github.com/clawpack/apps). The links below will take you to the nbviewer site, where you can view the notebooks as html. You can also play animations in them and interact with some plots, but to actually run the code yourself you should click the download link at the top-right of each nbviewer page.

You can find demonstrations of how to set up animated results in some of the notebooks below. Source for all of these notebooks can also be found here (https://github.com/clawpack/apps/tree/master/notebooks).

Examples using PyClaw:

- A quick introduction to PyClaw (http://nbviewer.ipython.org/8332861)
- A 2D fluid dynamics example (http://nbviewer.ipython.org/8333043)
- Stegotons: solitary waves arising in non-dispersive periodic media (http://nbviewer.ipython.org/gist/ketch/8554686)
- Demonstration of different limiters for advection (http://nbviewer.ipython.org/gist/ketch/9508222)
- More to come

Examples using the Fortran code:

- Demo of running the code and producing an animation of results (http://nbviewer.ipython.org/gist/rjleveque/8328720)
- Demo of AMRClaw with adjustment of runtime and plotting parameters (http://nbviewer.ipython.org/gist/rjleveque/8642159)
- Riemann solutions of the shallow water equations (http://nbviewer.ipython.org/gist/rjleveque/8994740)
- More to come

Examples for GeoClaw:

- IPython notebook illustrating topotools (http://nbviewer.ipython.org/url/clawpack.github.io/notebooks/topotools_examples.ipynb)
- IPython notebook illustrating dtopotools (http://nbviewer.ipython.org/url/clawpack.github.io/notebooks/dtopotools_examples.ipyn
- IPython notebook illustrating the Okada model (http://nbviewer.ipython.org/url/clawpack.github.io/notebooks/Okada.ipynb)

3.1.7 Clawpack Makefiles

Makefiles for the Fortran code in many repositories use the common Makefile found in *$CLAW/clawutil/src/Makefile.common*, so you must have the *clawutil* repository.

In most directories with a *Makefile* you can type:

```
$ make help
```

to find out what options are available.

Applications directory Makefiles

output

In applications directories, compiling and running the code can usually be accomplished via:

```
$ make .output
```

This checks dependencies using the data of the hidden file *.output* that is created after the code has successfully run. If any Fortran codes have been modified since this date, the code is first recompiled. If the *setrun.py* script has been changed more recently, then the data files are first recreated.

If you want to re-run the code and you get:

```
$ make .output
make: '.output' is up to date.
```

then you can force it to run again by removing the file *.output*:

```
$ rm -f .output
$ make .output
```

This happens for example if you changed something that you know will affect the output but that isn't in the Makefile's set of dependencies, or if the code bombed or was aborted before completion.

The hidden file `.output` contains a single line, which is the path to the directory where the output resides (as specified by the `CLAW_outdir` variable in the `Makefile`). This file is used by the interactive plotting routines, as described in *Visclaw Plotting options* (page 247).

You can also do:

```
$ make output
```

(with no dot before `output`) to run the code without checking dependencies. This is sometimes handy but note that if you modify the `setrun` function and then do `make output`, it will not use the new parameter values. You must do `make .data` to regenerate the data files used by Clawpack. This would be done automatically by `make .output`, for which `.data` is a dependency.

plots

In applications directories, plotting results computed by Clawpack can generally be accomplished via:

```
$ make .plots
```

This checks dependencies using the date of the hidden file *.plots*.

This creates a set of webpages that show the plots, as described further in *Visclaw Plotting options* (page 247). There are other interactive plotting options also described there.

Starting in 4.5.1, you can also do

    $ make plots

(with no dot before `plots`) to plot the output without checking dependencies. This insures that the code will not be run again and is sometime safer than `make .plots`, which may attempt to run the code if something appears out of date.

Variables

A number of variables are defined in the Makefiles of application directories. For example, output is directed to the subdirectory specified by the variable *OUTDIR*. To change this, simply modify the Makefile before typing "make .output". Alternatively, you can modify the variable from the command line, e.g.:

```
$ make .output OUTDIR=run1
```

to direct output to a subdirectory named *run1*.

Compiler flags

Compiler flags can be changed by modifying the *FFLAGS* variable in the Makefile. If you change compiler flags you will generally need to recompile all the Fortran files and the Makefile dependencies will not detect this. To force recompilation of all files, use the "make new" option, e.g. to recompile with the *-g* flag for debugging:

```
$ make new FFLAGS=-g
```

See *Fortran Compilers* (page 46) for more about compiler flags.

3.1.8 Application documentation

Converting README.rst to README.html

In Fortran versions of Clawpack-5, the main documentation page for an application is often found in a file named *README.rst*, which can be written using the markup language reStructured Text (http://docutils.sourceforge.net/rst.html). This is the markup language used by Sphinx (http://sphinx-doc.org/) for all of the Clawpack documentation.

The *README.rst* file can be coverted into a *README.html* file by doing:

```
python $CLAW/clawutil/src/python/clawutil/convert_readme.py
```

This is also automatically done by the command:

```
make .htmls
```

which also converts code files into html files for easy browsing. A list of code files in the directory is automatically generated by *convert_readme.py* and links inserted in *README.html*.

Converting code to html with clawcode2html

The Python script *$CLAW/clawutil/src/python/clawutil/clawcode2html.py* is invoked by "make .htmls" (in applications directories having Makefiles).

This script does minor syntax highlighting by attempting to put comments in blue and adds a header at the top.

3.1.9 Specifying classic run-time parameters in *setrun.py*

It may be useful to look at a specific example, e.g. *Sample setrun.py module for classic Clawpack* (page 61).

Note: Many parameters have changed name since Version 4.X and some new ones have been added. See *setrun_changes* for a summary.

To convert a Version 4.x *setrun.py* file to Version 5.0, see *claw46to50*.

Input

setrun takes a single argument *claw_pkg* that should be set to *classic*.

Output

rundata, an object of class *ClawRunData*, created in the setrun file with the commands:

```
from clawpack.clawutil import clawdata
rundata = clawdata.ClawRunData(claw_pkg, num_dim)
```

The *rundata* object has an attribute *rundata.clawdata* whose attributes are described below.

This section explains the parameters needed for the classic single-grid Clawpack code. Additional parameters are needed by extensions of the code. For these, see:

- AMRClaw (adaptive mesh refinement): *Specifying AMRClaw run-time parameters in setrun.py* (page 70)
- GeoClaw (geophysical flows): *Specifying GeoClaw parameters in setrun.py* (page 92)

Run-time parameters

The parameters needed in 1 space dimension (*ndim=1*) are described. In 2d and 3d there are analogous parameters in y and z required, as mentioned below.

num_dim : integer from [1,2,3]
number of space dimensions.

lower : list of floats
lower limits in the x, [y,z] directions.

upper : list of floats
upper limits in the x, [y ,z] directions.

num_cells : list of integers
The number of grid cells in the x, [y, ,z] directions.

Note that when AMR is used, *num_cells* determines the number of cells in each dimension on the coarsest Level 1 grid. Additional parameters described below determine refinement ratios to finer levels.

num_eqn : integer
Number of equations in the system (e.g. *num_eqn=1* for a scalar problem).

num_aux : integer
Number of auxiliary variables in the aux array (initialized in *setaux.f*)

capa_index : integer
Index of aux array corresponding to capacity function, if there is one.

t0 : float
Initial time, often *t0 = 0.*

restart : boolean
Currently only available in amrclaw and geoclaw.

Set True to restart a previous computation. To use this option, see *Checkpointing and restarting* (page 68). Note that a change in the *Makefile* is also required.

restart_file : str
If *restart == True* then this should be the name of the checkpoint file containing all the information needed to do a restart. This will generally be of the form *fort.chkNNNNN* where *NNNNN* is the (coarse grid) timestep from the previous computation to restart from. This file is assumed to be in the directory specified for output from this run. See *Checkpointing and restarting* (page 68) for more details.

output_style: integer
There are three possible ways to specify the output times. This parameter selects the desired manner to specify the times, and affects what other attributes are required.

- *output_style = 1* : Output at fixed time intervals.

  Requires additional parameters:

  - *num_output_times* : integer, number of output times
  - *tfinal* : float, final time
  - *output_t0* : boolean, whether to also output at initial time *t0*.

  The time steps will be adjusted to hit these times exactly. (Provided *dt_variable = True*. Otherwise *dt_initial* must divide *tfinal/num_output_times* an integer number of times.)

- *output_style = 2* : Output at specified times.

  Requires the additional parameter:

– *output_times* : list of floats, times to output (include *t0* explicitly if desired)

- *output_style = 3* : Output every so many steps. Most often used for debugging, e.g to output every time step.

  Requires additional parameters:

  – *output_step_interval* : integer, number of steps between outputs

  – *total_steps* : integer, total number of steps to take

  – *output_t0* : boolean, whether to also output at initial time *t0*.

output_format: str
Format of output. Currently the following are supported:

- *'ascii'* : the files *fort.q0000* etc. are ASCII files.
- *'binary'* : Raw binary dump. Working??
- *'netcdf'* : NetCDF format. Working??

output_q_components: list of booleans or str

- A list such as *[1,0,1]* would indicate to output *q[0]* and *q[2]* only. *This might not be working yet.*
- The string *'all'* indicates that all components should be output
- The string *'none'* indicates that no components should be output

output_aux_components: list of booleans or str

- A list such as *[1,0,1]* would indicate to output *aux[0]* and *aux[2]* only. *This might not be working yet.*
- The string *'all'* indicates that all components should be output
- The string *'none'* indicates that no components should be output

output_aux_onlyonce: boolean
If *output_aux_components* is not *'none'* or an empty list, this indicates whether *aux* arrays should be only output at time *t0* or at every output time. The latter is generally necessary for AMR applications unless the grids never change (and the component of *aux* are never modified except in *setaux*).

verbosity: integer >= 0
A line of output (reporting t, dt and CFL number) is written to the terminal every time step, but only at Level *verbosity* or coarser.

Set to 0 to suppress all such output.

dt_initial: float >= 0.
Initial time step to try in first step. If using *dt_variable == True* and are unsure of an appropriate timestep, set to a very small value (e.g. *1.e-10*). After the first step the wave speeds observed in all Riemann solutions will be used to set the time step appropriately for the next step.

dt_variable: boolean
If True, time steps are adjusted automatically based on the desired Courant number *cfl_desired*.

If False, fixed time steps of lenght *dt_initial* are used.

dt_max: float >= 0.
If *dt_variable = True* then this is an upper bound on the allowable time step regardless of the Courant number. Useful if there are other reasons to limit the time step (e.g. stiff source terms).

cfl_desired: float >= 0.
If *dt_variable = True* then this is the desired Courant number. Time steps will be adjusted based on the maximum

wave speed seen in the *last* time step taken. For a nonlinear problem this may not result in the Courant number being exactly the desired value in the next step.

Usually *cfl_desired = 0.9* or less.

cfl_max: float
If *dt_variable = True* then this is the maximum Courant number that can be allowed. If a time step results in a Courant number that is greater than *cfl_desired* but less than or equal to *cfl_max*, the step is accepted. If the Courant number is greater than *cfl_max* then the step is rejected and a smaller step is taken. (At this point the maximum wave speed from Riemann solutions is known, so the step can be adjusted to exactly hit the desired value *cfl_desired*.)

Note: With AMRClaw it is impossible to retake a step and so if *cfl* > *cfl_max* then a warning message is printed and the computation continues. *Note that results may be contaminated if the Courant number is much above 1.* This means that with AMR it is important to choose an appropriate time step *dt_initial* for the first time step, or use a very small value.

Usually *cfl_max = 1.0* is fine, e.g. 500000.

steps_max: int
Maximum number of time steps allowed between output times. This is just to avoid infinite loops and generally a large value is fine.

order : int
order == 1 : Use Godunov's method

order == 2 : Use second order corrections with limiters in normal direction.

dimensional_split : str
dimensional_split == 'unsplit' is the only option currently allowed for AMRClaw.

transverse_waves : int or str
transverse_waves == 0 or 'none' : No transverse correction terms (Donor cell upwind if also *order == 1*).

transverse_waves == 1 or 'increment' : Only the increment waves are transmitted transversely. (Corner transport upwind if also *order == 1*, should be second order accurate if *order == 2*).

transverse_waves == 2 or 'all' : Corner tranpsort of second order corrections as well. (Somewhat improved stability.)

num_waves : int
Number of waves the Riemann solver returns.

limiter : list of int or str, of length num_waves
Each element of the list can take the values:

- 0 or 'none' : no limiter (Lax-Wendroff)
- 1 or 'minmod' : minmod
- 2 or 'superbee' : superbee
- 3 or 'mc' : monotonized central (MC) limiter
- 4 or 'vanleer' : van Leer

See Chapter 6 of [LeVeque-FVMHP] (page 292) for details.

use_fwaves : boolean
If True, the Riemann solvers should return f-waves (a decomposition of the the flux difference) rather than the usual waves (which give a decomposition of the jump in Q between adjacent states). See *wp_fwave*, *riemann_fwave* and Section 16.4 of [LeVeque-FVMHP] (page 292) or [BaleLevMitRoss02] (page 291) for details.

source_split : list of int or str, of length num_waves
Determines form of fractional step algorithm used to apply source terms (if any). Source terms must be implemented by providing a subroutine *srcN.f* (in *N* space dimensions) that is called each time step and should advance the solution by solving the source term equations (the PDE after dropping the hyperolic terms). See *Using src for source terms* (page 50).

- src_split == 0 or 'none' : no source term (*srcN* routine never called)
- src_split == 1 or 'godunov' : Godunov (1st order) splitting used,
- src_split == 2 or 'strang' : Strang (2nd order) splitting used.

The Strang splitting requires calling the source term routine twice each time step (before and after the hyperbolic step, with half the time step) and is generally not recommended. It is often no more accurate thn the Godunov splitting, requires more work, and can make it harder to properly set ghost cells for boundary conditions.

num_ghost : int
number of ghost cells at each boundary. Should be at least 1 if *order == 1* and at least 2 if *order == 2*.

bc_lower : list of int or str, of length num_ghost
Choice of boundary conditions at the lower boundary in each dimension. Each element can take the following values:

- 0 or 'user' : user specified (must modify *bcNamr.f* to use this option)
- 1 or 'extrap' : extrapolation (non-reflecting outflow)
- 2 or 'periodic' : periodic (must specify this at both boundaries)
- 3 or 'wall' : solid wall for systems where q(2) is normal velocity

If the value is 0 or 'user', then the user must modify the boundary condition routine *bcNamr.f* to fill ghost cells in the desired manner. See *Boundary conditions* (page 65) for more details.

bc_upper : list of int or str, of length num_ghost
Choice of boundary conditions at the upper boundary in each dimension. The same choices are available as for *bc_lower*.

Note that if periodic boundary conditions are specified at the lower boundary in some dimension then the same should be specified at the upper.

checkpt_style :: int
Currently only available in amrclaw and geoclaw.

Specify how often checkpoint files should be created that can be used to restart a computation. See *Checkpointing and restarting* (page 68) for more details.

- *checkpt_style = 0* : Do not checkpoint at all
- *checkpt_style = 1* : Checkpoint only at the final time.
- *checkpt_style = 2* : Specify a list of checkpoint times.

  This is generally **not** recommended because time steps will be adjusted to hit the checkpoint times, but may be useful in order to create a checkpoint file just before some event of interest (e.g. when debugging a code that is known to crash at a certain time).

  Requires additional parameter:

  – checkpt_times : list of floats
- *checkpt_style = 3* : Specify an interval for checkpointing.

  Requires additional parameter:

–checkpt_interval : int

Checkpoint every *checkpt_interval* time steps on Level 1 (coarsest level).

3.1.10 Sample *setrun.py* module for classic Clawpack

Warning: Need to update link? Add 2d example?

This sample *setrun.py* script is from the example in *$CLAW/classic/tests/advection*.

```
"""
Module to set up run time parameters for Clawpack.

The values set in the function setrun are then written out to data files
that will be read in by the Fortran code.

"""

import clawpack.clawutil.clawdata


#------------------------------
def setrun(claw_pkg='Classic'):
#------------------------------

    """
    Define the parameters used for running Clawpack.

    INPUT:
        claw_pkg expected to be "Classic4" for this setrun.

    OUTPUT:
        rundata - object of class ClawRunData

    """

    assert claw_pkg.lower() == 'classic',  "Expected claw_pkg = 'classic'"

    rundata = clawpack.clawutil.clawdata.ClawRunData(pkg=claw_pkg, num_dim=1)

    #------------------------------------------------------------------
    # Problem-specific parameters to be written to setprob.data:
    #------------------------------------------------------------------

    probdata = rundata.new_UserData(name='probdata',fname='setprob.data')
    probdata.add_param('u',     1.0,  'advection velocity')
    probdata.add_param('beta', 200.,  'Gaussian width parameter')


    #------------------------------------------------------------------
    # Standard Clawpack parameters to be written to claw.data:
    #------------------------------------------------------------------

    clawdata = rundata.clawdata  # initialized when rundata instantiated

    # ---------------
    # Spatial domain:
```

```
# ---------------

# Number of space dimensions:
clawdata.num_dim = 1

# Lower and upper edge of computational domain:
clawdata.lower[0] = 0.0
clawdata.upper[0] = 1.0

# Number of grid cells:
clawdata.num_cells[0] = 100



# ---------------
# Size of system:
# ---------------

# Number of equations in the system:
clawdata.num_eqn = 1

# Number of auxiliary variables in the aux array (initialized in setaux)
clawdata.num_aux = 0

# Index of aux array corresponding to capacity function, if there is one:
clawdata.capa_index = 0



# -------------
# Initial time:
# -------------

clawdata.t0 = 0.0


# Restart from checkpoint file of a previous run?
# Note: If restarting, you must also change the Makefile to set:
#    RESTART = True
# If restarting, t0 above should be from original run, and the
# restart_file 'fort.chkNNNNN' specified below should be in
# the OUTDIR indicated in Makefile.

clawdata.restart = False               # True to restart from prior results
clawdata.restart_file = 'fort.chk00006'  # File to use for restart data


# -------------
# Output times:
#--------------

# Specify at what times the results should be written to fort.q files.
# Note that the time integration stops after the final output time.
# The solution at initial time t0 is always written in addition.

clawdata.output_style = 1

if clawdata.output_style == 1:
```

```
    # Output nout frames at equally spaced times up to tfinal:
    clawdata.num_output_times = 10
    clawdata.tfinal = 1.0
    clawdata.output_t0 = True  # output at initial (or restart) time?

elif clawdata.output_style == 2:
    # Specify a list of output times.
    clawdata.tout =  [0.5, 1.0]   # used if output_style == 2
    clawdata.num_output_times = len(clawdata.tout)

elif clawdata.output_style == 3:
    # Output every iout timesteps with a total of ntot time steps:
    clawdata.output_step_interval = 1
    clawdata.total_steps = 5
    clawdata.output_t0 = True


clawdata.output_format == 'ascii'      # 'ascii' or 'netcdf'

clawdata.output_q_components = 'all'   # could be list such as [True,True]
clawdata.output_aux_components = 'none'  # could be list
clawdata.output_aux_onlyonce = True    # output aux arrays only at t0



# ---------------------------------------------------
# Verbosity of messages to screen during integration:
# ---------------------------------------------------

# The current t, dt, and cfl will be printed every time step
# at AMR levels <= verbosity.  Set verbosity = 0 for no printing.
#   (E.g. verbosity == 2 means print only on levels 1 and 2.)
clawdata.verbosity = 1



# --------------
# Time stepping:
# --------------

# if dt_variable==1: variable time steps used based on cfl_desired,
# if dt_variable==0: fixed time steps dt = dt_initial will always be used.
clawdata.dt_variable = True

# Initial time step for variable dt.
# If dt_variable==0 then dt=dt_initial for all steps:
clawdata.dt_initial = 0.8 / float(clawdata.num_cells[0])

# Max time step to be allowed if variable dt used:
clawdata.dt_max = 1e+99

# Desired Courant number if variable dt used, and max to allow without
# retaking step with a smaller dt:
clawdata.cfl_desired = 0.9
clawdata.cfl_max = 1.0

# Maximum number of time steps to allow between output times:
clawdata.steps_max = 500
```

```
    # ------------------
    # Method to be used:
    # ------------------

    # Order of accuracy:  1 => Godunov,  2 => Lax-Wendroff plus limiters
    clawdata.order = 2

    # Use dimensional splitting?
    clawdata.dimensional_split = 0

    # For unsplit method, transverse_waves can be
    #  0 or 'none'      ==> donor cell (only normal solver used)
    #  1 or 'increment' ==> corner transport of waves
    #  2 or 'all'       ==> corner transport of 2nd order corrections too
    clawdata.transverse_waves = 0

    # Number of waves in the Riemann solution:
    clawdata.num_waves = 1

    # List of limiters to use for each wave family:
    # Required:  len(limiter) == num_waves
    # Some options:
    #   0 or 'none'     ==> no limiter (Lax-Wendroff)
    #   1 or 'minmod'   ==> minmod
    #   2 or 'superbee' ==> superbee
    #   3 or 'mc'       ==> MC limiter
    #   4 or 'vanleer'  ==> van Leer
    clawdata.limiter = ['mc']

    clawdata.use_fwaves = False    # True ==> use f-wave version of algorithms

    # Source terms splitting:
    #   src_split == 0 or 'none'    ==> no source term (src routine never called)
    #   src_split == 1 or 'godunov' ==> Godunov (1st order) splitting used,
    #   src_split == 2 or 'strang'  ==> Strang (2nd order) splitting used,  not recommended.
    clawdata.source_split = 'none'


    # --------------------
    # Boundary conditions:
    # --------------------

    # Number of ghost cells (usually 2)
    clawdata.num_ghost = 2

    # Choice of BCs at xlower and xupper:
    #   0 => user specified (must modify bcN.f to use this option)
    #   1 => extrapolation (non-reflecting outflow)
    #   2 => periodic (must specify this at both boundaries)
    #   3 => solid wall for systems where q(2) is normal velocity

    clawdata.bc_lower[0] = 2
    clawdata.bc_upper[0] = 2

    return rundata
```

```
    # end of function setrun
    # ----------------------


if __name__ == '__main__':
    # Set up run-time parameters and write all data files.
    import sys
    if len(sys.argv) == 2:
    rundata = setrun(sys.argv[1])
    else:
    rundata = setrun()

    rundata.write()
```

3.1.11 Boundary conditions

Boundary conditions are imposed each time step by filling ghost cells adjacent to the edge of each grid patch. See Chapter 4 of [LeVeque-FVMHP] (page 292) for more details.

Boundary conditions are set by the library routines:

- *$CLAW/classic/src/Nd/bcN.f* for the classic code (N = 1, 2, 3).
- *$CLAW/amrclaw/src/Nd/bcNamr.f* for the amrclaw code (N = 2, 3).

Several standard choices of boundary condition procedures are provided in these routines, and can be selected at each boundary by setting the input parameters *bc_lower* and *bc_upper* in each dimension (see *Specifying classic run-time parameters in setrun.py* (page 56)) to one of the following:

- 1 or 'extrap' : extrapolation (non-reflecting outflow)

  In this case values from the grid cell adjacent to the boundary are copied into all ghost cells moving in the direction normal to the boundary. This gives a fairly good approximation to a non-reflecting or outgoing boundary condition that lets waves pass out of the boundary without reflection, particularly in one space dimension. In more than one direction this is not perfect for waves that hit the boundary at an oblique angle.

- 2 or 'periodic' : periodic boundary conditions

  In this case ghost cell values are set by copying from interior cells at the opposite boundary so that periodic boundary conditions are perfectly imposed. Normally periodic boundary conditions would be imposed by setting this value for both *bc_lower* and *bc_upper* in some dimension, but this is not required.

- 3 or 'wall' : solid wall boundary conditions are imposed for systems where the second component of q is the x velocity or momentum in one dimension (and where the third component of q is also the y velocity/momentum in more dimensions, etc.) This is true, for example, if the acoustics equations are solved with components q = *(p, u, v)* or shallow water equations with q = *(h, hu, hv)*.

  In this case the normal velocity/momentum at a wall is reflected about the boundary (copied to a ghost cell from the cell equally far from the boundary on the interior side) while all other components are extrapolated.

  Reflecting boundary conditions can also often be used on a line of symmetry of a solution in order to reduce the computational domain to be only half of the physical domain.

  Note that this option does not work on a mapped grid... **Add pointer to modified version**

If none of the above boundary conditions are desired, the user can modify the subroutine *bcN* so that setting the appropriate component of *bc_lower* or *bc_upper* to 0 will execute code added by the user. In this case it is best to put the modified version of *bcN.f* in the application directory and modify the *Makefile* to point to the modified version. See *User-defined boundary conditions* (page 66) below.

Boundary conditions for adaptive refinement

When AMR is used, any interior patch edges (not at a domain boundary) are filled automatically each time step, either by copying from adjacent patches at the same level or by interpolating (in both space and time) from coarser levels if needed.

The user must still specify boundary conditions at the edges of the computational domain. The same set of choices for standard boundary conditions as described above are implemented in the library routine *bcNamr.f*, and so specifying these boundary conditions requires no change to *setrun.py* when going from Classic Clawpack to AMRClaw. However, if special boundary conditions have been implemented in a custom version of *bcN.f* then the same procedure for setting ghost cells will have to be implemented in a custom version of *bcNamr.f*. This routine is slightly more complicated than the single-grid Classic version, since one must always check whether each ghost cell lies outside the computational domain (in which case the custom boundary condition procedure must be applied) or lies within the domain (in which case ghost cell values are automatically set by the AMR code and the user *bcNamr* routine should leave these values alone.

Boundary conditions for GeoClaw

For tsunami modeling or other geophysical flows over topography the computational domain has artificial boundaries that are placed sufficiently far from the region of interest that any flow or waves leaving the domain can be ignored and there should be no incoming waves. Extrapolation boundary conditions are then appropriate. If the ocean is truncated at some point then these generally have been found to give very small spurious reflection of outgoing tsunami waves. Extroplation boundary conditions can also be used on dry land (where the depth h is zero).

In some cases reflecting boundary conditions might be more appropriate, e.g. along the walls of a wave tank.

The library routine *$CLAW/geoclaw/src/2d/shallow/bc2amr.f* is modified from the *amrclaw* version only by extrapolating the depth at the boundaries into ghost cells.

Boundary conditions for clamshell grids on the sphere

In 2D AMRClaw and GeoClaw, an additional option is available for *bc_lower* and *bc_upper* that is implemented in the library routines:

- 4 or 'sphere' : sphere boundary conditions

  Must set *bc_lower[0:2] = bc_upper[0:2] = 4* (i.e. at all 4 boundaries)

  These boundary conditions are similar to periodic boundary conditions, but for the clamshell grid introduced in [CalhounHelzelLeVeque] for solving problems on the sphere using a single logically rectangular grid. This is best envisioned by folding a rectangular piece of paper in half, gluing the edges together, and inflating to a sphere. See the animations on the website for the original paper See also [BergerCalhounHelzelLeVeque] for further examples.

User-defined boundary conditions

If none of the boundary conditions described above is suitable at one or more boundaries of the domain, then you will have to modify the library routine to implement the desired boundary condition. See Chapter 4 of [LeVeque-FVMHP] (page 292) for hints on how to specify the ghost cell values each time step.

Suppose you need to specify different boundary conditions at the boundary *xlower*, for example. Then in *setrun.py* you should set *bc_lower[0] = 0* and modify the library boundary condition routine to insert your desired boundary conditions at the point indicated in the code, where it says:

```
c     # user-specified boundary conditions go here in place of error output
```

in the section marked *left boundary*. The details of how this is done differ a bit between the classic and AMR codes and also depend on the number of space dimensions. Examine the way other boundary conditions are implemented and follow the model in your own code.

TODO: Give some hints on how things work in AMR code – must check which ghost cells extend outside the physical domain and which are filled automatically from adjacent grid patches or by interpolation from coarser patches if they are interior to the domain.

3.1.12 Output data formats

In *Specifying classic run-time parameters in setrun.py* (page 56), the format for the output data (solutions) can be specified by setting the parameter *output_style*.

To read the solution stored in these files into Python for plotting or other postprocessing purposes, utilities are provided that are described in *python_io*.

Setting *output_style = 'ascii'* gives ASCII text output. The data files can then be viewed with any standard text editor, which is particularly useful for debugging. However, ASCII files are generally much larger than is necessary to store the original data in binary form, and so when grid have many grid cells or when many output frames are saved it is often better to use some form of binary output, e.g. *Raw binary output data format* (page 68) or *NetCDF output data format* (page 68).

In AMRClaw, ASCII and binary output are both written by the library routine *valout.f*. The aux arrays are also dumped if requested, see *Output of aux arrays* (page 68).

ASCII output data format

Two output files are created at each output time (each frame). The frames are generally numbered 0, 1, 2, etc. The two files, at frame 2, for example, are called *fort.t0002* and *fort.q0002*.

fort.t0002

This file has the typical form:

```
 0.40000000E+00    time
 1                 meqn
36                 ngrids
 0                 naux
 2                 ndim
 2                 nghost
```

This file contains only 6 lines with information about the current time the number of AMR patches at this time.

In the above example, Frame 2 contains 36 patches. If you are using the classic code or PyClaw with only a single patch, then *ngrids* would be 1.

The data for all 36 patches is contained in *fort.q0002*. The data from each patch is preceeded by a header that tells where the patch is located in the domain, how many grid cells it contains, and what the cell size is, e.g.

fort.q0002

This header has the typical form:

```
1                grid_number
1                AMR_level
40               mx
40               my
0.00000000E+00   xlow
0.00000000E+00   ylow
0.25000000E-01   dx
0.25000000E-01   dy
```

This would be followed by 40*40 = 1600 lines with the data from cells (i,j). The order they are written is (in Fortran style):

```
do j = 1,my
    do i = 1,mx
        write (q(i,j,m), m=1,meqn)
```

Each line has *meqn* (change to *num_eqn*?) values, for the components of the system in this grid cell.

After the data for this patch, there would be another header for the next patch, followed by its data, etc.

In the header, *xlow* and *ylow* are the coordinates of the lower left corner of the patch, *dx* and *dy* are the cell width in *x* and *y*, and *AMR_level* is the level of refinement, where 1 is the coarsest level. Each patch has a unique *grid_number* that usually isn't needed for visualization purposes.

Raw binary output data format

The files for each frame are numbered as for the ASCII file and the *fort.t0002* file, for example, is still an ASCII file with 6 lines of metadata. There are also ASCII files such as *fort.q0002*, but these now contain only the headers for each grid patch and not the solution on each patch. In addition there are files such as *fort.b0002* that contain a raw binary dump of the data from all of the grid patches at this time, one after another. In order to decompose this data into patches for plotting, the *fort.q0002* file must be used.

Unlike the ASCII data files, the binary output files contain ghost cells as well as the interior cells (since a contiguous block of memory is dumped for each patch with a single *write* statement).

NetCDF output data format

See *pyclaw.io.netcdf* (page 219).

Output of aux arrays

Describe...

3.1.13 The mapc2p function

To appear.

3.1.14 Checkpointing and restarting

Warning: These instructions currently only apply to *amrclaw* and *geoclaw* codes.

Checkpointing a computation

In this section *clawdata* refers to the *rundata.clawdata* attribute of an object of class *ClawRunData*, as is generally set at the top of a *setrun.py* file.

The *rundata.clawdata.checkpt_style* parameter specified in *setrun.py* (see *Specifying classic run-time parameters in setrun.py* (page 56)) determines how often checkpointing is done, if at all.

See the comments in *Sample setrun.py module for classic Clawpack* (page 61) for examples.

The checkpoint files are saved in the same output directory as the solution output, with file names of the form *fort.tchkNNNNN* (a small ASCII file) and *fort.chkNNNNN* (a large binary file) where *NNNNN* is the step number on the coarsest level. These files containt all the information needed to restart the computation at this point.

Restarting a computation

To restart a computation from any point where checkpoint files have been saved, modify *setrun.py* to set:

```
clawdata.restart = True
clawdata.restart_file = 'fort.chkNNNNN'
```

where *NNNNN* is the time step number from which the restart should commence.

You should also modify the output time parameters to specify that the computation should go to a later time than the time of the restart file (which can be found in the file *fort.tchkNNNNN*).

Note the following in setting the new output times:

- The value *clawdata.t0* should generally be left to the original starting time of the computation.
- If *clawdata.output_style==1*, then *clawdata.t0* and *clawdata.tfinal* along with *clawdata.num_output_times* are used to determine equally spaced output times. Only those times greater than the restart time will be used as output times.

  If *clawdata.output_t0==True* then a time frame will be output at the restart time (not t0 in general). This may duplicate the final frame that was output from the original computation. Set *clawdata.output_t0 = False* to avoid this.
- If *clawdata.output_style==2*, then *clawdata.output_times* is a list of output times and only those times greater than or equal to the restart time will be used as output times.

Modifying the Makefile for a restart

To restart a computation, it is necessary to modify the *Makefile* as well as *setrun.py*. In the *Makefile*, set:

```
RESTART = True
```

This will ensure that the original set of output files (for times up to the restart time) and the checkpoint file will not be deleted. They will still be available in the output directory along with the newly created output files.

Output files after a restart

After running the restarted computation, the original set of output files should still be in the output directory along with a new set from the second run. Note that one output time may be repeated in two frames if *clawdata.output_t0 == True* in the restarted run.

Note that any gauge output from the first run will be overwritten by the second run. If you wish to preserve the gauge output from the first run, currently you must copy the output file *fort.gauge* to another location, say *fort.gauge1*, before doing the restart run. Then you could catenate the two to get gauge output for the entire run, e.g.:

```
cd _output
mv fort.gauge fort.gauge2
cat fort.gauge1 fort.gauge2 > fort.gauge
```

This creates a file *fort.gauge* that contains the entire gauge history.

TODO: This should be simplified.

3.2 AMRClaw: adaptive mesh refinement

3.2.1 AMRClaw

The AMRClaw version of Clawpack provides Adaptive Mesh Refinement (AMR) capabilities in 2 and 3 space dimensions. (The two-dimensional code can also be used for 1-dimensional problems, see *amrclaw_1d*.)

See also:

- *Adaptive mesh refinement (AMR) algorithms* (page 79)
- *AMR refinement criteria* (page 81)

Block-structured AMR is implemented, in which rectangular patches of the grid at level *L* are refined to level *L+1*. See *Specifying AMRClaw run-time parameters in setrun.py* (page 70) for a list of the input parameters that can be specified to help control how refinement is done. The general algorithms are described in [BergerLeVeque98] (page 291).

See *ClawPlotItem* (page 255) for a list of 2d plot types that can be used to create a *setplot* function to control plotting of two-dimensional results. Some of the attribute names start with the string *amr_*, indicating that a list of different values can be specified for each AMR level. See *Visclaw Plotting options* (page 247) and *Using setplot.py to specify the desired plots* (page 251) for more about plotting.

Python plotting tools for 3d are still under development. For now, the Matlab tools from Clawpack 4.3 can still be used, see *Plotting using Matlab* (page 268).

3.2.2 Specifying AMRClaw run-time parameters in *setrun.py*

It may be useful to look at a specific example, e.g. *Sample setrun.py module for AMRClaw* (page 73).

Note: Many parameters have changed name since Version 4.X and some new ones have been added. See *setrun_changes* for a summary.

To convert a Version 4.x *setrun.py* file to Version 5.0, see *claw46to50*.

Input

setrun takes a single argument *claw_pkg* that should be set to *amrclaw*.

Output

rundata, an object of class *ClawRunData*, created in the setrun file with the commands:

```
from clawpack.clawutil import clawdata
rundata = clawdata.ClawRunData(claw_pkg, num_dim)
```

The *rundata* object has an attribute *rundata.clawdata* whose attributes are described in *Specifying classic run-time parameters in setrun.py* (page 56).

In addition, for AMRClaw *rundata* has an attribute *rundata.amrdata* whose attributes are described below.

Run-time parameters

The parameters needed in 2 space dimensions (*ndim=2*) are described. In 3d there are analogous parameters in z required, as mentioned below.

In addition to the parameters in *rundata.clawdata* (see *Specifying classic run-time parameters in setrun.py* (page 56)), the AMR parameters that can be set are the following attributes of *rundata.amrdata*:

Special AMR parameters

amr_levels_max : int
Maximum levels of refinement to use.

refinement_ratios_x : list of int
Refinement ratios to use in the *x* direction.

Example: If *num_cells[0] = 10* and *refinement_ratios_x = [2,4]* then the Level 1 grid will have 10 cells in the x-direction, Level 2 patches will be refined by a factor of 2, and Level 3 will be refined by 4 relative to Level 2 (by 8 relative to Level 1).

refinement_ratios_y : list of int
Refinement ratios to use in the *y* direction.

refinement_ratios_t : list of int
Refinement ratios to use in time. For an explicit method, maintaining the Courant number usually requires refining in time by the same factor as in space (or the maximum of the refinement ratio in the different space directions).

Note: Rather than specifying this list, in GeoClaw it is possible to set to set *variable_dt_refinement_ratios = True* so refinement ratios in time are chosen automatically. This might be ported to AMRClaw?

aux_type : list of str, of length num_aux
Specifies the type of variable stored in each aux variable. These are used when coarsening aux arrays. Each element can be one of the following (but at most one can be 'capacity'):

- 'center' for cell-centered values (e.g. density)
- 'capacity' for a cell-centered capacity function (e.g. cell volume)
- 'xleft' for a value centered on the left edge in x (e.g. normal velocity u)
- 'yleft' for a value centered on the left edge in y (e.g. normal velocity v)

flag_richardson : boolean
Determines whether Richardson extrapolation will be used as an error estimator. If *True*, patches will be coarsened by a factor of 2 each time regridding is done and the result from a single step on the coarsened patch with double the time step will be compared to the solution after 2 steps on the original patch in order to estimate the error.

flag_richardson_tol : float
When *flag_richardson == True*, cells will be flagged for refinement if the absolute value of the estimated error exceeds this value.

When *flag_richardson == False*, this value is not used.

flag2refine : boolean
Determines whether the subroutine *flag2refine* is used to flag cells for refinement.

flag2refine_tol : float
When *flag2refine == True*, the default library version *flag2refine.f* checks the maximum absolute value of the difference between any component of q in this cell with the corresponding component in any of the neighboring cells. The cell is flagged for refinement if the maximum value is greater than this tolerance.

regrid_interval : int
The number of time steps to take on each level between regridding to the next finer level.

regrid_buffer_width : int
The number of points to flag for refining around any point flagged by error estimation or *flag2refine*. This buffer zone is to insure that waves do not leave the refined region before the next regridding and so is generally chosen based on the value of *regrid_interval*, typically to be the same value since waves can travel at most one grid cell per time step.

clustering_cutoff : float between 0 and 1
Cut-off used in clustering flagged points into rectangular patches for refinement. Clusters are chosen to minimize the number of patches subject to the constraint:

```
(# flagged pts) / (total # of cells refined) < clustering_cutoff
```

If *clustering_cutoff* is close to 1, only flagged cells will be refined, which could lead to many *1 x 1* patches.

The default value 0.7 usually works well.

verbosity_regrid : int
Additional information is printed to the terminal each time regridding is done at this level or coarser. Set to 0 to suppress regridding output.

regions : list
List of lists of the form *[minlevel,maxlevel,t1,t2,x1,x2,y1,y2]*. See *Specifying AMR regions* (page 82)

Debugging flags for additional printing

Setting one or more of these to *True* will cause additional information to be written to the file *fort.amr* in the output directory.

dprint : boolean
Print domain flags

eprint : boolean
Print error estimation flags

edebug : boolean
Print even more error estimation flags

gprint : boolean
Print grid bisection and clustering information

nprint : boolean
Print proper nesting output

pprint : boolean
Print projection of tagged points

rprint : boolean
Print regridding summary

sprint : boolean
Print space/memory output

tprint : boolean
Print time step info on each level

uprint : boolean
Print update/upbnd information

3.2.3 Sample *setrun.py* module for AMRClaw

This sample *setrun.py* module is for two-dimensional acoustics, from the example in *$CLAW/amrclaw/examples/acoustics_2d_radial*.

```
"""
Module to set up run time parameters for Clawpack.

The values set in the function setrun are then written out to data files
that will be read in by the Fortran code.

"""

import os
import numpy as np

#------------------------------
def setrun(claw_pkg='amrclaw'):
#------------------------------

    """
    Define the parameters used for running Clawpack.

    INPUT:
        claw_pkg expected to be "amrclaw" for this setrun.

    OUTPUT:
        rundata - object of class ClawRunData

    """

    from clawpack.clawutil import clawdata


    assert claw_pkg.lower() == 'amrclaw',  "Expected claw_pkg = 'amrclaw'"

    num_dim = 2
    rundata = clawdata.ClawRunData(claw_pkg, num_dim)

    #------------------------------------------------------------------
    # Problem-specific parameters to be written to setprob.data:
    #------------------------------------------------------------------

    probdata = rundata.new_UserData(name='probdata',fname='setprob.data')
```

```
probdata.add_param('rho',     1.,  'density of medium')
probdata.add_param('bulk',    4.,  'bulk modulus')

#------------------------------------------------------------------
# Standard Clawpack parameters to be written to claw.data:
#   (or to amrclaw.data for AMR)
#------------------------------------------------------------------

clawdata = rundata.clawdata  # initialized when rundata instantiated


# Set single grid parameters first.
# See below for AMR parameters.


# ---------------
# Spatial domain:
# ---------------

# Number of space dimensions:
clawdata.num_dim = num_dim

# Lower and upper edge of computational domain:
clawdata.lower[0] = -1.000000e+00          # xlower
clawdata.upper[0] = 1.000000e+00          # xupper
clawdata.lower[1] = -1.000000e+00          # ylower
clawdata.upper[1] = 1.000000e+00          # yupper

# Number of grid cells:
clawdata.num_cells[0] = 50      # mx
clawdata.num_cells[1] = 50      # my


# ---------------
# Size of system:
# ---------------

# Number of equations in the system:
clawdata.num_eqn = 3

# Number of auxiliary variables in the aux array (initialized in setaux)
clawdata.num_aux = 0

# Index of aux array corresponding to capacity function, if there is one:
clawdata.capa_index = 0


# -------------
# Initial time:
# -------------

clawdata.t0 = 0.000000


# Restart from checkpoint file of a previous run?
# Note: If restarting, you must also change the Makefile to set:
#    RESTART = True
# If restarting, t0 above should be from original run, and the
```

```
# restart_file 'fort.chkNNNNN' specified below should be in
# the OUTDIR indicated in Makefile.

clawdata.restart = False               # True to restart from prior results
clawdata.restart_file = 'fort.chk00006'  # File to use for restart data


# -------------
# Output times:
#--------------

# Specify at what times the results should be written to fort.q files.
# Note that the time integration stops after the final output time.

clawdata.output_style = 1

if clawdata.output_style==1:
    # Output ntimes frames at equally spaced times up to tfinal:
    # Can specify num_output_times = 0 for no output
    clawdata.num_output_times = 20
    clawdata.tfinal = 1.0
    clawdata.output_t0 = True  # output at initial (or restart) time?

elif clawdata.output_style == 2:
    # Specify a list or numpy array of output times:
    # Include t0 if you want output at the initial time.
    clawdata.output_times =  [0., 0.1]

elif clawdata.output_style == 3:
    # Output every step_interval timesteps over total_steps timesteps:
    clawdata.output_step_interval = 2
    clawdata.total_steps = 4
    clawdata.output_t0 = True  # output at initial (or restart) time?


clawdata.output_format = 'ascii'       # 'ascii', 'binary', 'netcdf'

clawdata.output_q_components = 'all'   # could be list such as [True,True]
clawdata.output_aux_components = 'none'  # could be list
clawdata.output_aux_onlyonce = True    # output aux arrays only at t0


# ---------------------------------------------------
# Verbosity of messages to screen during integration:
# ---------------------------------------------------

# The current t, dt, and cfl will be printed every time step
# at AMR levels <= verbosity.  Set verbosity = 0 for no printing.
#   (E.g. verbosity == 2 means print only on levels 1 and 2.)
clawdata.verbosity = 0



# --------------
# Time stepping:
# --------------

# if dt_variable==True:  variable time steps used based on cfl_desired,
```

```
# if dt_variable==False: fixed time steps dt = dt_initial always used.
clawdata.dt_variable = True

# Initial time step for variable dt.
# (If dt_variable==0 then dt=dt_initial for all steps)
clawdata.dt_initial = 1.00000e-02

# Max time step to be allowed if variable dt used:
clawdata.dt_max = 1.000000e+99

# Desired Courant number if variable dt used
clawdata.cfl_desired = 0.900000
# max Courant number to allow without retaking step with a smaller dt:
clawdata.cfl_max = 1.000000

# Maximum number of time steps to allow between output times:
clawdata.steps_max = 50000


# ------------------
# Method to be used:
# ------------------

# Order of accuracy:  1 => Godunov,  2 => Lax-Wendroff plus limiters
clawdata.order = 2

# Use dimensional splitting? (not yet available for AMR)
clawdata.dimensional_split = 'unsplit'

# For unsplit method, transverse_waves can be
#  0 or 'none'      ==> donor cell (only normal solver used)
#  1 or 'increment' ==> corner transport of waves
#  2 or 'all'       ==> corner transport of 2nd order corrections too
clawdata.transverse_waves = 2


# Number of waves in the Riemann solution:
clawdata.num_waves = 2

# List of limiters to use for each wave family:
# Required:  len(limiter) == num_waves
# Some options:
#   0 or 'none'     ==> no limiter (Lax-Wendroff)
#   1 or 'minmod'   ==> minmod
#   2 or 'superbee' ==> superbee
#   3 or 'mc'       ==> MC limiter
#   4 or 'vanleer'  ==> van Leer
clawdata.limiter = ['mc','mc']

clawdata.use_fwaves = False    # True ==> use f-wave version of algorithms

# Source terms splitting:
#   src_split == 0 or 'none'    ==> no source term (src routine never called)
#   src_split == 1 or 'godunov' ==> Godunov (1st order) splitting used,
#   src_split == 2 or 'strang'  ==> Strang (2nd order) splitting used,  not recommended.
clawdata.source_split = 0
```

```
# --------------------
# Boundary conditions:
# --------------------

# Number of ghost cells (usually 2)
clawdata.num_ghost = 2

# Choice of BCs at xlower and xupper:
#   0 or 'user'     => user specified (must modify bcNamr.f to use this option)
#   1 or 'extrap'   => extrapolation (non-reflecting outflow)
#   2 or 'periodic' => periodic (must specify this at both boundaries)
#   3 or 'wall'     => solid wall for systems where q(2) is normal velocity

clawdata.bc_lower[0] = 'extrap'   # at xlower
clawdata.bc_upper[0] = 'extrap'   # at xupper

clawdata.bc_lower[1] = 'extrap'   # at ylower
clawdata.bc_upper[1] = 'extrap'   # at yupper


# ---------------
# Gauges:
# ---------------
rundata.gaugedata.gauges = []
# for gauges append lines of the form  [gaugeno, x, y, t1, t2]
rundata.gaugedata.gauges.append([0, 0.0, 0.0, 0., 10.])
rundata.gaugedata.gauges.append([1, 0.7, 0.0, 0., 10.])
rundata.gaugedata.gauges.append([2, 0.7/np.sqrt(2.), 0.7/np.sqrt(2.), 0., 10.])

# --------------
# Checkpointing:
# --------------

# Specify when checkpoint files should be created that can be
# used to restart a computation.

clawdata.checkpt_style = 1

if clawdata.checkpt_style == 0:
    # Do not checkpoint at all
    pass

elif clawdata.checkpt_style == 1:
    # Checkpoint only at tfinal.
    pass

elif clawdata.checkpt_style == 2:
    # Specify a list of checkpoint times.
    clawdata.checkpt_times = [0.1,0.15]

elif clawdata.checkpt_style == 3:
    # Checkpoint every checkpt_interval timesteps (on Level 1)
    # and at the final time.
    clawdata.checkpt_interval = 5



# ---------------
```

```
# AMR parameters:
# ---------------

amrdata = rundata.amrdata

# max number of refinement levels:
amrdata.amr_levels_max = 3

# List of refinement ratios at each level (length at least amr_level_max-1)
amrdata.refinement_ratios_x = [2, 2]
amrdata.refinement_ratios_y = [2, 2]
amrdata.refinement_ratios_t = [2, 2]


# Specify type of each aux variable in clawdata.auxtype.
# This must be a list of length num_aux, each element of which is one of:
#   'center',  'capacity', 'xleft', or 'yleft'  (see documentation).
amrdata.aux_type = []


# Flag for refinement based on Richardson error estimater:
amrdata.flag_richardson = False    # use Richardson?
amrdata.flag_richardson_tol = 0.001000e+00  # Richardson tolerance

# Flag for refinement using routine flag2refine:
amrdata.flag2refine = True      # use this?
amrdata.flag2refine_tol = 0.2 # tolerance used in this routine
# User can modify flag2refine to change the criterion for flagging.
# Default: check maximum absolute difference of first component of q
# between a cell and each of its neighbors.

# steps to take on each level L between regriddings of level L+1:
amrdata.regrid_interval = 2

# width of buffer zone around flagged points:
# (typically the same as regrid_interval so waves don't escape):
amrdata.regrid_buffer_width  = 2

# clustering alg. cutoff for (# flagged pts) / (total # of cells refined)
# (closer to 1.0 => more small grids may be needed to cover flagged cells)
amrdata.clustering_cutoff = 0.7

# print info about each regridding up to this level:
amrdata.verbosity_regrid = 0


# ---------------
# Regions:
# ---------------
rundata.regiondata.regions = []
# to specify regions of refinement append lines of the form
#  [minlevel,maxlevel,t1,t2,x1,x2,y1,y2]


#  ----- For developers -----
# Toggle debugging print statements:
amrdata.dprint = False      # print domain flags
amrdata.eprint = False      # print err est flags
```

```
    amrdata.edebug = False      # even more err est flags
    amrdata.gprint = False      # grid bisection/clustering
    amrdata.nprint = False      # proper nesting output
    amrdata.pprint = False      # proj. of tagged points
    amrdata.rprint = False      # print regridding summary
    amrdata.sprint = False      # space/memory output
    amrdata.tprint = False      # time step reporting each level
    amrdata.uprint = False      # update/upbnd reporting

    return rundata

    # end of function setrun
    # ----------------------


if __name__ == '__main__':
    # Set up run-time parameters and write all data files.
    import sys
    rundata = setrun(*sys.argv[1:])
    rundata.write()
```

3.2.4 Adaptive mesh refinement (AMR) algorithms

The basic adaptive refinment strategy used in *AMRClaw* (page 70) is to refine on logically rectangular patches. A single Level 1 grid covers the entire domain (usually — if it is too large it may be split into multiple Level 1 grids). Some rectangular portions of this grid are covered by Level 2 grids refined by some refinement factor R in each direction (anisotropic refinement is now allowed too — see *Specifying AMRClaw run-time parameters in setrun.py* (page 70)). Regions of each Level 2 grid may be covered by Level 3 grids, that are further refined (perhaps with a different refinement ratio). And so on.

For the hyperbolic solvers in Clawpack the time step is limited by the Courant number (see Section *cfl*), and so if the spatial resolution is refined by a factor of R in each direction then the time step will generally have to be reduced by a factor R as well.

The AMR code thus proceeds as follows:

- In each time step on the Level 1 grid(s), the values in all grid cells (including those covered by finer grids) are advanced one time step. Before this time step is taken, ghost cells around the boundary of the full computational domain are filled based on the boundary conditions specified in the library routine *bcNamr.f* (where N is the number of space dimensions). Check the *Makefile* of an application to see where this file can be found.
- After a step on the Level 1 grid, R time steps must be taken on each Level 2 grid, where R denotes the desired refinement ratio in time from Level 1 to Level 2.

  For each of these time step, ghost cell values must be filled in around all boundaries of each Level 2 grid. This procedure is defined below in *Ghost cells and boundary conditions for AMR* (page 80).
- After taking R steps on Level 2 grids, values on the Level 1 grid are updated to be consistent with the Level 2 grids. Any cell on Level 1 that is covered by a Level 2 grid has its q value replaced by the average of all the Level 2 grid cells lying within this cell. This gives a cell average that should be a better approximation to the true cell average than the original value.
- The updating just described can lead to a change in the total mass calculated on the Level 1 grid. In order to restore global conservation, it is necessary to do a conservation fix up. (To be described...)

This style of AMR is often called *Berger-Oliger-Colella* adaptive refinement, after the papers of Berger and Oliger [BergerOliger84] (page 291) and [BergerColella89] (page 291).

The Fortran code in $CLAW/amrclaw is based on code originally written by Marsha Berger for gas dynamics, and merged in Clawpack in the early days of Clawpack development by MJB and RJL. The algorithms used in AMRClaw are described more fully in [BergerLeVeque98] (page 291).

Ghost cells and boundary conditions for AMR

Consider a Level $k > 1$ grid for which we need ghost cells all around the boundary at the start of each time step on this level. The same procedure is used at other levels.

- Some Level k grids will be adjacent to other Level k grids and so any ghost cell that is equivalent to a Level k cell on some other grid has values copied from this this grid.
- Some ghost cells will be in the interior of the full computational domain but in regions where there is no adjacent Level k grid. There will be a Level k-1 grid covering that region, however. In this case the ghost cells are obtained by space-time interpolation from values on the Level k-1 grid.
- Some ghost cells will lie outside the full computational domain, where the boundary of the Level k grid lies along the boundary of the full domain. For these cells the subroutine *bcNamr* (where *N* is the number of space dimensions) is used to fill ghost cell values with the proper user-specified boundary conditions, unless periodic boundary conditions are specified (see below).

For many standard boundary conditions it is not necessary for the user to do anything beyond setting appropriate parameters in *setrun.py* (see *Specifying classic run-time parameters in setrun.py* (page 56)). Only if user-specified boundary conditions are specified is it necessary to modify the library routine *bcNamr.f* (after copying to your application directory so as not to damage the library version, and modifying the *Makefile* to point to the new version).

There some differences between the *bcNamr.f* routine and the *bcN.f* routine used for the single-grid classic Clawpack routines (which are found in *$CLAW/classic/src/Nd/bcN.f*). In particular, it is necessary to check whether a ghost cell actually lies outside the full computational domain and only set ghost cell values for those that do. It should be clear how to do this from the library version of the routine.

If **periodic boundary conditions** are specified, this is handled by the AMRClaw software along with all internal boundaries, rather than in *bcNamr.f*. With AMR it is not so easy to apply periodic boundary conditions as it is in the case of a single grid, since it is necessary to determine whether there is a grid at the same refinement level at the opposite side of the domain to copy ghost cell values from, and if so which grid and what index corresponds to the desired location.

Choosing and initializing finer grids

Every few time steps on the coarsest level it is generally necessary to revise modify the regions of refinement at all levels, for example to follow a propagating shock wave. This is done by

1. Flagging cells that need refinement according to some criteria.
2. Clustering the flagged cells into rectangular patches that will form the new set of grids at the next higher level.
3. Creating the new grids and initializing the values of *q* and also any *aux* arrays for each new grid.

Clustering is done using and algorithm developed by Berger and Rigoutsis [BergerRigoutsis91] (page 291) that finds a nonoverlapping set of rectangles that cover all flagged points and balances the following conflicting goals:

- Cover as few points as possible that are not flagged, to reduce the number of grid cells that must be advanced in each time step.
- Create as few new grids as possible, to minimize the overhead associated with filling ghost cells and doing the conservation fix-up around edges of grids.

A parameter *cutoff* can be specified (see *Specifying AMRClaw run-time parameters in setrun.py* (page 70)) to control clustering. The algorithm will choose the grids in such a way that at least this fraction of all the grid points in all the

new grids will be in cells that were flagged as needing refinement. Usually *cutoff = 0.7* is used, so at least 70% of all grid cells in a computation are in regions where they are really needed.

Initializing the new grids at Level k+1 is done as follows:

- At points where there was already a Level k+1 grid present, this value is copied over.
- At points where there was not previously a Level k+1 grid, bilinear interpolation is performed based on the Level k grids.

Flagging cells for refinement

The user can control the criteria used for flagging cells for refinement.

See *AMR refinement criteria* (page 81) for details.

3.2.5 AMR refinement criteria

Several parameters controlling refinement can be set in the *setrun* function. See *Specifying AMRClaw run-time parameters in setrun.py* (page 70) for further description of these. Many of the parameters discussed below are attributes of *rundata.amrdata* in *setrun.py*.

Every *regrid_interval* time steps on each level, the error is estimated in all cells on grids at this level. Cells where some refinement criteria are satisfied are flagged for refinement. Default options for flagging are described below. Additional cells surrounding the flagged cells are also flagged to insure that moving features of the solution (e.g. shock waves) do not escape from the region of refinement before the next regridding time. The number of buffer cells flagged is specified by *regrid_buffer_width* and the number of steps between regridding on each level is specified by *regrid_interval*. Typically these are equal (assuming the Courant number is close to 1) and taken to be some small integer such as 2 or 3.

In addition to flagging individual cells based on the behavior of the solution, it is also possible to specify that certain regions of the domain should always be refined to a certain level (and/or never refined above some level). This is described further in *Specifying AMR regions* (page 82). These regions are used in conjunction with the methods described below to determine whether or not a given cell should be flagged for refinement.

The cells that have been flagged are then clustered into rectangular regions to form grids at the next finer level. The clustering is done in light of the tradeoffs between a few large grids (which usually means refinement of many additional cells that were not flagged) or many small grids (which typically results in fewer fine grid cells but more grids and hence more overhead and less efficient looping over shorter rows of cells). The parameter *clustering_cutoff* in amrNez.data is used to control this tradeoff. At least this fraction of the fine grid cells should result from coarse cells that were flagged as needing refinement. The value clustering_cutoff = 0.7 is usually reasonable.

Flagging criteria

Two possible approaches to flagging individual cells for refinement (based on the behavior of the solution) are built into AMRClaw. (A different default approach is used in GeoClaw, see *Flagging criteria in GeoClaw* (page 83)).

flag2refine

One approach to flagging cells for refinement (the default used in most examples) is to set *flag2refine == True* and specify a tolerance *flag2refine_tol*. This indicates that the library subroutine *$CLAW/amrclaw/src/Nd/flag2refine.f90* should be used to flag cells for refinement. This routine computes the maximum max-norm of the undivided difference

of $q_{i,j}$ based its four neighbors in two space dimensions (or 6 neighbors in 3d). If this is greater than the specified tolerance, then the cell is flagged for refinement (subject to limitations imposed by "regions"). The undivided difference (not divided by the mesh width) is used, e.g. $|q(m, i+1, j) - q(m, i-1, j)|$ for each component m.

Note that the user can change the criterion used for flagging cells by modifying this routine – best done by copying the library routine to your application directory and modifying the *Makefile* to point to the modified version.

Richardson extrapolation

The second approach to flagging individual cells is based on using Richardson extrapolation to estimate the error in each cell. This is used if *flag_richardson == True*. In this case a cell is flagged if the error estimate exceeds the value *flag_richardson_tol*. Richardson estimation requires taking two time steps on the current grid and comparing the result with what's obtained by taking one step on a coarsened grid. One time step on the fine grid is re-used, so only one additional time step on the fine grid and one on a coarsened grid are required. It is somewhat more expensive than the *flag2refine* approach, but may be more useful for cases where the solution is smooth and undivided differences do not identify the regions of greatest error.

Note: Both approaches can be used together: if *flag2refine == True* and *flag_richardson == True* then a cell will be flagged if either of the corresponding specified tolerances is exceeded.

Specifying AMR regions

In addition to specifying a tolerance or other criteria for flagging individual cells as described above, it is possible to specify regions of the domain so that all points in the region, over some time interval also specified, will be refined to at least some level *minlevel* and at most some level *maxlevel*. These are specified through the parameter *rundata.regiondata.regions* in *setrun.py*. This is a list of lists, each of which specifies a region. A new region can be added via:

```
rundata.regiondata.regions.append([minlevel,maxlevel,t1,t2,x1,x2,y1,y2])
```

This indicates that over the time period from *t1* to *t2*, cells in the rectangle *x1* <= *x* <= *x2* and *y1* <= *y* <= *y2* should be refined to at least *minlevel* and at most *maxlevel*.

To determine whether a grid cell lies in one of the regions specified, the center of the grid cell is used. If a mapped grid is being used, the limits for the regions should be in terms of the computational grid coordinates, not the physical coordinates.

If a cell center lies in more than one specified region, then the cell will definitely be flagged for refinement at level L (meaning it should be covered by a Level L+1 grid) if *L+1* <= *minlevel* for any of the regions, regardless of whether the general flagging criteria hold or not. This means the smallest of the various *minlevel* parameters for any region covering this point will take effect. Conversely it will **not** be flagged for refinement if *L+1* > *maxlevel* for **all** regions that cover this point. This means the largest of the various *maxlevel* parameters for any region covering this point will take effect. (However, note that since flagged cells are buffered as described above by flagging some adjacent cells, a cell may still end up flagged for refinement even if the above tests say it should not be.)

For example, suppose that *amr_levels_max = 6* has been specified along with these two regions:

```
rundata.regiondata.regions.append([2, 5, 10.0, 30.0, 0.0, 0.5, 0.0, 0.5])
rundata.regiondata.regions.append([3, 4, 20.0, 40.0, 0.2, 1.0, 0.2, 1.0])
```

The first region specifies that from time 10 to 30 there should be at least 2 levels and at most 5 levels of refinement for points in the spatial domain *0 < x < 0.5* and *0 < y < 0.5*.

The second region specifies that from time 20 to 40 there should be at least 3 level and at most 4 levels of refinement for points in the spatial domain *0.2 < x < 1.0* and *0.2 < y < 1.0*.

Note that these regions overlap in both space and time, and in regions of overlap the *maximum* of the *minlevel* and also the *maximum* of the *maxlevel* parameters applies. So in the above example, from time 20 to 30 there will be at least 3 levels and at most 5 levels in the region of overlap, $0.2 < x < 0.5$ and $0.2 < y < 0.5$.

Within these regions, how many levels are chosen at each point will be determined by the *error flagging criteria*, i.e. by the default or user-supplied routine *flag2refine* (page 81), or as determined by *Richardson extrapolation* (page 82), as described above.

Points that are not covered by either region are not constrained by the regions at all. With *amr_levels_max = 6* then they might be refined to any level from 1 to 6 depending on the error flagging criteria.

It is easiest to explain how this works by summarizing the implementation:

The regridding algorithm from level L to L+1 loops over all grid cells at Level L and flags them or not based on the following criteria, where *(xc,yc)* represents the cell center and *t* is the current regridding time:

- Initialize the flag by applying the error flagging criteria specified by Richardson extrapolation and/or the default or user-supplied routine *flag2refine* to determine whether this cell should be flagged.
- Loop over all regions (if any) for which *(xc,yc,t)* lies in the region specified.

TODO: This might be wrong!!!

- If *L* >= *maxlevel* for *any* such region, set *flag* = *False* for this cell and go on to the next cell.
- If *L* < *minlevel* for *every* such region, set *flag* = *True* and go on to the next grid cell.

Flagging criteria in GeoClaw

In GeoClaw, a special *flag2refine* subroutine is defined.

TODO: need to describe geoclaw flag2refine.

3.2.6 Gauges

With AMRClaw in two space dimensions and GeoClaw it is possible to specify gauge locations as points (x,y) where the values of all components of q should be output every time step during the computation over some time range (t1,t2).

Still need to add to 3d AMRClaw code, and to Classic codes.

Gauges are useful in several ways, e.g.:

1. To compare computational results to measurements from physical gauges such as a pressure gauge or tide gauge that record data as a function of time at a single point,
2. To better visualize how the solution behaves at a single point,
3. To better compare results obtained with different methods or grid resolutions. Comparing two-dimensional pcolor or contour plots can be difficult whereas comparing to curves that give the solution as a function of time often reveals more clearly differences in accuracy or nonphysical oscillations.

Gauge parameters in setrun

See also *Specifying AMRClaw run-time parameters in setrun.py* (page 70).

Gauges are specified in *setrun* by adding lists of gauge data for each desired gauge to the *ClawRunData* object *rundata.gaugedata.gauges*. This is initialized as an empty list and new gauges can be specified by:

```
rundata.gaugedata.gauges.append([gaugeno, x, y, t1, t2])
```

with values

- *gaugeno* : integer

  the number of this gauge

- *x, y* : floats

  the location of this gauge

- *t1, t2* : floats

  the time interval over which gauge data should be output.

During the computation the value of all components of q at all gauge locations will be output to a single file *fort.gauge* in the output directory. Lines of this file have the form:

```
gaugeno  level  t  q[0]  q[1] ...  q[meqn-1]
```

where level is the AMR level used to determine the q values at this time. Internally the finest level available at each gauge is used, with bilinear interpolation to the gauge locations from the 4 nearest cell centers.

If you wish to change what is output at these points, you should copy the library routine *dumpgauge.f* to your own directory and modify it appropriately.

Warning: When doing a restart, previous gauge output is deleted unless you are careful to preserve it. See *Output files after a restart* (page 69).

Plotting tools

Several Python plotting tools are available to plot the gauge data, so you do not have to parse the file *fort.gauge* yourself.

If you want to read in the data for a particular gauge to manipulate it yourself, you can do, for example:

```
from clawpack.visclaw.data import ClawPlotData
plotdata = ClawPlotData()
plotdata.outdir = '_output'    # set to the proper output directory
gaugeno = 1                    # gauge number to examine
g = plotdata.getgauge(gaugeno)
```

Then:

- *g.t* is the array of times,
- *g.q* is the array of values recorded at the gauges (*g.q[m,n]* is the *m'th variable at time 't[n]*)

In the *setplot* Python script you can specify plots that are to be done for each gauge, similar to the manner in which you can specify plots that are to be done for each time frame. For example, to plot the component q[0] at each gauge, include in *setplot* lines of this form:

```
plotfigure = plotdata.new_plotfigure(name='q[0] at gauges', figno=300, \
                type='each_gauge')

# Set up for axes in this figure:
plotaxes = plotfigure.new_plotaxes()
plotaxes.xlimits = 'auto'
plotaxes.ylimits = [-1.5, 1.5]
plotaxes.title = 'q[0]'
```

```
# Plot q[0] as blue line:
plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
plotitem.plot_var = 0
plotitem.plotstyle = 'b-'
```

Note that *plotdata.new_plotfigure* is called with *type='each_gauge'* which denotes that this plot is to be produced for each gauge found in *setgauges.data*. (When type is not specified, the default is *type='each_frame'* for time frame data).

If you type:

```
$ make .plots
```

then html files will be created for the gauge plots along with the time frame plots and will all show up in the index (usually in *_plots/_PlotIndex.html*).

When using Iplotclaw to interactively view plots, try:

```
PLOTCLAW> plotgauge 1
```

to produce the plot for gauge 1, or simply:

```
PLOTCLAW> plotgauge
```

to loop through all gauges. If you rerun the code without re-executing *Iplotclaw*, you can refresh the gauge data via:

```
PLOTCLAW> cleargauges
```

You can of course specify more than one plotitem on each plotaxes if you want. For example to plot the each gauge from the current run as a blue line and the same gauge from some previous run (perhaps with a different grid resolution) as a red line, you could add the following lines to the above example:

```
# Plot q[0] from previous run as red line:
plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
plotitem.plot_var = 0
plotitem.plotstyle = 'r-'
plotitem.outdir = '_output_from_previous_run'
```

Plotting gauge locations

It is often convenient to plot the locations of the gauges on pcolor or contour plots each time frame. You can do this as follows, for example:

```
plotfigure = plotdata.new_plotfigure(name='pcolor', figno=0)
plotaxes = plotfigure.new_plotaxes('pcolor')
plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
# set other attributes as desired

def addgauges(current_data):
    from pyclaw.plotters import gaugetools
    gaugetools.plot_gauge_locations(current_data.plotdata, \
         gaugenos='all', format_string='ko', add_labels=True)

plotaxes.afteraxes = addgauges
```

You can replace *gaugenos='all'* by *gaugenos=[1,2]* or other list of specific gauges to plot. The *format_string* above specifies a black dot at each gauge location and *add_labels=True* means that the gauge number will appear next to each gauge.

If you want more control over this plotting you can of course copy the function *plot_gauge_locations* from *pyclaw.plotters.gaugetools.py* to your setplot.py file and modify at will.

Examples

Several of the examples found in *$CLAW/amrclaw/examples/* and *$CLAW/geoclaw/examples/* contain the specification of gauges.

3.3 GeoClaw: geophysical flows

3.3.1 GeoClaw

See www.geoclaw.org (http://www.geoclaw.org) for more overview of the GeoClaw software and links to references and uses.

Warning: As with all of Clawpack, this code is provided as a research and teaching tool with no guarantee of suitability for any particular purpose, and no liability on the part of the authors. See the *License* (page 13) for more details and *Cautionary Hints on using GeoClaw* (page 87) for tips on exercising appropriate care in using the code.

See also:

- *Cautionary Hints on using GeoClaw* (page 87)
- *Topography data* (page 89)
- *Python tools for working with topo and dtopo* (page 91)
- *Specifying GeoClaw parameters in setrun.py* (page 92)
- *Plotting routines for GeoClaw* (page 96)
- *Quick start guide for tsunami modeling* (page 96)
- *Earthquake sources: Fault slip and the Okada model* (page 104)
- *Setting sealevel* (page 105)
- *Manning friction term* (page 106)
- *Fixed grid monitoring* (page 106)
- *Some sources of tsunami data* (page 110)
- Links (http://depts.washington.edu/clawpack/geoclaw/) to relevant papers and sample codes (some are based on the Clawpack 4.x version of GeoClaw).

Overview

The *$CLAW/geoclaw* directory contains a specialized version of some Clawpack and AMRClaw routines that have been modified to work well for certain geophysical flow problems.

Currently the focus is on 2d depth-averaged shallow water equations for flow over varying topography. The term *bathymetry* is often used for underwater topography (sea floor or lake bottom), but in this documentation and in the code the term *topography* is often used to refer to either.

A primary concern with such flows is handling the margins of the flow where the depth goes to 0, for example at the shore line. In GeoClaw this is handled by letting the depth variable h in the shallow water equations be 0 in some cells.

Robust Riemann solvers are used that allow for dry cells adjacent to wet cells and that allow wetting and drying, for example as a tsunami inundates dry land.

Some sample calculations can be viewed in the *Gallery of GeoClaw applications* (page 33). More will eventually appear in the *Clawpack Applications repository* (page 41).

Running a GeoClaw code

Setting up, running, and plotting a GeoClaw application follows the same pattern as other AMRClaw applications, which in turn use many of the same conventions as the classic single grid Clawpack code, in particular:

- Setting parameters is done in *setrun.py*, as for other versions of Clawpack, as described in *Specifying classic run-time parameters in setrun.py* (page 56). However, there are several new parameters that may or must be set for GeoClaw. See *Specifying GeoClaw parameters in setrun.py* (page 92) for more details on these.
- The program can be compiled and run using *make* and *make .output* as for other versions, see *Fortran version* (page 45).
- Plots of results can be created either as a set of webpages via *make .plots* or interactively using *Iplotclaw*. See *Visclaw Plotting options* (page 247) for more details. Some additional Python plotting tools that are useful for GeoClaw output (e.g. plotting land and water with different colormaps) are described in the section *Plotting routines for GeoClaw* (page 96).

Topography

To simulate flow over topography it is of course necessary to specify the topography. This is usually done by providing one or more files of surface elevation (relative to some reference, e.g. sea level) at a set of points on a rectangular grid (with x-y locations in Cartesian units or in latitude-longitude, depending on the application).

Several file formats are recognized by GeoClaw. See *Topography data* (page 89) for more information on how to specify topography and some on-line resources for obtaining topography.

Plotting GeoClaw results

GeoClaw results can be plotted with the usual Python plotting tools (see *Visclaw Plotting options* (page 247)).

Some special tools and colormaps are available, see *GeoClaw plotting tools* (page 267).

Setting up a new example

- *Quick start guide for tsunami modeling* (page 96)

3.3.2 Cautionary Hints on using GeoClaw

As with all of Clawpack, the GeoClaw code is provided as a research and teaching tool with no guarantee of suitability for any particular purpose, and no liability on the part of the authors. See the *License* (page 13) for more details.

The authors believe that GeoClaw can be used for some real-world modeling of geophysical hazards, but it is the responsibility of the user to fully understand the model and its limitations and validate it for the intended purpose.

Tsunami hazard modeling

GeoClaw is currently in use for tsunami hazard assessment by several research groups. Version 4.6.1 of the code was approved in 2012 by the US National Tsunami Hazard Mitigation Program (NHTMP (http://nthmp.tsunami.gov/)) for use in modeling work supported by the program, after an extensive benchmarking project, the results of which can be found on the NTHMP benchmarking page (http://www.clawpack.org/links/nthmp-benchmarks/index.html).

However, users who wish to apply GeoClaw to the real world should be aware that doing so properly requires a good understanding of the capabilities and limitations of the code, the equations they model, and the suitability of using these equations to model any particular real-world scenario.

The authors of this code have invested considerable time in learning about appropriate aspects of geohazard modeling, through reading the literature and working directly with geoscientists who are domain experts. Even so we are very cautious in using any results from GeoClaw without performing sensitivity studies, grid refinement studies, etc., and if possible comparing results with those obtained by other modeling groups and confirming with experts that the results are reasonable. It is impossible to encapsulate the knowledge needed to deal with all the inaccuracies and uncertainties of geohazard modeling in any piece of software or its documentation, and there is no replacement for extensively reading the literature and working with domain experts.

It is also important to understand the various parameters in GeoClaw and if necessary experiment with different settings and perform sensitivity studies. See *Specifying GeoClaw parameters in setrun.py* (page 92).

Here are a few of the things that should be considered in any GeoClaw simulation:

- The depth-averaged shallow water equations are a fairly good model for the fluid dynamics of tsunamis provided the wave length is long relative to the depth of the water. In particular, for large tsunamis generated by subduction zone earthquakes propagating over the ocean, these equations may be adequate. However, even then, they are only an approximation. More accurate depth-averaged equations such as Boussinesq equations that include dispersive terms may be more accurate.
- For short wavelength tsunamis such as those generated by landslides, shallow water equations are less accurate since dispersive terms can be very important. Incorporating dispersive terms in GeoClaw is planned for the future but not yet available. These limitations should be clearly understood.
- GeoClaw solves the nonlinear shallow water equations and can capture turbulent bore formation to some extent via the formation of shock waves. It does not model wave breaking directly and in the nearshore region the use of depth-averaged equations may be inaccurate since the flow becomes fully three-dimensional. Reasonable agreement with observations from historic events and wave tank experiments have been seen in validation studies, both of GeoClaw and other shallow water codes, but caution is required.
- The empirical Manning formulation is used to model bottom friction, as described further in the section *Manning friction term* (page 106), where some limitations are discussed.
- For most tsunami simulations including the Coriolis terms in the momentum equations makes little difference in the observed results and so these terms are often turned off for efficiency (*coriolis_forcing = False*).
- The geoclaw parameter *sea_level* determines the initial fluid depth relative to the topography, as specified by the *topo* files. It is important to know what vertical datum (http://tidesandcurrents.noaa.gov/datum_options.html) the topography is relative to. Coastal bathymetry developed for tsunami modeling is often relative to Mean High Water (MHW) at some point, in which case setting *sea_level = 0.* corresponds to assuming the water level being initially at MHW. See *sea_level* for more information.
- Tsunami modeling generally requires specifying a seafloor displacement in order to initiate the tsunami, by specifying a *dtopo* file. This may be a time-dependent displacement, as explained in *dtopo*. However, it is important to understand that any displacement of the seafloor causes the entire water column above this point to be shifted upwards by the same amount (since the depth h is held constant), and so is immediately observed in the sea surface elevation. In reality, displacement of the seafloor leads to the propagation of acoustic waves that result in a surface displacement

3.3.3 Topography data

The *GeoClaw* (page 86) software for flow over topography requires at least one topo file to be input, see *Specifying GeoClaw parameters in setrun.py* (page 92).

Currently topo files are restricted to three possible formats as ASCII files. A future project is to allow other formats including NetCDF.

In the descriptions below it is assumed that the topo file gives the elevation of the topography (relative to some reference level) as a value of z at each (x,y) point on a rectangular grid. Only uniformly spaced rectangular topo grids are currently recognized.

More than one topo file can be specified (see *Topography data file parameters* (page 93)) that might cover overlapping regions at different resolutions. The union of all the topo files should cover the full computational domain specified (and may extend outside it). Internally in *GeoClaw* (page 86) a single piecewise-bilinear function is constructed from the union of the topo files, using the best information available in regions of overlap. This function is then integrated over computational grid cells to obtain the single topo value in each grid cell needed when solving depth averaged equations such as the shallow water equations with these finite volume methods. Note that this has the feature that if a grid cell is refined at some stage in the computation, the topo used in the fine cells have an average value that is equal to the coarse cell value. This is crucial in maintaining the ocean-at-rest steady state, for example.

The recognized topotypes are:

topotype = 1

x,y,z values on each line, progressing from upper left (NW) corner across rows (moving east), then down in standard GIS form. The size of the grid and spacing between the grid points is deduced from the data.

Example: if you want a flat bottom at B = -1000. over a domain 0. <= x <= 10. and 20. <= y <= 30. then the topo file could be simply:

```
0.  30.  -1000.
10. 30.  -1000.
0.  20.  -1000.
10. 20.  -1000.
```

These files are larger than necessary since they store the x,y values at each point even though the points are required to be equally spaced. Many data sets come this way, but note that you can convert a file of this type to one of the more compact types below using *clawpack.geoclaw.topotools.converttopotype(inputfile, outputfile, topotypein=1, topotypeout=2, nodata_value=None)*.

topotype = 2

The file starts with a header consisting of 6 lines containing:

```
mx
my
xllcorner
yllcorner
cellsize
nodataval
```

and is followed by mx*my lines containing the z values at each x,y, again progressing from upper left (NW) corner across rows (moving east), then down in standard GIS form. The lower left corner of the grid is *(xllcorner, yllcorner)* and the distance between grid points in both x and y is *cellsize*. The value *nodataval* indicates what value of z is specified for missing data points (often something like 9999. in data sets with missing values).

Example: For the same example as above, the topo file with topotype==2 would be:

```
2          mx
2          my
0.         xllcorner
20.        yllcorner
10.        cellsize
9999.      nodatavalue
-1000.
-1000.
-1000.
-1000.
```

topotype = 3

The file starts with a header consisting of 6 lines as for *topotype=2*, followed by *my* lines, each containing *mx* values for one row of data (ordered as before, so the first line of data is the northernmost line of data, going from west to east).

Example: For the same example as above, the topo file with topotype==3 would be:

```
2          mx
2          my
0.         xllcorner
20.        yllcorner
10.        cellsize
9999.      nodatavalue
-1000.  -1000.
-1000.  -1000.
```

It is also possible to specify values -1, -2, or -3 for *topotype*, in which case the z values will be negated as they are read in (since some data sets use different convensions for positive and negative values relative to sea level).

For *GeoClaw* (page 86) applications in the ocean or lakes (such as tsunami modeling), it is generally assumed that *sea_level = 0* has been set in *Specifying GeoClaw parameters in setrun.py* (page 92) and that $z<0$ corresponds to subsurface bathymetry and $z>0$ to topograpy above sea level.

Downloading topography files

The example $CLAW/examples/tsunami/chile2010 is set up to automatically download topo files via:

```
$ make topo
```

See the *maketopo.py* file in that directory.

Other such examples will appear in the future.

Several on-line databases are available for topograpy, see *Some sources of tsunami data* (page 110) for some links.

Some Python tools for working with topography files are available, see *Python tools for working with topo and dtopo* (page 91).

Topography displacement files

Warning: Some problems have recently been observed when trying to specify time-varying topography with *dtopo* files. Nearly instantaneous displacement occuring at the start seems to work ok, but slowly varying displacement does not always work well when AMR is also being used. A better version of this code is currently being developed, but for now use with caution!
This has been fixed in Clawpack 5.1.0.

For tsunami generation a file *dtopo* is generally used to specify the displacement of the topography relative to that specified in the topo files.

Currently two formats are supported for this file:

dtopotype=1:

Similar to topo files with *topotype=1* as described above, except that each line starts with a *t* value for the time, so each line contains t,x,y,dz

The x,y,dz values give the displacement dz at x,y at time t. It is assumed that the grid is uniform and that the file contains mx*my*mt lines if mt different times are specified for an mx*my grid.

dtopotype=3:

Similar to topo files with *topotype=3* as described above, but the header is different, and contains lines specifying *mx, my, mt, xlower, ylower, t0, dx, dy*, and *dt*. These are followed by *mt* sets of *my* lines, each line containing *mx* values of *dz*.

The Okada model can be used to generate *dtopo* files from fault parameters, as described in *Earthquake sources: Fault slip and the Okada model* (page 104).

Note that if the topography is moving, it is important to insure that the time step is small enough to capture the motion. Starting in Version 5.1.0, there is a new parameter that can be specified in *setrun.py* to limit the size time step used during the time when topography is moving. See *Topography data file parameters* (page 93).

qinit data file

Instead of (or in addition to) specifying a displacement of the topography it is possible to specify a perturbation to the depth, momentum, or surface elevation of the initial data. This is generally useful only for tsunami modeling where the initial data specified in the default *qinit.f90* function is the stationary water with surface elevation equal to *sea_level* as set in *setrun.py* (see *Specifying GeoClaw parameters in setrun.py* (page 92)).

Of course it is possible to copy the *qinit.f90* function to your directory and modify it, but for some applications the initial elevation may be given on grid of the same type as described above. In this case file can be provided as described at *qinit data file parameters* (page 94) containing this perturbation.

The file format is similar to what is described above for *topotype=1*, but now each line contains *x,y,dq* where *dq* is a perturbation to one of the components of *q* as specified by the value of *iqinit* specified (see *qinit data file parameters* (page 94)). If *iqinit = 4*, the value *dq* is instead the surface elevation desired for the initial data and the depth *h* (first component of *q*) is set accordingly.

3.3.4 Python tools for working with topo and dtopo

This describes new tools added in Clawpack 5.2.1

- ***topotools_module* - Tools for working with topo files**
  - IPython notebook illustrating topotools (http://nbviewer.ipython.org/url/clawpack.github.io/notebooks/topotools_examp
- ***dtopotools_module* - Tools for working with dtopo files**
  - IPython notebook illustrating dtopotools (http://nbviewer.ipython.org/url/clawpack.github.io/notebooks/dtopotools_exa
  - IPython notebook illustrating the Okada model (http://nbviewer.ipython.org/url/clawpack.github.io/notebooks/Okada.ipy
- *geoclaw_util_module* - Other utility functions
- *kmltools_module* - Other utility functions

3.3.5 Specifying GeoClaw parameters in *setrun.py*

Since *GeoClaw* (page 86) is a modified version of *AMRClaw* (page 70), all of the parameters that are required for AMRClaw are also needed by GeoClaw. See *Specifying AMRClaw run-time parameters in setrun.py* (page 70) for a discussion of these, and *Specifying classic run-time parameters in setrun.py* (page 56) for a description of *setrun.py* input scripts more generally.

In addition, a number of other parameters should be set in the *setrun.py* file in any *GeoClaw* (page 86) application. See also the *Cautionary Hints on using GeoClaw* (page 87) for more about parameter choices.

It is best to look at a specific example while reading this section, for example in one of the subdirectories of *$CLAW/geoclaw/examples/tsunami*.

The function *setrun* in this module is essentially the same as for AMRClaw, except that it expects to be called with *claw_pkg = 'geoclaw'*. This call should be performed properly by the Makefile if you have *CLAW_PKG = geoclaw* set properly there.

The new section *setrun_setgeo* in this module contains the new GeoClaw parameters.

A brief summary of these:

Additional AMR parameters

In addition to the standard AMRClaw parameters described in *Specifying AMRClaw run-time parameters in setrun.py* (page 70), some additional parameters governing how refinement is done should be specified for GeoClaw applications:

rundata.refinement_data.variable_dt_refinement_ratios : bool
: The default is False, in which case refinement factors in time are specified by the user as usual in the array *rundata.amrdata.refinement_ratios_t*.

  When True, this indicates that GeoClaw should automatically choose refinement factors in time on each level based on an estimate of the maximum wave speed on all grids at this level. For most hyperbolic problems the CFL condition suggests that one should refine in time by the same factor as in space. However, for GeoClaw applications where fine grids appear only in shallow coastal regions this may not be the case.

rundata.refinement_data.wave_tolerance : float
: Cells are flagged for refinement if the difference between the surface elevation and sea level is larger than this tolerance. Note that whether refinement is actually done depends also on how various AMR regions have been set (see Section *regions*) and also on several other attributes described below that contain information on minimum and maximum refinement allowed in various regions.

rundata.refinement_data.max_level_deep : int
: For simulations over the ocean, it is often useful to specify a *maximum refinement level* allowed in deep parts of the ocean. This is useful if a high level of refinement is specified on some rectangular region but only the parts of this region near the shore actually need to be refined.

rundata.refinement_data.max_level_deep : float
: The deepness that triggers the refinement limitation imposed by *max_level_deep* above.

General geo parameters

rundata.geo_data has the following additional attributes:

gravity : float
: gravitational constant in m/s**2, e.g. *gravity = 9.81*.

coordinate_system : integer
coordinate_system = 1 for Cartesian x-y in meters,

coordinate_system = 2 for latitude-longitude on the sphere.

earth_radius : float
radius of the earth in meters, e.g. *earth_radius = 6367.5e3*.

coriolis_forcing : bool
coriolis_forcing = True to include Coriolis terms in momentum equations

coriolis_forcing = False to omit Coriolis terms (usually fine for tsunami modeling)

sea_level : float
sea level (often *sea_level = 0.*) This is relative to the 0 vertical datum of the topography files used. It is important to set this properly for tsunami applications, see *Setting sealevel* (page 105).

friction_forcing : bool
Whether to apply friction source terms in momentum equations. See *Manning friction term* (page 106) for more discussion of the next three parameters.

friction_depth : float
Friction source terms are only applied in water shallower than this, i.e. if $h <$ *friction_depth*, assuming they have negligible effect in deeper water.

manning_coefficient : float or list of floats
For friction source terms, the Manning coefficient. If a single value is given, this value will be used where ever h < friction_depth. If a list of values is given, then the next parameter delineates the regions where each is used based on values of the topography B.

manning_break : list of floats
If manning_coefficient is a list of length N, then this should be a monotonically increasing list of length N-1 giving break points in the topo B used to determine where each Manning coefficient is used.

For example, if

```
manning_coefficient = [0.025, 0.06]
manning_break = [0.0]
```

then 0.025 will be used where B<0 and 0.06 used where B>0. (Subject still to the restriction that no friction is applied where h >= friction_depth.)

Topography data file parameters

See *Topography data* (page 89) for more information about specifying topography (and bathymetry) data files in GeoClaw.

rundata.topo_data.topofiles : list of lists
topofiles should be a list of the form *[file1info, file2info, etc.]* where each element is itself a list of the form

[topotype, minlevel, maxlevel, t1, t2, fname]

with values

topotype : integer

1,2 or 3 depending on the format of the file (see *Topography data* (page 89)).

minlevel : integer

the minimum refinement level that should be enforced in the region covered by this grid (for times between *t1* and *t2*).

maxlevel : integer

the maximum refinement level that should be allowed in the region covered by this grid (for times between *t1* and *t2*).

t1, t2 : floats

the time interval over which refinement should be controlled.

fname : string

the name of the topo file.

For more about controlling AMR in various regions, see *regions*.

rundata.dtopo_data.dtopofiles : list of lists

Information about topography displacement files, giving perturbations to topography generated by an earthquake, for example.

dtopofiles should be a list of the form *[]* or *[file1info]* where each element (currently at most 1 is allowed!) is itself a list of the form

[dtopotype, minlevel, maxlevel, fname]

with values

dtopotype : integer

1 or 3 depending on the format of the file (see *Topography displacement files* (page 90)).

minlevel : integer

the minimum refinement level that should be enforced in the region covered by this grid.

maxlevel : integer

the maximum refinement level that should be allowed in the region covered by this grid.

fname : string

the name of the dtopo file. See *Topography displacement files* (page 90) for information about the format of data in this file.

rundata.dtopo_data.dt_max_dtopo : float

the maximum time step allowed during the time interval over which the topography is moving. This is assumed to start at time *t0* and to extend to the maximum time that any of the dtopo files specified is active. This avoids issues where the time step selected by the CFL condition is much larger than the time scale over which the topography changes. You must also set *rundata.clawdata.dt_initial* to the same value (or smaller) to insure that the first time step is sufficiently small.

qinit data file parameters

A modification to the initial data specified by default can be made as described at *qinit data file* (page 91).

iqinit : integer

Specifies what type of perturbation is stored in the *qinitfile*, see *qinit data file* (page 91) for more information. Valid values for *iqinit* are

- 0 = No perturbation specified
- 1 = Perturbation to depth *h*
- 2 = Perturbation to x-momentum *hu*
- 3 = Perturbation to y-momentum *hv*

- 4 = Perturbation to surface level

qinitfiles : list of lists

qinitfiles should be a list of the form *[]* or *[file1info]* where each element (currently at most 1 is allowed!) is itself a list of the form

[minlevel, maxlevel, fname]

with values

minlevel : integer

the minimum refinement level that should be enforced in the region covered by this grid.

maxlevel : integer

the maximum refinement level that should be allowed in the region covered by this grid.

fname : string

the name of the qinitdata file. See *Topography data* (page 89) for information about the format of data in this file.

See *qinit data file* (page 91) for more details about the format.

AMR refinement region parameters

This is now a general AMRClaw parameter, still available in particular for GeoClaw applications. See *Specifying AMR regions* (page 82).

Fixed grid output parameters

fixedgrids : list of lists

This can be used to specify a set of grids where output should be produced at the specified resolution regardless of how the AMR grids look at each time. Interpolation from the best available grid near each point is used. This is useful for comparing AMR output to results obtained with other codes that use a fixed grid.

fixedgrids should be a list of the form *[grid1info, grid2info, etc.]* where each element is itself a list of the form

[t1, t2, x1, x2, y1, y2, xpoints, ypoints]

with values

t1, t2 : floats

the time interval over which output should be written for this grid.

x1, x2, y1, y2 : floats

the spacial extent of this grid.

xpoints, ypoints : floats

the number of grid points in the x and y directions (the grid will include *x1*, *x2* and *xpoints-2* points in between, for example).

ioutarrivaltimes : int

Deprecated feature. This should generally be set to 0. If you want to keep track of arrival times, it is recommended to use the new *fgmax* parameters described below.

ioutsurfacemax :

Deprecated feature. This should generally be set to 0. If you want to keep track of surface or depth maxima, it is recommended to use the new *fgmax* parameters described below.

Fixed grid maximum monitoring / arrival times

`fgmax_files : list of strings`
This can be used to specify a set of grids on which to monitor the maximum flow depth (or other quantities) observed over the course of the computation, and/or the arrival time of the flow or wave.

This works better than using the older *fixedgrids* approach since it now correctly interpolates when a grid point lies near the junction of two grid patches, which was not always handled properly before.

The "grids" also do not have to be rectangular grids aligned with the coordinate directions, but can consist of an arbitrary list of points that could also be points along a one-dimensional transect or points following a coastline, for example.

fgmax_files should be a list of strings specifying the file names of files that list the points on each grid and additional information required for each grid, as described at *Fixed grid monitoring* (page 106).

`fgmax_data.num_fgmax_val : int`
Should take the value 1, 2, or 5 and indicates how many values to monitor. See *Fixed grid monitoring* (page 106) for more details.

3.3.6 Plotting routines for GeoClaw

See *Visclaw Plotting options* (page 247) for general information about plotting Clawpack results. GeoClaw results can be viewed using these same tools. Some addition functions and useful colormaps are available in the visclaw module *geoplot*.

In particular, the following functions are useful to specify as *plot_var* attributes of a *ClawPlotItem* (page 255):

topo, land, depth, surface, surface_or_depth

The function *plot_topo_file* is useful for plotting the topography in a file of the type described in *Topography data* (page 89).

See the module for more documentation, found in the file *$CLAW/visclaw/src/python/visclaw/geoplot.py*.

3.3.7 Quick start guide for tsunami modeling

Warning: As with all of Clawpack, this code is provided as a research and teaching tool with no guarantee of suitability for any particular purpose, and no liability on the part of the authors. See the *License* (page 13) for more details.

This is a brief outline of how to set up and run GeoClaw to model a real event, with pointers to various useful data sources.

As always, the best way to get started is to copy a working example and modify it to do what you want. We'll start with the example in *$CLAW/geoclaw/examples/tsunami/chile2010*.

Copy this directory somewhere new with an appropriate name.

As an example we will convert this to a code modeling the Great Tohoku Tsunami of 11 March 2011, and will assume an environment variable has been set so that *$TOHOKU* points to the directory. This is easily done by going into this directory and typing (at the Unix prompt *$*):

```
$ export TOHOKU=`pwd`
```

(Make sure you use back ticks not quotes. This runs the *pwd* command to print the current working directory and inserts the results on the right hand side of the assignment statement.)

Get bathymetry and topography

You need one or more files that contain both bathymetry and topography on a single rectangular grid of points (generally in lat-long coordinates). Bathymetry is underwater topography. Here (and in the code) we refer to both as *topo*. In general these files have negative values of z to indicate distance below sea level for bathymetry, and positive values of z to indicate height above sea level on shore. But some files have these switched. See *Topography data* (page 89) for more details on the formats GeoClaw can handle.

You probably need two types of topo files: fairly coarse resolution over a large area of an ocean, and finer scale over one or more small regions: those where you want to model inundation and perhaps also for the source region where the tsunami was generated.

Warning: The design-a-grid website discussed below has disappeared. ETOPO1 data can now be downloaded from http://maps.ngdc.noaa.gov/viewers/wcs-client/, but it is not quite as flexible. We are working on a tool for scripting this, and this documentation page will be updated in due course.

Coarse-scale topo can be obtained from various on-line databases. The easiest is the NGDC GEODAS Design-a-grid (http://www.ngdc.noaa.gov/mgg/gdas/gd_designagrid.html) website. Simply type in the latitude and longitude of the edges of the region you need.

Choose the ETOPO 1-minute Global Relief database and then select the grid resolution you want.

Typically 4 arc-minute or even 10-minute resolution is sufficient for tsunami propagation across the ocean. Recall that 1 degree of latitude is 111 km and 1 degree of longitude is about the same at the equator, so 4 minutes is roughly 7.5 km. You can also select 1- or 2-minute data.

Finer-scale topo for much of the US coast is also available from this website; select the "US Coastal Relief Model Grids" database.

To get an idea of what region you need to get topo, it's often easiest to use Google Earth, which shows latitude and longitude.

For the Tohoku event, we'll use 10-minute topo over the region from 115E to 150W and from 15N to 50N, covering part of the north Pacific. Type these coordinates into Design-a-grid and choose the 10-minute grid and the ASCII Raster format. It should tell you this will create a grid with 211 latitude cells and 571 longitude cells. (Points would be more accurate than cells, since these are equally spaced grid points including the end points in each direction.)

Select *ASCII Raster Format* (with header).

Give your grid the name *npacific* and then click "Submit".

On the next page click on "Compress and Retrieve Your Grid". You do not need to check any of the boxes on this page.

On the next page click on "Retrieve Compressed File".

This should save a zip file called *npacific-6353.zip* (the number may be different).

Move that file into your $TOHOKU directory and then unpack it:

```
$ cd $TOHOKU
$ mv ~/Downloads/npacific-6353.zip ./  # or from whereever it is
$ unzip npacific-6353.zip
```

This unzips the file to create a directory *geodas_npacific-6353*

Most of the files contain metadata. The actual grid is the *.asc* file, the one with the longest name, so move that one to this directory:

```
$ mv geodas_npacific-6353/npacific-6353/npacific-6353.asc ./
```

The first 6 lines of the *.asc* file are the header:

```
$ head -6 npacific-6353.asc
NCOLS   571
NROWS   211
XLLCENTER  115.0000000000
YLLCENTER  15.0000000000
CELLSIZE 0.166666666667
NODATA_VALUE   -32768
```

The next 211 lines each consist of 571 values, the topo value going along one particular latitude. The first line is at northernmost latitude $50 = 15 + 210 \times 0.166666667$. The last line is at the southernmost latitude 15. On each lines the values correspond to z at points going from west to east, from longitude 115 to $210 = 115 + 570 \times 0.16666667$. Note that 210E is the same as 150W. In GeoClaw the computational domain will go from *xlower = 115* to *xupper = 210*.

This file is almost in the form required by GeoClaw (with *topotype = 3* as described at *Topography data* (page 89)). The only problem is that GeoClaw wants the numbers to appear first on the header lines, so you can delete the words before the numbers (which aren't needed), or move them to the end of the line for future reference. There's a Python script available if you have Clawpack installed:

```
$ python
>>> from clawpack.geoclaw import topotools
>>> topotools.swapheader('npacific-6353.asc','npacific.tt3')
>>> quit()
```

We've simplified the file name too in the process, so it is now called *npacific.tt3*. The extension *tt3* is to remind us that this file has the format required by *topo_type* 3.

You can now delete all the files and directory starting with *npacific-6353* unless you want to inspect the metadata:

```
$ rm -rf npacific-6353*
```

You may also need to retreive other topo files for particular regions of interest. As an example, suppose we want to look at the tsunami behavior near Hawaii.

To keep the file sizes manageable we'll just get a 1-minute data from the ETOPO1 database. To study inundation much finer bathymetry would be required.

Use design-a-grid to obtain a 1-minute grid of the area from 161W to 153W and from 18N to 23N. This gives a 481 by 301 grid.

Go through the same process as above to retrieve this grid and create a file *hawaii.tt3*.

There is one additional change that needs to be made in this file. The line

```
-1.610000000000000e+02              xlower
```

containing x at the lower-left corner must be incremented by 360 and changed to

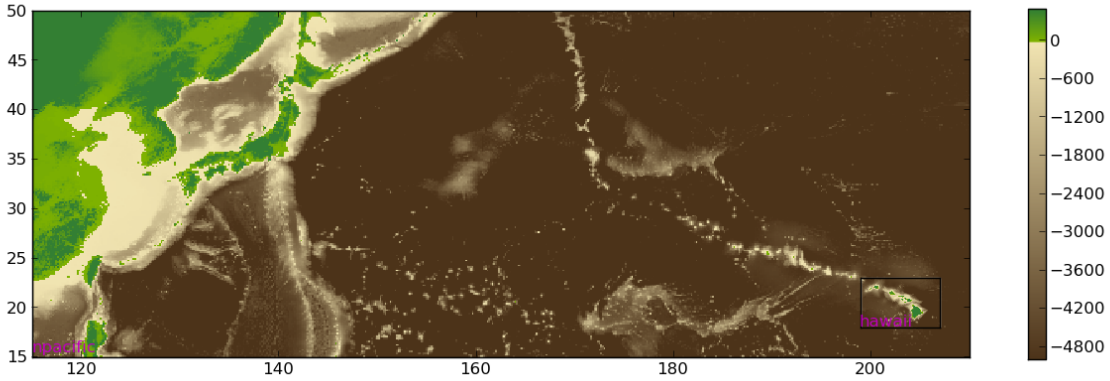
```
1.990000000000000e+02              xlower
```

Since we are using coordiates from 115E to 210E we need to specify the left longitude value in eastern hemisphere coordinates rather than western.

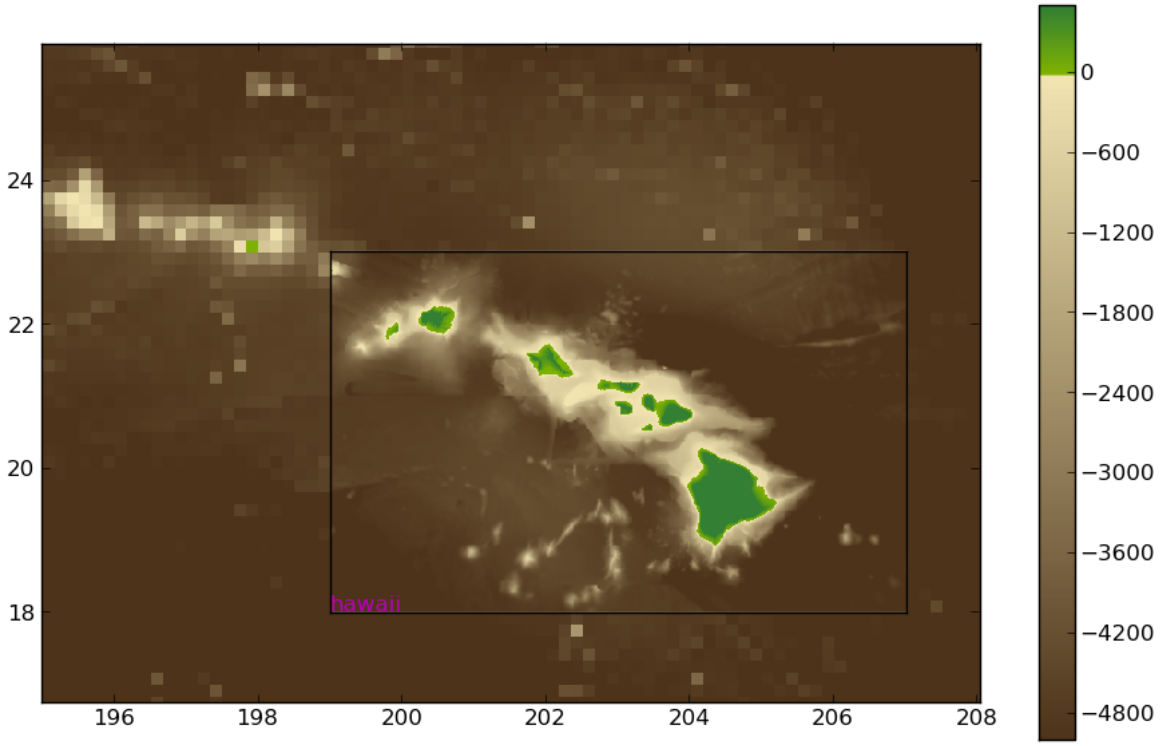
If you want to examine the grids you have obtained, you can use the following in Python:

```
from clawpack.geoclaw import topotools
import matplotlib.pyplot as plt
plt.figure(figsize=(15,6))  # create figure with appropriate aspect ratio
ax = plt.axes()

npacific = topotools.Topography('npacific.tt3', topo_type=3)
npacific.plot(axes=ax, limits=(-6000,3000))

hawaii = topotools.Topography('hawaii.tt3', topo_type=3)
hawaii.plot(axes=ax, limits=(-6000,3000), add_colorbar=False, plot_box='w')
plt.axis(npacific.extent)
```

This should give a plot similar to



Zooming in near Hawaii shows



See *topotools_module* for more information on the *Topography* class and the *plot* function.

Specifying topo in *setrun.py*

The file *setrun.py* we copied from the *chilie2010* directory contains the lines:

```
# == settopo.data values ==
topo_data = rundata.topo_data
# for topography, append lines of the form
#    [topotype, minlevel, maxlevel, t1, t2, fname]
topo_path = os.path.join(scratch_dir, 'etopo10min120W60W60S0S.asc')
topo_data.topofiles.append([2, 1, 3, 0., 1.e10, topo_path])
```

Note that this was pointing to a directory in *scratch_dir*, which is set earlier in *setrun.py* to *$CLAW/geoclaw/scratch*. This directory is set up to hold large topography files that have been downloaded. If you want to you can put the new ones you have created there, and adjust the path specified below.

We wish to change this to use the two topo files we have created. After changing the file, these lines should be

```
# == settopo.data values ==
topo_data = rundata.topo_data
# for topography, append lines of the form
#   [topotype, minlevel, maxlevel, t1, t2, fname]
topo_data.topofiles.append([3, 1, 3, 0., 1.e10, 'npacific.tt3'])
topo_data.topofiles.append([3, 1, 3, 0., 1.e10, 'hawaii.tt3'])
```

We have specified two topo files. Each file has topotype 3 and we are allowing at most 3 levels of AMR in the regions covered by each file. Later we will see how to allow more levels in specific regions.

The tsunami source – seafloor motion from an earthquake

We also need to specify how the seafloor moves, which generates the tsunami. This is specified to GeoClaw by providing a *dtopo* file as described further in the section *Topography data* (page 89). This is a file with a similar structure to a topo file but gives the displacement of the topo over some rectangular grid, possibly at a sequence of different times.

Often earthquake data is specified in the form of a set of *fault parameters* that describe the slip along a fault plane of some finite size at some depth below the seafloor. A single earthquake may be described by a collection of such fault planes. All of this subsurface slip must be combined to generate the resulting seafloor motion. Ideally this would be done by solving elastic wave equations in the three-dimensional earth, taking into acount the spatially-varying elastic parameters and the irregularity of the seafloor.

In practice, the *Okada model* is often used to translate slip along one small fault plane into motion of the seafloor. This is essentially a Greens function solution to the problem of a point dislocation in an elastic half space, so it assumes the region of slip is small, the elastic parameters in the earth are constant, and the seafloor is flat.

For more discussion of this, see *Earthquake sources: Fault slip and the Okada model* (page 104) and *dtopotools_module* for some of the available tools for applying the Okada model and creating *dtopo* files from given source parameters.

The *chile2010* example that we are starting with has a very simple fault model for the source, consisting of a single fault plane with one set of parameters specified in the file *maketopo.py* in the lines

```
usgs_subfault = dtopotools.SubFault()
usgs_subfault.strike = 16.
usgs_subfault.length = 450.e3
usgs_subfault.width = 100.e3
usgs_subfault.depth = 35.e3
usgs_subfault.slip = 15.
usgs_subfault.rake = 104.
usgs_subfault.dip = 14.
usgs_subfault.longitude = -72.668
usgs_subfault.latitude = -35.826
usgs_subfault.coordinate_specification = "top center"

fault = dtopotools.Fault()
fault.subfaults = [usgs_subfault]
```

This is a 450 km by 100 km fault plane with the length oriented at 16 degrees from north (the *Strike_Direction*). The fault plane is not horizontal but instead dips at 14 degrees from horizontal along the axis oriented with the length. The slip along this plane has a magnitude of 15 m and the slip is in the direction 104 degrees from the strike direction (the *rake*). The top of the fault plane is 35 km below the surface.

The *dtopo* file needed for GeoClaw is specified in these lines from *maketopo.py*:

```
x = numpy.linspace(-77, -67, 100)
y = numpy.linspace(-40, -30, 100)
times = [1.]

fault.create_dtopography(x,y,times)
dtopo = fault.dtopo

dtopo_fname = os.path.join(scratch_dir, "dtopo_usgs100227.tt3")
dtopo.write(dtopo_fname, dtopo_type=3)
```

A 100 by 100 grid is used to evaluate the Okada model and is stored as

Currently a good source for the Tohoku event is Preliminary Model III of the UCSB group, which can be found at http://www.geol.ucsb.edu/faculty/ji/big_earthquakes/2011/03/0311_v3/Honshu.html. Scroll to the bottom of that page and click on "SUBFAULT FORMAT".

Download this file and then... **need to describe how to create 'honshu_ucsb3.tt3'**

Other source models for the same event can be found elsewhere, often in the same format, e.g. the USGS model at http://earthquake.usgs.gov/earthquakes/eqinthenews/2011/usc0001xgp/finite_fault.php.

Specifying dtopo in *setrun.py*

The file *setrun.py* we copied from the *chilie2010* directory contains the lines:

```
# == setdtopo.data values ==
geodata.dtopofiles = []
# for moving topography, append lines of the form:
#   [topotype, minlevel,maxlevel,fname]
geodata.dtopofiles.append([1,3,3,'usgs100227.tt1'])
```

We wish to change this to use the dtopo file we created above, so these lines should be changed to:

```
# == setdtopo.data values ==
geodata.dtopofiles = []
# for moving topography, append lines of the form:
#   [topotype, minlevel,maxlevel,fname]
geodata.dtopofiles.append([3,3,3,'honshu-ucsb3.tt3'])
```

Note that we are forcing 3 levels of refinement in the region covered by the fault at the initial time. This value should be chosen to insure that the fault region has reasonable resolution. (If fewer than 3 levels of refinement are used, i.e. *mxnext* < *3*, then this will insure that as many levels as available are used in this region.)

Note: Dynamic fault motion, in which the *dtopo* file contains time-dependent displacements *dz*, is also supported. **Need to document**.

Setting other parameters in setrun.py

Several parameters can be adjusted, see *Specifying classic run-time parameters in setrun.py* (page 56) for a description of general Clawpack parameters and *Specifying GeoClaw parameters in setrun.py* (page 92) for a description of additional GeoClaw parameters.

For our test problem we will change the following:

The domain size

We choose a domain that extends from the source region to Hawaii, and that is covered by the topo files:

```
# Lower and upper edge of computational domain:
clawdata.xlower = 140.
clawdata.xupper = 210.

clawdata.ylower = 15.
clawdata.yupper = 50.
```

The coarse grid

The coarse grid will have a 2-degree resolution:

```
# Number of grid cells:
clawdata.mx = 35
clawdata.my = 15
```

The output times

For a first test, let's plot the solution every half hour for 12 hours:

```
clawdata.outstyle = 1

if clawdata.outstyle==1:
    # Output nout frames at equally spaced times up to tfinal:
    clawdata.nout = 24
    clawdata.tfinal = 3600.*12
```

AMR parameters

We will first do a fairly coarse run to get an idea of what time interval we care about near Hawaii:

```
# max number of refinement levels:
mxnest = 2

# List of refinement ratios at each level (length at least mxnest-1)
clawdata.inratx = [4]
clawdata.inraty = [4]
clawdata.inratt = [4]
```

This causes refinement by a factor of 4 in both x and y (down to 0.5 degree) and also in time (so 4 steps will be taken on the Level 2 grids for each step on Level 1).

Regions and Gauges

For the first attempt, we will not specify any refinement regions or gauges for output, so give an empty list for each of these parameters:

```
# == setregions.data values ==
geodata.regions = []
# to specify regions of refinement append lines of the form
#  [minlevel,maxlevel,t1,t2,x1,x2,y1,y2]

# == setgauges.data values ==
geodata.gauges = []
# for gauges append lines of the form  [gaugeno, x, y, t1, t2]
```

Setting gauges

Gauges can be specified where the solution is recorded at every time step. This is specified in the *setrun.py* file. The file copied from the *chile2010* example contains the lines

```
# == setgauges.data values ==
geodata.gauges = []
# for gauges append lines of the form  [gaugeno, x, y, t1, t2]
geodata.gauges.append([32412, -86.392, -17.975, 0., 1.e10])
```

Note that *geodaa.gauges* is initialized to an empty list and then a list has been appended that specifies a gauge numbered 32412 at longitude -86.392 and latitude -17.975. This is the location of DART buoy 32412 (http://www.ndbc.noaa.gov/station_page.php?station=32412) off the coast of Chile. The values of *t1* and *t2* specified means that this gauge data will be output for all times.

This location is not in our new computational domain, so this line can be deleted. We might want to add one or more lines corresponding to the locations of DART buoys or tide gauges for the new computation. Tide gauges are generally in shallow water and we would need much finer bathymetry than we are using to resolve the flow near a tide gauge.

Let's add a line for DART buoy 51407 (http://www.ndbc.noaa.gov/station_page.php?station=51407), which is near the island of Hawaii:

```
geodata.gauges.append([51407, 203.484, 19.642, 3600*7., 1.e10])
```

Note that gauge output is only requested after time *t1 = 3600*7* seconds since the tsunami doesn't reach this gauge until more than 7 hours after the earthquake (which could be determined by first doing a coarse grid simulation).

3.3.8 Earthquake sources: Fault slip and the Okada model

To initiate a tsunami from an earthquake, it is necessary to generate a model of how the seafloor moves, which is generally specified in a *dtopo* file as described in *Topography displacement files* (page 90).

Fault slip

For historic earthquakes, it is generally possible to find many different models for the distribution of slip on one or more fault planes, see for example the pointers at *Earthquake source models* (page 111).

An earthquake subfault model is typically given in the form of a set of rectangular patches on the fault plane. Each patch has a set of parameters defining the relative slip of rock on one side of the planar patch to slip on the other side. The minimum set of parameters required is:

- *length* and *width* of the fault plane (typically in m or km),
- *latitude* and *longitude* of some point on the fault plane, typically either the centroid or the center of the top (shallowest edge),
- *depth* of the specified point below the sea floor,
- *strike*, the orientation of the top edge, measured in degrees clockwise from North. Between 0 and 360. The fault plane dips downward to the right when moving along the top edge in the strike direction.
- *dip*, angle at which the plane dips downward from the top edge, a positive angle between 0 and 90 degrees.
- *rake*, the angle in the fault plane in which the slip occurs, measured in degrees counterclockwise from the strike direction. Between -180 and 180.
- *slip* > *0*, the distance (typically in cm or m) the hanging block moves relative to the foot block, in the direction specified by the rake. The "hanging block" is the one above the dipping fault plane (or to the right if you move in the strike direction).

Note that for a strike-slip earthquake, *rake* is near 0 or 180. For a subduction earthquake, the rake is usually closer to 90 degrees.

Okada model

The slip on the fault plane(s) must be translated into seafloor deformation. This is often done using the "Okada model", which is derived from a Green's function sollution to the elastic half space problem, following [Okada85] (page 294). Uniform displacement of the solid over a finite rectangular patch specified using the parameters described above, when inserted in a homogeneous elastic half space a distance *depth* below the free surface, leads to a steady state solution in which the free surface is deformed. This deformation is used as the seafloor deformation. Of course this is only an approximation since the actual seafloor in rarely flat, and the actual earth is not a homogeneous isotropic elastic material as assumed in this model. However, it is often assumed to be a reasonable approximation for tsunami

modeling, particularly since the fault slip parameters are generally not known very well even for historical earthquakes and so a more accurate modeling of the resulting seafloor deformation may not be justified.

In addition to the parameters above, the Okada model also requires an elastic parameter, the Poisson ratio, which is usually taken to be 0.25.

This IPython notebook (http://nbviewer.ipython.org/url/clawpack.github.io/notebooks/Okada.ipynb) illustrates how the Okada model works and how to generate the seafloor deformation needed in GeoClaw using this model.

The Python module *$CLAW/geoclaw/src/python/geoclaw/dtopotools.py* provides tools to convert a file specifying a collection of subfaults into a *dtopofile* by applying the Okada model to each subfault and adding the results together (valid by linear superposition of the solutions to the linear elastic halfspace problems). See *dtopotools_module* for more documentation and illustrations.

3.3.9 Setting sealevel

GeoClaw has a parameter *sealevel* (see *Specifying GeoClaw parameters in setrun.py* (page 92)) that can be used to specify the initialization of the fluid depth relative to the specified topography (see *Topography data* (page 89)). Unless a different set of initial conditions is specified (see *qinit data file parameters* (page 94)), the default is to initialize with zero velocity and depth h chosen so that $h+B$ = *sealevel* at any point where $B <$ *sealevel*, where B is the topography or bathymetry in the grid cell (as determined by interpolation from the specified *topo* files as described in *Topography data* (page 89)).

It is important to know what vertical datum (http://tidesandcurrents.noaa.gov/datum_options.html) the topography is relative to. Coastal bathymetry developed for tsunami modeling (e.g. from NOAA NGDC inundataion relief (http://www.ngdc.noaa.gov/mgg/coastal/coastal.html)) is often relative to Mean High Water (MHW), in which case setting *sealevel = 0.* corresponds to assuming the water level is initially at MHW. To adjust to use a different tide level, the value of *sealevel* must be set appropriately. The relation between MHW and other tide levels such as Mean Sea Level (MSL) can often be found from the NGDC webpages for a nearby tide gauge. For example, if you go to a station page such as Hilo Bay (http://tidesandcurrents.noaa.gov/data_menu.shtml?stn=1617760%20Hilo,%20Hilo%20Bay,%20Kuhio%20Bay,%20HI&type=Historic you will see a *Datums* link on the left menu. (Be sure to switch from feet to meters!)

Note that the difference between MHW and MSL can vary greatly between different locations. Global bathymetry data such as the ETOPO1 data (available from GEODAS Grid Translator - Design-a-Grid (http://www.ngdc.noaa.gov/mgg/gdas/gd_designagrid.html)) is generally relative to MSL. However, this data has a resolution of 1 arc-minute, more than 1.5 km, and is not suitable as coastal bathymetry, so this data will presumably only be used in grid cells away from the region where coastal bathymetry is available. Since the difference between MSL and MHW is at most a few meters, the use of different vertical datums for regions of vastly different resolution will generally have little effect.

If GeoClaw is used to compare inundation or tide gauge values to observations from past tsunamis, it may be important to know the tide stage when the largest tsunami waves arrived. Ideally it would be possible to model the actual rise and fall of the tides during the duration of the event, but this is not currently possible. Tidal currents may also have a significant effect on observed inundation patterns, but these are also ignored in GeoClaw since the water is assumed to be at rest before the tsunami arrives.

If GeoClaw is used for hazard assessment based on potential tsunami scenarios, then thought should be given to the appropriate value of *sealevel* to assume. The NGDC coastal bathymetry data is referenced to MHW because this is often the level assumed for tsunami hazard assessment, but higher tide levels such as Mean Higher High Water (MHHW) or the Astronomical High Tide (AHT) are sometimes used for worst-case scenarios.

See *Some sources of tsunami data* (page 110) for some other sources of data.

3.3.10 Manning friction term

When using GeoClaw to model inundation, it is important to include an appropriate bottom friction term in the equations. This takes the form of a source term added to the right hand side of the momentum equations:

$$(hu)_t + \cdots = -\gamma(hu),$$
$$(hv)_t + \cdots = -\gamma(hv),$$

The form built into GeoClaw is the Manning formulation, in which γ is a function of the depth and momentum:

$$\gamma = \frac{gn^2\sqrt{(hu)^2+(hv)^2}}{h^{7/3}}.$$

with g the gravitational constant and n the "Manning coefficient". This is an empirical formula and the proper value of n to use depends on the roughness of the terrain or seabed, as shown for example in this table (http://www.engineeringtoolbox.com/mannings-roughness-d_799.html). Often for generic tsunami modeling, the constant value $n = 0.025$ is used. An enhancement of GeoClaw planned for the future is to allow spatially-varying Manning coefficient.

The friction term is only applied in regions where the depth is below a threshold specified by *friction_depth* (see *Specifying GeoClaw parameters in setrun.py* (page 92)).

New in 5.0: A list of Manning coefficients can be specifed to be used in different regions based on the topography B, e.g. one value offshore and a different value onshore. See *General geo parameters* (page 92).

Warning: Changing the Manning coefficient can have a significant effect on the extent of inundation and runup. If GeoClaw (or any other code) is used for estimating real-world hazards, users should think carefully about chosing an appropriate value, and may want to run sensitivity studies. A smaller value of n (less friction) will generally lead to greater inundation.

Warning: A bug was recently discovered in GeoClaw that was corrected in Version 4.6.3: The exponent (7/3) was used in the Fortran code, which evaluates as 2 in integer arithmetic rather than 2.3333. This has now been corrected by writing it as (7.d0/3.d0). This can make a difference in the extent of inundation and runup. Given the uncertainty in the proper value of n to use and the inadequacy of using the same value everywhere, the effect of this bug on the resulting accuracy was probably small, but users may want to test this.

3.3.11 Fixed grid output

GeoClaw has the capability to output the results at specified output times on a specified "fixed grid" by interpolating from the AMR grids active at each output time. This feature is largely unchanged from Clawpack 4.6.

Describe further.

An improved version for monitoring maximum values and arrival times has been added, see *fgmax*.

3.3.12 Fixed grid monitoring

Warning: This feature has been modified and this documentation describes the version introduced in 5.2.1.

See also:

- *fgmax_tools_module* - Tools for working with fgmax files

GeoClaw has the capability to monitor certain quantities on a specified "fixed grid" by interpolating from the AMR grids active at each time step, or at specified time increments. This is useful in particular to record the maximum flow depth observed at each point over the course of a computation, or the maximum flow velocity, momentum, or momentum flux. These quantities are often of interest in hazard modeling.

It is also possible to record the *arrival time* of a flow or wave at each point on the grid.

The "grids" do not have to be rectangular grids aligned with the coordinate directions, but can consist of an arbitrary list of points that could also be points along a one-dimensional transect or points following a coastline, for example. It is also possible to specify logically rectangular grids of points covering an arbitrary quadrilateral.

Each grid is specified by an input file in a specified form described below. The list of file names for desired grids is specified in the *setrun* function, see *Fixed grid maximum monitoring / arrival times* (page 96).

This is an improved version of the algorithms used in earlier versions of GeoClaw, and now correctly interpolates when a grid point lies near the junction of two grid patches, which was not always handled properly before. The earlier version can still be used for outputing results at intermediate times on a fixed grid (see *Fixed grid output* (page 106)), but is not recommended for the purpose of monitoring maxima or arrival times.

Input file specification

(changed in Clawpack 5.2.0.)

The GeoClaw Fortran code reads in one or more files that specify grid(s) for monitoring values during the computation.

The input file(s) are specified to GeoClaw by a list of file names set in *setrun.py* by setting *rundata.fgmax_data.fgmax_files*. The order the files appear in this list determines the number assigned to this grid (starting with 1) that may be needed for processing or plotting the results.

Currently at most 5 fgmax grids are allowed by default. If you need more, you can adjust the parameter *FG_MAXNUM_FGRIDS* in *$CLAW/geoclaw/src/2d/shallow/fgmax_module.f90* and the do *make new* to recomile everything that depends on this module.

Each input file describing a grid of points has the following form:

```
tstart_max
tend_max
dt_check
min_level_check
arrival_tol
point_style
```

followed by additional lines that depend on the value of *point_style*.

If *point_style* == *0*, an arbitrary collection of *(x,y)* points is allowed and all must be listed, preceeded by the number of points:

```
npts      # number of points
x1 y1     # first point
x2 y2     # second point
...       # etc.
```

These points need not lie on a regular grid and can be specified in any order.

If *point_style* == *1*, a 1-dimensional transect of points is specified by the next three lines of the file, in the form:

```
npts       # number of points to generate
x1, y1     # first point
x2, y2     # last point
```

If *point_style* == *2*, a 2-dimensional cartesian of points is specified by the next three lines of the file, in the form:

```
nx, ny      # number of points in x and y  (nx by ny grid)
x1, y1      # lower left corner of cartesian grid
x2, y2      # upper right corner of cartesian grid
```

If *point_style == 3*, a 2-dimensional logically rectangular array of points is specified by the next five lines of the file, in the form:

```
n12, n23      # number of points along adjacent edges (see below)
x1, y1        # first corner of grid
x2, y2        # second corner of grid
x3, y3        # third corner of grid
x4, y4        # fourth corner of grid
```

The corners should define a convex quadrilateral (ordered clockwise around the perimeter). An array of points will be defined as the intersection points of two sets of lines. The first set is obtained by connecting *n12* equally spaced points on the side from *(x1,y1)* to *(x2,y2)* with the same number of points equally spaced on the side from *(x3,y3)* to *(x4,y4)*. The second set of lines is obtained by connecting *n23* equally spaced points on the side from *(x2,y2)* to *(x3,y3)* with the same number of points equally spaced on the side from *(x4,y4)* to *(x1,y1)*

The other paramters in the input file are:

- *tstart_max* : float

  starting time to monitor maximum

- *tend_max* : float

  ending time to monitor maximum

- *dt_check* : float

  time increment for monitoring maximum and arrivals. Interpolate to fixed grid and update values only if the time since the last updating exceeds this time increment. Set to 0 to monitor every time step.

- *min_level_check* : integer

  Minimum AMR level to check for updating the maximum value observed and the arrival time. Care must be taken in selecting this value since the maximum observed when interpolating to a point from a coarse AMR level may be much larger than the value that would be seen on a fine grid that better resolves the topography at this point. Often AMR "regions" are used to specify that a fine grid at some level *L* should always be used in the region of interest over the time period from *start_max* to *tend_max*, and then it is natural to set *min_level_check* to *L*.

- *arrival_tol* : float

  The time reported as the "arrival time" is the first time the value of the surface elevation is greater than *sea_level* + *arrival_tol*.

Tools to create a input file

See class *FGmaxGrid* in the *fgmax_tools_module*. The function *FGmaxGrid.write_input_data* can be used to create an input file of the form described above, and may be useful if you want to use Python to assist in setting the parameters or defining a set of points to list with *point_style == 0*.

Values to monitor

The values to be monitored are specified by the subroutine *fgmax_values*. The default subroutine found in the library *$CLAW/geoclaw/src/2d/shallow/fgmax_values.f90* is now set up to monitor the depth *h* (rather than the value *eta_tilde*

used in Version 5.1) and optionally will also monitor the speed $s = \sqrt{u^2 + v^2}$ and three other quantities (the momentum hs, the momentum flux hs^2, and $-h$, which is useful to monitor the minimum depth at each point, e.g. in a harbor where ships may be grounded).

The values monitored by the default routine described above is determined by the value of the *fgmax_module* variable *FG_NUM_VAL*, which can be set to 1, 2, or 5. This value is now read in from the data file *fgmax.data* and can be set by specifying the value of *rundata.fgmax_data.num_fgmax_val* in *setrun.py*.

Choice of interpolation procedure

The library routine *geoclaw/src/2d/shallow/fgmax_interpolate.f90* has been improved in 5.2.0 to fix some bugs. This routine does bilinear interpolation the finite volume grid centers to the fixed grid in order to update the maximum of values such as depth or velocity.

An alternative version of this routine has been added in 5.2.0 that does piecewise constant interpolation instead. This simply uses the value in the finite volume grid cell that contains the fixed grid point (0 order extrapolation) and avoids problems sometimes seen when doing linear interpolation near the margins of the flow.

This routine is in *fgmax_interpolate0.f90* and is now recommended. To use this routine, modify the *Makefile* in an application directory to replace the line

```
$(GEOLIB)/fgmax_interpolate.f90 \
```

by

```
$(GEOLIB)/fgmax_interpolate0.f90 \
```

Processing and plotting fgmax output

After GeoClaw has run, the output directory should contain the following files:

- *fort.FG1.valuemax* containing values at each fgmax grid point,
- *fort.FG1.aux1* containing the bathymetry at each fgmax grid point.

If more than one fgmax grid was specified by *rundata.fgmax_data.fgmax_files* then there will be similar files *fort.FG2.**, etc. They will be numbered in the order they appear in the list of input files.

These files are most easily dealt with using *fgmax_tools_module* by defining an object of class *fgmax_tools.FGmaxGrid* and using the class function *read_output* to read the output.

For some examples, see *apps/tsunami/chile2010_fgmax* and *apps/tsunami/bowl_radial_fgmax* in the *Clawpack Applications repository* (page 41). Sample results appear in the *Gallery of GeoClaw applications* (page 33).

TODO: Add a simple example here?

Format of the output files

The paragraphs below describe in more detail the structure of the output files for users who need to process them differently.

If *point_style == 0* for a grid then the points will be listed in the same order as specified in the input file. For other values of *point_style* (1-dimensional transects or 2-dimensional arrays) the values will be output in a natural order. In all cases the first two columns of each output file are the longitude and latitude of the point.

The remaining columns of *fort.FG1.aux1* contain the bathymetry (the first component of the *aux* array in GeoClaw) interpolated to this fgmax grid point. There will be one column for each level of AMR (up to the number specified in *setrun.py* by the parameter *amr_levels_max*). These values are initialize to *-0.99999000E+99* and only updated if

interpolation at this level is used to update a value at this particular grid point. Values at different levels may be needed to interpret the output stored *fort.FG1.valuemax*, e.g. to determine if a point is onshore or off-shore, and to compute the maximum surface elevation at a point $\eta = h + B$ from the maximum depth recorded at this point.

The file *fort.FG1.valuemax* contains the longitude and latitude of each point in columns 1 and 2. Column 3 contains the AMR level at which the maximum that is recorded was observed. (This is used to index into the array of bathymetry values from *fort.FG1.aux1* when doing computations as described in the previous paragraph).

The **last** column of *fort.FG1.valuemax* contains the arrival time of the wave at this grid point, as determined by the tolerance *arrival_tol* specified in the input file. The time reported as the "arrival time" is the first time the value of the surface elevation is greater than *sea_level* + *arrival_tol*. Points where this value is *-0.99999000E+99* never met this criterion, perhaps because the point was never inundated.

The intermediate columns of *fort.FG1.valuemax* contain the maximum observed value of a quantity such as the flow depth along with the time at which the maximum was observed. How many values are recorded depends on the setting of *rundata.fgmax_data.num_fgmax_val* in *setrun.py*:

- **If *rundata.fgmax_data.num_fgmax_val == 1*:**
  - Column 4 contains maximum value of depth *h*,
  - Column 5 contains time of maximum *h*.
- **If *rundata.fgmax_data.num_fgmax_val == 2*:**
  - Column 4 contains maximum value of depth *h*,
  - Column 5 contains maximum value of speed,
  - Column 6 contains time of maximum *h*,
  - Column 7 contains time of maximum speed.
- **If *rundata.fgmax_data.num_fgmax_val == 5*:**
  - Columns 4,5,6,7,8 contain maximum value depth, speed, momentum, momentum flux, and *hmin*, respectively,
  - Columns 9,10,11,12,13 contain times the maximum was recorded, for each value above.

3.3.13 Some sources of tsunami data

Topography / bathymetry

Note that it is important to know what elevation $B = 0$ corresponds to for each topography dataset you might use (i.e. the vertical datum (http://tidesandcurrents.noaa.gov/datum_options.html)) Global ETOPO1 bathymetry is relative to MSL (Mean Sea Level), while tsunami inundation relief is often relative to MHW (Mean High Water). These can often be combined since the difference is small relative to the resolution of the global bathymetry and the result assumed to be relative to MHW. This is important if comparing to tide gauge observation or when modeling inundation.

- GEODAS Grid Translator - Design-a-Grid (http://www.ngdc.noaa.gov/mgg/gdas/gd_designagrid.html): ETOPO 1 minute resolution of all oceans. **This has disappeared — replaced by:**
- NGDC's WCS Grid Extraction Tool (http://maps.ngdc.noaa.gov/viewers/wcs-client/)
- NOAA NGDC inundataion relief (http://www.ngdc.noaa.gov/mgg/coastal/coastal.html): High resolution data near US coastlines.

Earthquake source models

An earthquake source is typically specified by giving the slip along the fault on a set of fault planes or on subfaults making up a single plane. This data must then be converted into seafloor deformation to create the *dtopo* file needed for GeoClaw (see *Topography displacement files* (page 90)). This conversion is often done using the Okada model as described at *Earthquake sources: Fault slip and the Okada model* (page 104).

- USGS archive (http://earthquake.usgs.gov/earthquakes/eqinthenews/2012/)
- Chen Ji's archive, UCSB (http://www.geol.ucsb.edu/faculty/ji/big_earthquakes/home.html)

DART buoy data

- Information page (http://www.ngdc.noaa.gov/hazard/DARTData.shtml)
- Real-time and archived data (http://www.ndbc.noaa.gov/dart.shtml)

Tide gauges

Tide gauge data is often recorded relative to MLLW (Mean Lower-Low Water), so be sure to check the vertical datum (http://tidesandcurrents.noaa.gov/datum_options.html).

For example, if you go to a station page such as Hilo Bay (http://tidesandcurrents.noaa.gov/data_menu.shtml?stn=1617760%20Hilo,%20
you will see a *Datums* link on the left menu that gives the difference between MLLW and other water levels such as MHW, which might be the reference level for the bathymetry. (Be sure to switch from feet to meters!) Sometimes you can also select the Datum to use when retrieving data.

- NGDC (http://www.ngdc.noaa.gov/hazard/tide.shtml)
- NOAA Tides & Currents: Historic verified data (http://tidesandcurrents.noaa.gov/station_retrieve.shtml?type=Historic+Tide+Data
  ... Preliminary data (http://tidesandcurrents.noaa.gov/station_retrieve.shtml?type=Tide+Data)
- NOAA 1-minute water level data (http://tidesandcurrents.noaa.gov/1mindata.shtml) at tsunami-capable stations.
- GLOSS / SONEL (http://www.sonel.org/-Tide-gauges,29-.html?lang=en)

CHAPTER

FOUR

PYCLAW

4.1 Pyclaw

Installation:

```
pip install clawpack
```

To run an example, launch an IPython session and then:

```
from clawpack.pyclaw import examples
claw = examples.shock_bubble_interaction.setup()
claw.run()
claw.plot()
```

Features:

- A **hyperbolic PDE solver** in 1D, 2D, and 3D, including mapped grids and surfaces, built on Clawpack;
- **Massively parallel** – the same simple script that runs on your laptop will scale efficiently on the world's biggest supercomputers (see *Running in parallel* (page 183));
- **High order accurate**, with WENO reconstruction and Runge-Kutta time integration (see *Using PyClaw's solvers: Classic and SharpClaw* (page 208));
- Simple and intuitive thanks to its Python interface.

PyClaw makes use of the additional Clawpack packages, Riemann (http://github.com/clawpack/riemann) and VisClaw (http://github.com/clawpack/visclaw) for Riemann solvers and visualization, respectively.

If you have any issues or need help using PyClaw, contact us (claw-users@googlegroups.com).

4.2 PyClaw Documentation

4.2.1 PyClaw Basics

Installing PyClaw

The fastest way to install the latest release of PyClaw is:

```
pip install clawpack
```

To get the latest development version, do this instead:

```
git clone git@github.com:clawpack/clawpack.git
cd clawpack
python setup.py install
```

If you encounter any difficulties in the installation process, please contact us (claw-users@googlegroups.com) or raise an issue (http://github.com/clawpack/pyclaw/issues/).

To run an example, launch an IPython session and then:

```
from clawpack.pyclaw import examples
claw = examples.shock_bubble_interaction.setup()
claw.run()
claw.plot()
```

This will run the code and then place you in an interactive plotting shell. To view the simulation output frames in sequence, simply press 'enter' repeatedly. To exit the shell, type 'q'. For help, type '?' or see *Plotting PyClaw results* (page 173).

Dependencies: Python, gfortran, numpy, and matplotlib

PyClaw requires Python 2.7 or greater and a modern Fortran 95 compiler. PyClaw is known to work with GNU gfortran 4.2 and higher and the IBM XLF compiler.

- Python (http://python.org) version >= 2.7.
- numpy (http://numpy.scipy.org/) version >= 1.6.
- matplotlib (http://matplotlib.sourceforge.net/) version >= 1.0.1 (optional – only for plotting).
- pip (http://www.pip-installer.org/en/latest/installing.html)
- A Fortran 90 compiler, such as gfortran version >= 4.2. If you do not have one already, we recommend getting one via GCC Wiki GFortranBinaries (http://gcc.gnu.org/wiki/GFortranBinaries).

There are some additional dependencies for running in parallel; see *Running in parallel* (page 183).

Obtaining Python, numpy, and matplotlib If you don't already have these on your system, we recommend Anaconda CE (https://store.continuum.io/) or Enthought Canopy Express (https://www.enthought.com/products/epd/free/) (both free).

Clawpack

PyClaw is part of Clawpack, which includes several other packages; see *Clawpack components* (page 16). Note that the installation instructions above will install PyClaw, Riemann and VisClaw. If you also wish to use AMRClaw or GeoClaw, you should follow the more general Clawpack *Installation instructions* (page 7).

Testing your installation with nose

If you've manually downloaded the source, or cloned from Github, then you can easily test your installation. First install nose:

```
pip install nose
```

Then

```
cd clawpack/pyclaw
nosetests
```

If you have followed the instructions for *Running in parallel* (page 183), you can run the tests in parallel:

```
mpirun -n 4 nosetests
```

Note: PyClaw automatically enables tests that require PETSc if it detects a petsc4py installation. Otherwise, tests that use PETSc are disabled.

Next steps

Now you're ready to set up your own PyClaw simulation. Try the *PyClaw tutorial: Solve the acoustics equations* (page 115)!

PyClaw tutorial: Solve the acoustics equations

PyClaw is designed to solve general systems of hyperbolic PDEs of the form

$$\kappa(x)q_t + A(q,x)q_x = 0. \tag{4.1}$$

As an example, in this tutorial we'll set up a simulation that solves the acoustics equations in one dimension:

to

$$p_t + Ku_x = 0$$
$$u_t + \frac{1}{\rho}p_x = 0 \tag{4.2}$$

$$u_t + \frac{1}{\rho}p_x = 0 \tag{4.2}$$

The key to solving a particular system of equations with PyClaw or other similar codes is a Riemann solver. Riemann solvers for many systems are available as part of the clawpack/riemann package.

We'll assume that you've already followed the *Installing PyClaw* (page 113) instructions.

The following instructions show how to set up a problem step-by-step in an interactive shell. See *acoustics_1d* for the full source on which this is based.

The commands below should be typed at the Python prompt; we recommend using IPython.

```
>>> from clawpack import pyclaw
>>> from clawpack import riemann
```

The Solver

PyClaw includes various algorithms for solving hyperbolic PDEs; each is implemented in a `Solver` object. So the first step is to create a solver

```
>>> solver = pyclaw.ClawSolver1D(riemann.acoustics_1D)
```

Next we set the boundary conditions. We'll use a wall (wall) condition at the left boundary and a non-wall (zero-order extrapolation) condition at the right boundary

```
>>> solver.bc_lower[0] = pyclaw.BC.wall
>>> solver.bc_upper[0] = pyclaw.BC.extrap
```

The domain

Next we need to set up the grid. We do so by defining the physical domain and the computational resolution. We'll use the interval $(-1, 1)$ and 200 grid cells:

```
>>> domain = pyclaw.Domain([-1.0], [1.0], [200])
```

Notice that the calling sequence is similar to numpy's `linspace` command.

Finally, we set up a `Solution` object, which will hold the solution values:

```
>>> solution = pyclaw.Solution(solver.num_eqn, domain)
```

Initial condition

Now we will set the initial value of the solution

```
>>> state = solution.state
>>> xc = state.grid.p_centers[0]      # Array containing the cell center coordinates
>>> from numpy import exp
>>> state.q[0,:] = exp(-100 * (xc-0.75)**2) # Pressure: Gaussian centered at x=0.75.
>>> state.q[1,:] = 0.                       # Velocity: zero.
```

Problem-specific parameters

The acoustics equations above have some coefficients – namely, the bulk modulus K and density ρ – that must be defined. Furthermore, checking the code for the Riemann solver we've chosen reveals that it expects us to provide values for the impedance Z and sound speed c. These values are stored in a Python dictionary called problem_data that is a member of the `State` (page 224)

```
>>> from math import sqrt
>>> rho = 1.0
>>> bulk = 1.0
>>> state.problem_data['rho'] = rho
>>> state.problem_data['bulk'] = bulk
>>> state.problem_data['zz'] = sqrt(rho*bulk)
>>> state.problem_data['cc'] = sqrt(bulk/rho)
```

The controller

The most convenient way to run a PyClaw simulation is by using a `Controller` (page 206) object. The controller directs the solver in advancing the solution and handles output.

```
>>> controller = pyclaw.Controller()
>>> controller.solution = solution
>>> controller.solver = solver
>>> controller.tfinal = 1.0
```

At last everything is set up! Now run the simulation

```
>>> status = controller.run()
```

This should print out a few lines indicating the output times. It also prints the minimum and maximum tipe-step used, the number of steps required for the computation and the maximum CFL number. The simplest way to plot the solution is

```
>>> from clawpack.pyclaw import plot
>>> plot.interactive_plot()
```

That's it! Your first PyClaw simulation. Of course, we've only scratched the surface of what PyClaw can do, and there are many important options that haven't been discussed here. To get an idea, take a look through the pyclaw/examples directory and try running some other examples. It's also a good idea to get more deeply acquainted with the main *Understanding Pyclaw Classes* (page 188).

Working with PyClaw's built-in examples

PyClaw comes with many example problem scripts that can be accessed from the module *clawpack.pyclaw.examples*. If you have downloaded the PyClaw source, you can find them in the directory *clawpack/pyclaw/examples/*. These examples demonstrate the kinds of things that can be done with PyClaw and are a great way to learn how to use PyClaw.

Running and plotting examples

Interactively in IPython A built-in example can be run and plotted as follows:

```
from clawpack.pyclaw import examples
claw = examples.shock_bubble_interaction.setup()
claw.run()
claw.plot()
```

To run and plot a different example, simply replace *shock_bubble_interaction* with another example name. A number of keyword arguments may be passed to the setup function; see its docstring for details. These usually include the following:

- `use_petsc`: set to 1 to run in parallel
- `solver_type`: set to `classic` or `sharpclaw`
- `iplot`: set to 1 to automatically launch interactive plotting after running. Note that this shouldn't be used in parallel, as every process will try to plot.
- `htmlplot`: set to 1 to automatically create HTML plot pages after running.
- `outdir`: the name of the subdirectory in which to put output files. Defaults to `./_output`.

From the command line If you have downloaded the Clawpack source, you can run the examples from the command line. Simply do the following at the command prompt:

```
$ cd clawpack/pyclaw/examples/acoustics_1d_homogeneous
$ python acoustics.py iplot=1
```

You can run any of the examples similarly by going to the appropriate directory and executing the Python script. For convenience, the scripts are set up to pass any command-line options as arguments to the setup function.

List of built-in examples

You can see results from many of the examples in the *Galleries of all Clawpack applications* (page 25).

One-dimensional advection Solve the linear advection equation:

$$q_t + uq_x = 0.$$

Here q is the density of some conserved quantity and u is the velocity.

The initial condition is a Gaussian and the boundary conditions are periodic. The final solution is identical to the initial data because the wave has crossed the domain exactly once.

```
#!/usr/bin/env python
# encoding: utf-8

r"""
One-dimensional advection
=========================

Solve the linear advection equation:

.. math::
    q_t + u q_x & = 0.

Here q is the density of some conserved quantity and u is the velocity.

The initial condition is a Gaussian and the boundary conditions are periodic.
The final solution is identical to the initial data because the wave has
crossed the domain exactly once.
"""
import numpy as np
from clawpack import riemann

def setup(nx=100, kernel_language='Python', use_petsc=False, solver_type='classic', weno_order=5,
          time_integrator='SSP104', outdir='./_output'):

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if kernel_language == 'Fortran':
        riemann_solver = riemann.advection_1D
    elif kernel_language == 'Python':
        riemann_solver = riemann.advection_1D_py.advection_1D

    if solver_type=='classic':
```

```
        solver = pyclaw.ClawSolver1D(riemann_solver)
    elif solver_type=='sharpclaw':
        solver = pyclaw.SharpClawSolver1D(riemann_solver)
        solver.weno_order = weno_order
        solver.time_integrator = time_integrator
    else: raise Exception('Unrecognized value of solver_type.')

    solver.kernel_language = kernel_language

    solver.bc_lower[0] = pyclaw.BC.periodic
    solver.bc_upper[0] = pyclaw.BC.periodic

    x = pyclaw.Dimension('x',0.0,1.0,nx)
    domain = pyclaw.Domain(x)
    state = pyclaw.State(domain,solver.num_eqn)

    state.problem_data['u'] = 1.  # Advection velocity

    # Initial data
    xc = state.grid.x.centers
    beta = 100; gamma = 0; x0 = 0.75
    state.q[0,:] = np.exp(-beta * (xc-x0)**2) * np.cos(gamma * (xc - x0))

    claw = pyclaw.Controller()
    claw.keep_copy = True
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver

    if outdir is not None:
        claw.outdir = outdir
    else:
        claw.output_format = None

    claw.tfinal =1.0
    claw.setplot = setplot

    return claw

def setplot(plotdata):
    """
    Plot solution using VisClaw.
    """
    plotdata.clearfigures()  # clear any old figures,axes,items data

    plotfigure = plotdata.new_plotfigure(name='q', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.ylimits = [-.2,1.0]
    plotaxes.title = 'q'

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
    plotitem.plot_var = 0
    plotitem.plotstyle = '-o'
    plotitem.color = 'b'
    plotitem.kwargs = {'linewidth':2,'markersize':5}
```

```
    return plotdata


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Two-dimensional advection Solve the two-dimensional advection equation

$$q_t + uq_x + vq_y = 0$$

Here q is a conserved quantity, and (u,v) is the velocity vector.

```
#!/usr/bin/env python
# encoding: utf-8
r"""
Two-dimensional advection
=========================

Solve the two-dimensional advection equation

.. math::
    q_t + u q_x + v q_y & = 0

Here q is a conserved quantity, and (u,v) is the velocity vector.
"""

import numpy as np
from clawpack import riemann


def qinit(state):
    """Set initial condition for q.
       Sample scalar equation with data that is piecewise constant with
       q = 1.0  if  0.1 < x < 0.6   and   0.1 < y < 0.6
           0.1  otherwise
    """
    X, Y = state.grid.p_centers
    state.q[0,:,:] = 0.9*(0.1<X)*(X<0.6)*(0.1<Y)*(Y<0.6) + 0.1


def setup(use_petsc=False,outdir='./_output',solver_type='classic'):

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if solver_type == 'classic':
        solver = pyclaw.ClawSolver2D(riemann.advection_2D)
        solver.dimensional_split = 1
        solver.limiters = pyclaw.limiters.tvd.vanleer
    elif solver_type == 'sharpclaw':
        solver = pyclaw.SharpClawSolver2D(riemann.advection_2D)

    solver.bc_lower[0] = pyclaw.BC.periodic
    solver.bc_upper[0] = pyclaw.BC.periodic
    solver.bc_lower[1] = pyclaw.BC.periodic
```

```
    solver.bc_upper[1] = pyclaw.BC.periodic

    solver.cfl_max = 1.0
    solver.cfl_desired = 0.9

    # Domain:
    mx = 50; my = 50
    x = pyclaw.Dimension('x',0.0,1.0,mx)
    y = pyclaw.Dimension('y',0.0,1.0,my)
    domain = pyclaw.Domain([x,y])

    num_eqn = 1
    state = pyclaw.State(domain,num_eqn)

    state.problem_data['u'] = 0.5 # Advection velocity
    state.problem_data['v'] = 1.0

    qinit(state)

    claw = pyclaw.Controller()
    claw.tfinal = 2.0
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.setplot = setplot
    claw.keep_copy = True

    return claw


def setplot(plotdata):
    """
    Plot solution using VisClaw.
    """
    from clawpack.visclaw import colormaps

    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Figure for pcolor plot
    plotfigure = plotdata.new_plotfigure(name='q[0]', figno=0)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'q[0]'
    plotaxes.scaled = True

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = 0
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.pcolor_cmin = 0.0
    plotitem.pcolor_cmax = 1.0
    plotitem.add_colorbar = True

    # Figure for contour plot
    plotfigure = plotdata.new_plotfigure(name='contour', figno=1)

    # Set up for axes in this figure:
```

```
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'q[0]'
    plotaxes.scaled = True

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_contour')
    plotitem.plot_var = 0
    plotitem.contour_nlevels = 20
    plotitem.contour_min = 0.01
    plotitem.contour_max = 0.99
    plotitem.amr_contour_colors = ['b','k','r']

    return plotdata


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Advection in an annular domain Solve the linear advection equation:

$$q_t + (u(x,y)q)_x + (v(x,y)q)_y = 0$$

in an annular domain, using a mapped grid.

Here q is the density of some conserved quantity and (u,v) is the velocity field. We take a rotational velocity field: $u = \cos(\theta), v = \sin(\theta)$.

This is the simplest example that shows how to use a mapped grid in PyClaw.

```
#!/usr/bin/env python
# encoding: utf-8
r"""
Advection in an annular domain
==============================

Solve the linear advection equation:

.. math::
    q_t + (u(x,y) q)_x + (v(x,y) q)_y & = 0

in an annular domain, using a mapped grid.

Here q is the density of some conserved quantity and (u,v) is the velocity
field.  We take a rotational velocity field: :math:'u = \cos(\theta), v = \sin(\theta) '.

This is the simplest example that shows how to use a mapped grid in PyClaw.
"""
import numpy as np

def mapc2p_annulus(grid,c_centers):
    """
    Specifies the mapping to curvilinear coordinates.

    Inputs: c_centers = Computational cell centers
                [array ([Xc1, Xc2, ...]), array([Yc1, Yc2, ...])]

    Output: p_centers = Physical cell centers
```

```
                [array ([Xp1, Xp2, ...]), array([Yp1, Yp2, ...])]
    """
    p_centers = []

    # Polar coordinates (first coordinate = radius,  second coordinate = theta)
    p_centers.append(c_centers[0][:]*np.cos(c_centers[1][:]))
    p_centers.append(c_centers[0][:]*np.sin(c_centers[1][:]))

    return p_centers


def qinit(state):
    """
    Initialize with two Gaussian pulses.
    """
    # First gaussian pulse
    A1     = 1.    # Amplitude
    beta1  = 40.   # Decay factor
    r1     = -0.5  # r-coordinate of the center
    theta1 = 0.    # theta-coordinate of the center

    # Second gaussian pulse
    A2     = -1.   # Amplitude
    beta2  = 40.   # Decay factor
    r2     = 0.5   # r-coordinate of the centers
    theta2 = 0.    # theta-coordinate of the centers

    R, Theta = state.grid.p_centers
    state.q[0,:,:] = A1*np.exp(-beta1*(np.square(R-r1) + np.square(Theta-theta1)))\
                   + A2*np.exp(-beta2*(np.square(R-r2) + np.square(Theta-theta2)))


def ghost_velocities_upper(state,dim,t,qbc,auxbc,num_ghost):
    """
    Set the velocities for the ghost cells outside the outer radius of the annulus.
    In the computational domain, these are the cells at the top of the grid.
    """
    from mapc2p import mapc2p

    grid=state.grid
    if dim == grid.dimensions[0]:
        dx, dy = grid.delta
        X_edges, Y_edges = grid.c_edges_with_ghost(num_ghost=2)
        R_edges,Theta_edges = mapc2p(X_edges,Y_edges)  # Compute edge coordinates in physical domain

        auxbc[:,-num_ghost:,:] = edge_velocities_and_area(R_edges[-num_ghost-1:,:],Theta_edges[-num_g

    else:
        raise Exception('Custom BC for this boundary is not appropriate!')


def ghost_velocities_lower(state,dim,t,qbc,auxbc,num_ghost):
    """
    Set the velocities for the ghost cells outside the inner radius of the annulus.
    In the computational domain, these are the cells at the bottom of the grid.
    """
    from mapc2p import mapc2p
```

```
    grid=state.grid
    if dim == grid.dimensions[0]:
        dx, dy = grid.delta
        X_edges, Y_edges = grid.c_edges_with_ghost(num_ghost=2)
        R_edges,Theta_edges = mapc2p(X_edges,Y_edges)

        auxbc[:,0:num_ghost,:] = edge_velocities_and_area(R_edges[0:num_ghost+1,:],Theta_edges[0:num_

    else:
        raise Exception('Custom BC for this boundary is not appropriate!')


def edge_velocities_and_area(R_edges,Theta_edges,dx,dy):
    """This routine fills in the aux arrays for the problem:

        aux[0,i,j] = u-velocity at left edge of cell (i,j)
        aux[1,i,j] = v-velocity at bottom edge of cell (i,j)
        aux[2,i,j] = physical area of cell (i,j) (relative to area of computational cell)
    """
    mx = R_edges.shape[0]-1
    my = R_edges.shape[1]-1
    aux = np.empty((3,mx,my), order='F')

    # Bottom-left corners
    Xp0 = R_edges[:mx,:my]
    Yp0 = Theta_edges[:mx,:my]

    # Top-left corners
    Xp1 = R_edges[:mx,1:]
    Yp1 = Theta_edges[:mx,1:]

    # Top-right corners
    Xp2 = R_edges[1:,1:]
    Yp2 = Theta_edges[1:,1:]

    # Top-left corners
    Xp3 = R_edges[1:,:my]
    Yp3 = Theta_edges[1:,:my]

    # Compute velocity component
    aux[0,:mx,:my] = (stream(Xp1,Yp1)- stream(Xp0,Yp0))/dy
    aux[1,:mx,:my] = -(stream(Xp3,Yp3)- stream(Xp0,Yp0))/dx

    # Compute area of the physical element
    area = 1./2.*( (Yp0+Yp1)*(Xp1-Xp0) +
                   (Yp1+Yp2)*(Xp2-Xp1) +
                   (Yp2+Yp3)*(Xp3-Xp2) +
                   (Yp3+Yp0)*(Xp0-Xp3) )

    # Compute capa
    aux[2,:mx,:my] = area/(dx*dy)

    return aux


def stream(Xp,Yp):
    """
    Calculates the stream function in physical space.
```

```
    Clockwise rotation. One full rotation corresponds to 1 (second).
    """
    return np.pi*(Xp**2 + Yp**2)


def setup(use_petsc=False,outdir='./_output',solver_type='classic'):
    from clawpack import riemann

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if solver_type == 'classic':
        solver = pyclaw.ClawSolver2D(riemann.vc_advection_2D)
        solver.dimensional_split = False
        solver.transverse_waves = 2
        solver.order = 2
    elif solver_type == 'sharpclaw':
        solver = pyclaw.SharpClawSolver2D(riemann.vc_advection_2D)

    solver.bc_lower[0] = pyclaw.BC.extrap
    solver.bc_upper[0] = pyclaw.BC.extrap
    solver.bc_lower[1] = pyclaw.BC.periodic
    solver.bc_upper[1] = pyclaw.BC.periodic

    solver.aux_bc_lower[0] = pyclaw.BC.custom
    solver.aux_bc_upper[0] = pyclaw.BC.custom
    solver.user_aux_bc_lower = ghost_velocities_lower
    solver.user_aux_bc_upper = ghost_velocities_upper
    solver.aux_bc_lower[1] = pyclaw.BC.periodic
    solver.aux_bc_upper[1] = pyclaw.BC.periodic

    solver.dt_initial = 0.1
    solver.cfl_max = 0.5
    solver.cfl_desired = 0.4

    solver.limiters = pyclaw.limiters.tvd.vanleer

    r_lower = 0.2
    r_upper = 1.0
    m_r = 40

    theta_lower = 0.0
    theta_upper = np.pi*2.0
    m_theta = 120

    r     = pyclaw.Dimension('r',    r_lower,r_upper,m_r)
    theta = pyclaw.Dimension('theta',theta_lower,theta_upper,m_theta)
    domain = pyclaw.Domain([r,theta])
    domain.grid.mapc2p = mapc2p_annulus

    num_eqn = 1
    state = pyclaw.State(domain,num_eqn)

    qinit(state)

    dx, dy = state.grid.delta
```

```
    p_corners = state.grid.p_edges
    state.aux = edge_velocities_and_area(p_corners[0],p_corners[1],dx,dy)
    state.index_capa = 2 # aux[2,:,:] holds the capacity function

    claw = pyclaw.Controller()
    claw.tfinal = 1.0
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.setplot = setplot
    claw.keep_copy = True

    return claw


def setplot(plotdata):
    """
    Plot solution using VisClaw.
    """
    from mapc2p import mapc2p
    import numpy as np
    from clawpack.visclaw import colormaps

    plotdata.clearfigures()  # clear any old figures,axes,items data
    plotdata.mapc2p = mapc2p

    # Figure for contour plot
    plotfigure = plotdata.new_plotfigure(name='contour', figno=0)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.xlimits = 'auto'
    plotaxes.ylimits = 'auto'
    plotaxes.title = 'q[0]'
    plotaxes.scaled = True

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_contour')
    plotitem.plot_var = 0
    plotitem.contour_levels = np.linspace(-0.9, 0.9, 10)
    plotitem.contour_colors = 'k'
    plotitem.patchedges_show = 1
    plotitem.MappedGrid = True

    # Figure for pcolor plot
    plotfigure = plotdata.new_plotfigure(name='q[0]', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.xlimits = 'auto'
    plotaxes.ylimits = 'auto'
    plotaxes.title = 'q[0]'
    plotaxes.scaled = True

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = 0
    plotitem.pcolor_cmap = colormaps.red_yellow_blue
```

```
    plotitem.pcolor_cmin = -1.
    plotitem.pcolor_cmax = 1.
    plotitem.add_colorbar = True
    plotitem.MappedGrid = True


    return plotdata


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

One-dimensional advection with variable velocity Solve the conservative variable-coefficient advection equation:

$$q_t + (u(x)q)_x = 0.$$

Here q is the density of some conserved quantity and u(x) is the velocity. The velocity field used is

$$u(x) = 2 + sin(2\pi x).$$

The boundary conditions are periodic. The initial data get stretched and compressed as they move through the fast and slow parts of the velocity field.

```
#!/usr/bin/env python
# encoding: utf-8
r"""
One-dimensional advection with variable velocity
================================================

Solve the conservative variable-coefficient advection equation:

.. math:: q_t + (u(x)q)_x = 0.

Here q is the density of some conserved quantity and u(x) is the velocity.
The velocity field used is

.. math:: u(x) = 2 + sin(2\pi x).

The boundary conditions are periodic.
The initial data get stretched and compressed as they move through the
fast and slow parts of the velocity field.
"""


import numpy as np

def qinit(state):

    # Initial Data parameters
    ic = 3
    beta = 100.
    gamma = 0.
    x0 = 0.3
    x1 = 0.7
    x2 = 0.9

    x =state.grid.x.centers
```

```
    # Gaussian
    qg = np.exp(-beta * (x-x0)**2) * np.cos(gamma * (x - x0))
    # Step Function
    qs = (x > x1) * 1.0 - (x > x2) * 1.0

    if   ic == 1: state.q[0,:] = qg
    elif ic == 2: state.q[0,:] = qs
    elif ic == 3: state.q[0,:] = qg + qs


def auxinit(state):
    # Initilize petsc Structures for aux
    xc=state.grid.x.centers
    state.aux[0,:] = np.sin(2.*np.pi*xc)+2


def setup(use_petsc=False,solver_type='classic',kernel_language='Python',outdir='./_output'):
    from clawpack import riemann

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if solver_type=='classic':
        if kernel_language == 'Fortran':
            solver = pyclaw.ClawSolver1D(riemann.vc_advection_1D)
        elif kernel_language=='Python':
            solver = pyclaw.ClawSolver1D(riemann.vc_advection_1D_py.vc_advection_1D)
    elif solver_type=='sharpclaw':
        if kernel_language == 'Fortran':
            solver = pyclaw.SharpClawSolver1D(riemann.vc_advection_1D)
        elif kernel_language=='Python':
            solver = pyclaw.SharpClawSolver1D(riemann.vc_advection_1D_py.vc_advection_1D)
        solver.weno_order=weno_order
    else: raise Exception('Unrecognized value of solver_type.')

    solver.kernel_language = kernel_language

    solver.limiters = pyclaw.limiters.tvd.MC
    solver.bc_lower[0] = 2
    solver.bc_upper[0] = 2
    solver.aux_bc_lower[0] = 2
    solver.aux_bc_upper[0] = 2

    xlower=0.0; xupper=1.0; mx=100
    x    = pyclaw.Dimension('x',xlower,xupper,mx)
    domain = pyclaw.Domain(x)
    num_aux=1
    num_eqn = 1
    state = pyclaw.State(domain,num_eqn,num_aux)

    qinit(state)
    auxinit(state)

    claw = pyclaw.Controller()
    claw.outdir = outdir
    claw.solution = pyclaw.Solution(state,domain)
```

```
    claw.solver = solver

    claw.tfinal = 1.0
    claw.setplot = setplot
    claw.keep_copy = True

    return claw

#--------------------------
def setplot(plotdata):
#--------------------------
    """
    Specify what is to be plotted at each frame.
    Input:  plotdata, an instance of visclaw.data.ClawPlotData.
    Output: a modified version of plotdata.
    """
    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Figure for q[0]
    plotfigure = plotdata.new_plotfigure(name='q', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.ylimits = [-.1,1.1]
    plotaxes.title = 'q'

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
    plotitem.plot_var = 0
    plotitem.plotstyle = '-o'
    plotitem.color = 'b'
    plotitem.kwargs = {'linewidth':2,'markersize':5}

    return plotdata


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

One-dimensional acoustics Solve the (linear) acoustics equations:

$$p_t + K u_x = 0$$
$$u_t + p_x/\rho = 0.$$

Here p is the pressure, u is the velocity, K is the bulk modulus, and ρ is the density.

The initial condition is a Gaussian and the boundary conditions are periodic. The final solution is identical to the initial data because both waves have crossed the domain exactly once.

```
#!/usr/bin/env python
# encoding: utf-8

r"""
One-dimensional acoustics
=========================

Solve the (linear) acoustics equations:
```

```
.. math::
    p_t + K u_x & = 0 \\
    u_t + p_x / \rho & = 0.

Here p is the pressure, u is the velocity, K is the bulk modulus,
and :math:`\rho` is the density.

The initial condition is a Gaussian and the boundary conditions are periodic.
The final solution is identical to the initial data because both waves have
crossed the domain exactly once.
"""
from numpy import sqrt, exp, cos
from clawpack import riemann

def setup(use_petsc=False, kernel_language='Fortran', solver_type='classic',
              outdir='./_output', ptwise=False, weno_order=5,
              time_integrator='SSP104',
              disable_output=False):

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if kernel_language == 'Fortran':
        if ptwise:
            riemann_solver = riemann.acoustics_1D_ptwise
        else:
            riemann_solver = riemann.acoustics_1D

    elif kernel_language=='Python':
        riemann_solver = riemann.acoustics_1D_py.acoustics_1D

    if solver_type=='classic':
        solver = pyclaw.ClawSolver1D(riemann_solver)
        solver.limiters = pyclaw.limiters.tvd.MC
    elif solver_type=='sharpclaw':
        solver = pyclaw.SharpClawSolver1D(riemann_solver)
        solver.weno_order=weno_order
        solver.time_integrator=time_integrator
    else: raise Exception('Unrecognized value of solver_type.')

    solver.kernel_language=kernel_language

    x = pyclaw.Dimension('x',0.0,1.0,100)
    domain = pyclaw.Domain(x)
    num_eqn = 2
    state = pyclaw.State(domain,num_eqn)

    solver.bc_lower[0] = pyclaw.BC.periodic
    solver.bc_upper[0] = pyclaw.BC.periodic

    rho  = 1.0 # Material density
    bulk = 1.0 # Material bulk modulus

    state.problem_data['rho']=rho
    state.problem_data['bulk']=bulk
    state.problem_data['zz']=sqrt(rho*bulk) # Impedance
```

```
    state.problem_data['cc']=sqrt(bulk/rho) # Sound speed

    xc=domain.grid.x.centers
    beta=100; gamma=0; x0=0.75
    state.q[0,:] = exp(-beta * (xc-x0)**2) * cos(gamma * (xc - x0))
    state.q[1,:] = 0.

    solver.dt_initial=domain.grid.delta[0]/state.problem_data['cc']*0.1

    claw = pyclaw.Controller()
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.keep_copy = True
    claw.num_output_times = 10
    if disable_output:
        claw.output_format = None
    claw.tfinal = 1.0
    claw.setplot = setplot

    return claw


def setplot(plotdata):
    """
    Specify what is to be plotted at each frame.
    Input:  plotdata, an instance of visclaw.data.ClawPlotData.
    Output: a modified version of plotdata.
    """
    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Figure for pressure
    plotfigure = plotdata.new_plotfigure(name='Pressure', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.axescmd = 'subplot(211)'
    plotaxes.ylimits = [-.2,1.0]
    plotaxes.title = 'Pressure'

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
    plotitem.plot_var = 0
    plotitem.plotstyle = '-o'
    plotitem.color = 'b'
    plotitem.kwargs = {'linewidth':2,'markersize':5}

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.axescmd = 'subplot(212)'
    plotaxes.xlimits = 'auto'
    plotaxes.ylimits = [-.5,1.1]
    plotaxes.title = 'Velocity'

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
    plotitem.plot_var = 1
    plotitem.plotstyle = '-'
```

```
    plotitem.color = 'b'
    plotitem.kwargs = {'linewidth':3,'markersize':5}

    return plotdata


def run_and_plot(**kwargs):
    claw = setup(kwargs)
    claw.run()
    from clawpack.pyclaw import plot
    plot.interactive_plot(setplot=setplot)

if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Two-dimensional acoustics Solve the (linear) acoustics equations:

$$\begin{aligned} p_t + K(u_x + v_y) &= 0 \\ u_t + p_x/\rho &= 0 \\ v_t + p_y/\rho &= 0. \end{aligned}$$

Here p is the pressure, (u,v) is the velocity, K is the bulk modulus, and ρ is the density.

```
#!/usr/bin/env python
# encoding: utf-8
r"""
Two-dimensional acoustics
=========================

Solve the (linear) acoustics equations:

.. math::
    p_t + K (u_x + v_y) & = 0 \\
    u_t + p_x / \rho & = 0 \\
    v_t + p_y / \rho & = 0.

Here p is the pressure, (u,v) is the velocity, K is the bulk modulus,
and :math:`\rho` is the density.
"""
from clawpack import riemann
import numpy as np

def setup(kernel_language='Fortran', use_petsc=False, outdir='./_output',
              solver_type='classic', time_integrator='SSP104', ptwise=False,
              disable_output=False):
    """
    Example python script for solving the 2d acoustics equations.
    """
    if use_petsc:
        from clawpack import petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if solver_type == 'classic':
        if ptwise:
            solver = pyclaw.ClawSolver2D(riemann.acoustics_2D_ptwise)
```

```
        else:
            solver = pyclaw.ClawSolver2D(riemann.acoustics_2D)
        solver.dimensional_split=True
        solver.cfl_max = 0.5
        solver.cfl_desired = 0.45
        solver.limiters = pyclaw.limiters.tvd.MC
    elif solver_type=='sharpclaw':
        solver=pyclaw.SharpClawSolver2D(riemann.acoustics_2D)
        solver.time_integrator=time_integrator
        if solver.time_integrator=='SSP104':
            solver.cfl_max = 0.5
            solver.cfl_desired = 0.45
        elif solver.time_integrator=='SSPMS32':
            solver.cfl_max = 0.2
            solver.cfl_desired = 0.16
        else:
            raise Exception('CFL desired and CFL max have not been provided for the particular time i

    solver.bc_lower[0]=pyclaw.BC.extrap
    solver.bc_upper[0]=pyclaw.BC.extrap
    solver.bc_lower[1]=pyclaw.BC.extrap
    solver.bc_upper[1]=pyclaw.BC.extrap

    mx=100; my=100
    x = pyclaw.Dimension('x',-1.0,1.0,mx)
    y = pyclaw.Dimension('y',-1.0,1.0,my)
    domain = pyclaw.Domain([x,y])

    num_eqn = 3
    state = pyclaw.State(domain,num_eqn)

    rho  = 1.0  # Material density
    bulk = 4.0  # Material bulk modulus
    cc = np.sqrt(bulk/rho)  # sound speed
    zz = rho*cc             # impedance
    state.problem_data['rho']= rho
    state.problem_data['bulk']=bulk
    state.problem_data['zz']= zz
    state.problem_data['cc']=cc

    solver.dt_initial=np.min(domain.grid.delta)/state.problem_data['cc']*solver.cfl_desired

    qinit(state)

    claw = pyclaw.Controller()
    claw.keep_copy = True
    if disable_output:
        claw.output_format = None
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.num_output_times = 10
    claw.tfinal = 0.12
    claw.setplot = setplot

    return claw

def qinit(state,width=0.2):
```

```
    X, Y = state.grid.p_centers
    r = np.sqrt(X**2 + Y**2)

    state.q[0,:,:] = (np.abs(r-0.5)<=width)*(1.+np.cos(np.pi*(r-0.5)/width))
    state.q[1,:,:] = 0.
    state.q[2,:,:] = 0.


def setplot(plotdata):
    """
    Plot output with VisClaw.
    This example demonstrates how to plot a 1D projection from 2D data.
    """

    from clawpack.visclaw import colormaps

    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Figure for pressure
    plotfigure = plotdata.new_plotfigure(name='Pressure', figno=0)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'Pressure'
    plotaxes.scaled = True      # so aspect ratio is 1

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = 0
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.add_colorbar = True

    # Figure for scatter plot
    plotfigure = plotdata.new_plotfigure(name='scatter', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'Scatter plot'

    # Set up for item on these axes: scatter of 2d data
    plotitem = plotaxes.new_plotitem(plot_type='1d_from_2d_data')

    def p_vs_r(current_data):
        # Return radius of each patch cell and p value in the cell
        from pylab import sqrt
        x = current_data.x
        y = current_data.y
        r = sqrt(x**2 + y**2)
        q = current_data.q
        p = q[0,:,:]
        return r,p

    plotitem.map_2d_to_1d = p_vs_r
    plotitem.plot_var = 0
    plotitem.plotstyle = 'ob'

    return plotdata
```

```
if __name__=="__main__":
    import sys
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Two-dimensional variable-coefficient acoustics Solve the variable-coefficient acoustics equations in 2D:

$$\begin{aligned} p_t + K(x,y)(u_x + v_y) &= 0 \\ u_t + p_x/\rho(x,y) &= 0 \\ v_t + p_y/\rho(x,y) &= 0. \end{aligned}$$

Here p is the pressure, (u,v) is the velocity, $K(x,y)$ is the bulk modulus, and $\rho(x,y)$ is the density.

This example shows how to solve a problem with variable coefficients. The left and right halves of the domain consist of different materials.

```
#!/usr/bin/env python
# encoding: utf-8
r"""
Two-dimensional variable-coefficient acoustics
==============================================

Solve the variable-coefficient acoustics equations in 2D:

.. math::
    p_t + K(x,y) (u_x + v_y) & = 0 \\
    u_t + p_x / \rho(x,y) & = 0 \\
    v_t + p_y / \rho(x,y) & = 0.

Here p is the pressure, (u,v) is the velocity, :math:`K(x,y)` is the bulk modulus,
and :math:`\rho(x,y)` is the density.

This example shows how to solve a problem with variable coefficients.
The left and right halves of the domain consist of different materials.
"""

import numpy as np

def setup(kernel_language='Fortran', use_petsc=False, outdir='./_output',
          solver_type='classic', time_integrator='SSP104', lim_type=2,
          disable_output=False, num_cells=(200, 200)):
    """
    Example python script for solving the 2d acoustics equations.
    """
    from clawpack import riemann

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if solver_type=='classic':
        solver=pyclaw.ClawSolver2D(riemann.vc_acoustics_2D)
        solver.dimensional_split=False
        solver.limiters = pyclaw.limiters.tvd.MC
    elif solver_type=='sharpclaw':
        solver=pyclaw.SharpClawSolver2D(riemann.vc_acoustics_2D)
```

```
    solver.time_integrator=time_integrator
    if time_integrator=='SSPMS32':
        solver.cfl_max = 0.25
        solver.cfl_desired = 0.24

solver.bc_lower[0]=pyclaw.BC.wall
solver.bc_upper[0]=pyclaw.BC.extrap
solver.bc_lower[1]=pyclaw.BC.wall
solver.bc_upper[1]=pyclaw.BC.extrap
solver.aux_bc_lower[0]=pyclaw.BC.wall
solver.aux_bc_upper[0]=pyclaw.BC.extrap
solver.aux_bc_lower[1]=pyclaw.BC.wall
solver.aux_bc_upper[1]=pyclaw.BC.extrap

x = pyclaw.Dimension('x',-1.0,1.0,num_cells[0])
y = pyclaw.Dimension('y',-1.0,1.0,num_cells[1])
domain = pyclaw.Domain([x,y])

num_eqn = 3
num_aux = 2 # density, sound speed
state = pyclaw.State(domain,num_eqn,num_aux)

grid = state.grid
X, Y = grid.p_centers

rho_left   = 4.0  # Density in left half
rho_right  = 1.0  # Density in right half
bulk_left  = 4.0  # Bulk modulus in left half
bulk_right = 4.0  # Bulk modulus in right half
c_left = np.sqrt(bulk_left/rho_left)      # Sound speed (left)
c_right = np.sqrt(bulk_right/rho_right)   # Sound speed (right)
state.aux[0,:,:] = rho_left*(X<0.) + rho_right*(X>=0.) # Density
state.aux[1,:,:] = c_left*(X<0.) + c_right*(X>=0.)     # Sound speed

# Set initial condition
x0 = -0.5; y0 = 0.
r = np.sqrt((X-x0)**2 + (Y-y0)**2)
width = 0.1; rad = 0.25
state.q[0,:,:] = (np.abs(r-rad)<=width)*(1.+np.cos(np.pi*(r-rad)/width))
state.q[1,:,:] = 0.
state.q[2,:,:] = 0.

claw = pyclaw.Controller()
claw.keep_copy = True
if disable_output:
    claw.output_format = None
claw.solution = pyclaw.Solution(state,domain)
claw.solver = solver
claw.outdir = outdir
claw.tfinal = 0.6
claw.num_output_times = 20
claw.write_aux_init = True
claw.setplot = setplot
if use_petsc:
    claw.output_options = {'format':'binary'}

return claw
```

```
def setplot(plotdata):
    """
    Plot solution using VisClaw.

    This example shows how to mark an internal boundary on a 2D plot.
    """

    from clawpack.visclaw import colormaps

    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Figure for pressure
    plotfigure = plotdata.new_plotfigure(name='Pressure', figno=0)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'Pressure'
    plotaxes.scaled = True      # so aspect ratio is 1
    plotaxes.afteraxes = mark_interface

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = 0
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.add_colorbar = True
    plotitem.pcolor_cmin = 0.0
    plotitem.pcolor_cmax=1.0

    # Figure for x-velocity plot
    plotfigure = plotdata.new_plotfigure(name='x-Velocity', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'u'
    plotaxes.afteraxes = mark_interface

    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = 1
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.add_colorbar = True
    plotitem.pcolor_cmin = -0.3
    plotitem.pcolor_cmax=   0.3

    return plotdata

def mark_interface(current_data):
    import matplotlib.pyplot as plt
    plt.plot((0.,0.),(-1.,1.),'-k',linewidth=2)


if __name__=="__main__":
    import sys
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Burgers' equation Solve the inviscid Burgers' equation:

$$q_t + \frac{1}{2}(q^2)_x = 0.$$

This is a nonlinear PDE often used as a very simple model for fluid dynamics.

The initial condition is sinusoidal, but after a short time a shock forms (due to the nonlinearity).

```
#!/usr/bin/env python
# encoding: utf-8

r"""
Burgers' equation
=========================

Solve the inviscid Burgers' equation:

.. math::
    q_t + \frac{1}{2} (q^2)_x & = 0.

This is a nonlinear PDE often used as a very simple
model for fluid dynamics.

The initial condition is sinusoidal, but after a short time a shock forms
(due to the nonlinearity).
"""
import numpy as np
from clawpack import riemann

def setup(use_petsc=0,kernel_language='Fortran',outdir='./_output',solver_type='classic'):

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if kernel_language == 'Python':
        riemann_solver = riemann.burgers_1D_py.burgers_1D
    elif kernel_language == 'Fortran':
        riemann_solver = riemann.burgers_1D

    if solver_type=='sharpclaw':
        solver = pyclaw.SharpClawSolver1D(riemann_solver)
    else:
        solver = pyclaw.ClawSolver1D(riemann_solver)
        solver.limiters = pyclaw.limiters.tvd.vanleer

    solver.kernel_language = kernel_language

    solver.bc_lower[0] = pyclaw.BC.periodic
    solver.bc_upper[0] = pyclaw.BC.periodic

    x = pyclaw.Dimension('x',0.0,1.0,500)
    domain = pyclaw.Domain(x)
    num_eqn = 1
    state = pyclaw.State(domain,num_eqn)

    xc = state.grid.x.centers
    state.q[0,:] = np.sin(np.pi*2*xc) + 0.50
```

```
    state.problem_data['efix']=True

    claw = pyclaw.Controller()
    claw.tfinal = 0.5
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.setplot = setplot
    claw.keep_copy = True

    return claw


def setplot(plotdata):
    """
    Plot solution using VisClaw.
    """
    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Figure for q[0]
    plotfigure = plotdata.new_plotfigure(name='q[0]', figno=0)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.xlimits = 'auto'
    plotaxes.ylimits = [-1., 2.]
    plotaxes.title = 'q[0]'

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d')
    plotitem.plot_var = 0
    plotitem.plotstyle = '-o'
    plotitem.color = 'b'

    return plotdata


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Shallow water flow Solve the one-dimensional shallow water equations:

$$
\begin{aligned}
h_t + (hu)_x + (hv)_y &= 0 \\
(hu)_t + (hu^2 + \frac{1}{2}gh^2)_x + (huv)_y &= 0 \\
(hv)_t + (huv)_x + (hv^2 + \frac{1}{2}gh^2)_y &= 0.
\end{aligned}
$$

Here h is the depth, (u,v) is the velocity, and g is the gravitational constant.

Include file u'/Users/rjl/git/clawpack/pyclaw/examples/shallow_1d/shallow_water_shocktube.py' not found or reading it failed

Woodward-Colella blast wave problem Solve the one-dimensional Euler equations for inviscid, compressible flow:

$$\begin{aligned} \rho_t + (\rho u)_x &= 0 \\ (\rho u)_t + (\rho u^2 + p)_x &= 0 \\ E_t + (u(E+p))_x &= 0. \end{aligned}$$

The fluid is an ideal gas, with pressure given by $p = \rho(\gamma - 1)e$ where e is internal energy.

This script runs the Woodward-Colella blast wave interaction problem, involving the collision of two shock waves.

This example also demonstrates:

- How to use a total fluctuation solver in SharpClaw
- How to use characteristic decomposition with an evec() routine in SharpClaw

```
#!/usr/bin/env python
# encoding: utf-8
r"""
Woodward-Colella blast wave problem
===================================

Solve the one-dimensional Euler equations for inviscid, compressible flow:

.. math::
    \rho_t + (\rho u)_x & = 0 \\
    (\rho u)_t + (\rho u^2 + p)_x & = 0 \\
    E_t + (u (E + p) )_x & = 0.

The fluid is an ideal gas, with pressure given by :math:`p=\rho (\gamma-1)e` where
e is internal energy.

This script runs the Woodward-Colella blast wave interaction problem,
involving the collision of two shock waves.

This example also demonstrates:

 - How to use a total fluctuation solver in SharpClaw
 - How to use characteristic decomposition with an evec() routine in SharpClaw
"""
from clawpack import riemann
from clawpack.riemann.euler_with_efix_1D_constants import *

# Compile Fortran code if not already compiled
try:
    import sharpclaw1
except ImportError:
    import os
    from clawpack.pyclaw.util import inplace_build
    this_dir = os.path.dirname(__file__)
    if this_dir == '':
        this_dir = os.path.abspath('.')
    inplace_build(this_dir)
    try:
        # Now try to import again
        import sharpclaw1
    except ImportError:
        import logging
        logger = logging.getLogger()
        logger.warn('unable to compile Fortran modules; some SharpClaw options will not be available
```

```
        print 'unable to compile Fortran modules; some SharpClaw options will not be available for th
        raise

gamma = 1.4 # Ratio of specific heats

def setup(use_petsc=False,outdir='./_output',solver_type='sharpclaw',kernel_language='Fortran',tfluct

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if kernel_language =='Python':
        rs = riemann.euler_1D_py.euler_roe_1D
    elif kernel_language =='Fortran':
        rs = riemann.euler_with_efix_1D

    if solver_type=='sharpclaw':
        solver = pyclaw.SharpClawSolver1D(rs)
        solver.time_integrator = 'SSP33'
        solver.cfl_max = 0.65
        solver.cfl_desired = 0.6
        solver.tfluct_solver = tfluct_solver
        if solver.tfluct_solver:
            try:
                import euler_tfluct
                solver.tfluct = euler_tfluct
            except ImportError:
                import logging
                logger = logging.getLogger()
                logger.error('Unable to load tfluct solver, did you run make?')
                print 'Unable to load tfluct solver, did you run make?'
                raise
        solver.lim_type = 1
        solver.char_decomp = 2
        try:
            import sharpclaw1
            solver.fmod = sharpclaw1
        except ImportError:
            pass
    elif solver_type=='classic':
        solver = pyclaw.ClawSolver1D(rs)
        solver.limiters = 4

    solver.kernel_language = kernel_language

    solver.bc_lower[0]=pyclaw.BC.wall
    solver.bc_upper[0]=pyclaw.BC.wall

    mx = 800;
    x = pyclaw.Dimension('x',0.0,1.0,mx)
    domain = pyclaw.Domain([x])
    state = pyclaw.State(domain,num_eqn)

    state.problem_data['gamma'] = gamma
    if kernel_language =='Python':
        state.problem_data['efix'] = False
```

```
    x = state.grid.x.centers

    state.q[density ,:] = 1.
    state.q[momentum,:] = 0.
    state.q[energy  ,:] = ( (x<0.1)*1.e3 + (0.1<=x)*(x<0.9)*1.e-2 + (0.9<=x)*1.e2 ) / (gamma - 1.)

    claw = pyclaw.Controller()
    claw.tfinal = 0.038
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.num_output_times = 10
    claw.outdir = outdir
    claw.setplot = setplot
    claw.keep_copy = True

    return claw

#--------------------------
def setplot(plotdata):
#--------------------------
    """
    Specify what is to be plotted at each frame.
    Input:  plotdata, an instance of visclaw.data.ClawPlotData.
    Output: a modified version of plotdata.
    """
    plotdata.clearfigures()  # clear any old figures,axes,items data

    plotfigure = plotdata.new_plotfigure(name='', figno=0)

    plotaxes = plotfigure.new_plotaxes()
    plotaxes.axescmd = 'subplot(211)'
    plotaxes.title = 'Density'

    plotitem = plotaxes.new_plotitem(plot_type='1d')
    plotitem.plot_var = density
    plotitem.kwargs = {'linewidth':3}

    plotaxes = plotfigure.new_plotaxes()
    plotaxes.axescmd = 'subplot(212)'
    plotaxes.title = 'Energy'

    plotitem = plotaxes.new_plotitem(plot_type='1d')
    plotitem.plot_var = energy
    plotitem.kwargs = {'linewidth':3}

    return plotdata

if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Compressible Euler flow in cylindrical symmetry Solve the Euler equations of compressible fluid dynamics in 2D r-z coordinates:

$$\rho_t + (\rho u)_x + (\rho v)_y = -\rho v/r$$
$$(\rho u)_t + (\rho u^2 + p)_x + (\rho uv)_y = -\rho uv/r$$
$$(\rho v)_t + (\rho uv)_x + (\rho v^2 + p)_y = -\rho v^2/r$$
$$E_t + (u(E+p))_x + (v(E+p))_y = -(E+p)v/r.$$

Here ρ is the density, (u,v) is the velocity, and E is the total energy. The radial coordinate is denoted by r.

The problem involves a planar shock wave impacting a spherical low-density bubble. The problem is 3-dimensional but has been reduced to two dimensions using cylindrical symmetry.

This problem demonstrates:

- how to incorporate source (non-hyperbolic) terms using both Classic and SharpClaw solvers
- how to impose a custom boundary condition
- how to use the auxiliary array for spatially-varying coefficients

```
#!/usr/bin/env python
# encoding: utf-8
r"""
Compressible Euler flow in cylindrical symmetry
===============================================

Solve the Euler equations of compressible fluid dynamics in 2D r-z coordinates:

.. math::
    \rho_t + (\rho u)_x + (\rho v)_y & = - \rho v / r \\
    (\rho u)_t + (\rho u^2 + p)_x + (\rho uv)_y & = -\rho u v / r \\
    (\rho v)_t + (\rho uv)_x + (\rho v^2 + p)_y & = - \rho v^2 / r \\
    E_t + (u (E + p) )_x + (v (E + p))_y & = - (E + p) v / r.

Here :math:`\rho` is the density, (u,v) is the velocity, and E is the total energy.
The radial coordinate is denoted by r.

The problem involves a planar shock wave impacting a spherical low-density bubble.
The problem is 3-dimensional but has been reduced to two dimensions using
cylindrical symmetry.

This problem demonstrates:

    - how to incorporate source (non-hyperbolic) terms using both Classic and SharpClaw solvers
    - how to impose a custom boundary condition
    - how to use the auxiliary array for spatially-varying coefficients
"""

import numpy as np
from clawpack import riemann
from clawpack.riemann.euler_5wave_2D_constants import density, x_momentum, y_momentum, \
        energy, num_eqn

gamma = 1.4 # Ratio of specific heats

x0=0.5; y0=0.; r0=0.2


def ycirc(x,ymin,ymax):
```

```
    if ((x-x0)**2)<(r0**2):
        return max(min(y0 + np.sqrt(r0**2-(x-x0)**2),ymax) - ymin,0.)
    else:
        return 0

def qinit(state,rhoin=0.1,pinf=5.):
    from scipy import integrate

    gamma1 = gamma - 1.

    grid = state.grid

    rhoout = 1.
    pout   = 1.
    pin    = 1.
    xshock = 0.2

    rinf = (gamma1 + pinf*(gamma+1.))/ ((gamma+1.) + gamma1*pinf)
    vinf = 1./np.sqrt(gamma) * (pinf - 1.) / np.sqrt(0.5*((gamma+1.)/gamma) * pinf+0.5*gamma1/gamma)
    einf = 0.5*rinf*vinf**2 + pinf/gamma1

    X, Y = grid.p_centers

    r = np.sqrt((X-x0)**2 + (Y-y0)**2)

    #First set the values for the cells that don't intersect the bubble boundary
    state.q[0,:,:] = rinf*(X<xshock) + rhoin*(r<=r0) + rhoout*(r>r0)*(X>=xshock)
    state.q[1,:,:] = rinf*vinf*(X<xshock)
    state.q[2,:,:] = 0.
    state.q[3,:,:] = einf*(X<xshock) + (pin*(r<=r0) + pout*(r>r0)*(X>=xshock))/gamma1
    state.q[4,:,:] = 1.*(r<=r0)

    #Now compute average density for the cells on the edge of the bubble
    d2 = np.linalg.norm(state.grid.delta)/2.
    dx = state.grid.delta[0]
    dy = state.grid.delta[1]
    dx2 = state.grid.delta[0]/2.
    dy2 = state.grid.delta[1]/2.
    for i in xrange(state.q.shape[1]):
        for j in xrange(state.q.shape[2]):
            ydown = Y[i,j]-dy2
            yup   = Y[i,j]+dy2
            if abs(r[i,j]-r0)<d2:
                infrac,abserr = integrate.quad(ycirc,X[i,j]-dx2,X[i,j]+dx2,args=(ydown,yup),epsabs=1
                infrac=infrac/(dx*dy)
                state.q[0,i,j] = rhoin*infrac + rhoout*(1.-infrac)
                state.q[3,i,j] = (pin*infrac + pout*(1.-infrac))/gamma1
                state.q[4,i,j] = 1.*infrac


def auxinit(state):
    """
    aux[1,i,j] = radial coordinate of cell centers for cylindrical source terms
    """
    y = state.grid.y.centers
    for j,r in enumerate(y):
        state.aux[0,:,j] = r
```

```
def incoming_shock(state,dim,t,qbc,auxbc,num_ghost):
    """
    Incoming shock at left boundary.
    """
    gamma1 = gamma - 1.

    pinf=5.
    rinf = (gamma1 + pinf*(gamma+1.))/ ((gamma+1.) + gamma1*pinf)
    vinf = 1./np.sqrt(gamma) * (pinf - 1.) / np.sqrt(0.5*((gamma+1.)/gamma) * pinf+0.5*gamma1/gamma)
    einf = 0.5*rinf*vinf**2 + pinf/gamma1

    for i in xrange(num_ghost):
        qbc[0,i,...] = rinf
        qbc[1,i,...] = rinf*vinf
        qbc[2,i,...] = 0.
        qbc[3,i,...] = einf
        qbc[4,i,...] = 0.


def step_Euler_radial(solver,state,dt):
    """
    Geometric source terms for Euler equations with cylindrical symmetry.
    Integrated using a 2-stage, 2nd-order Runge-Kutta method.
    This is a Clawpack-style source term routine, which approximates
    the integral of the source terms over a step.
    """
    dt2 = dt/2.

    q = state.q
    rad = state.aux[0,:,:]

    rho = q[0,:,:]
    u   = q[1,:,:]/rho
    v   = q[2,:,:]/rho
    press  = (gamma - 1.) * (q[3,:,:] - 0.5*rho*(u**2 + v**2))

    qstar = np.empty(q.shape)

    qstar[0,:,:] = q[0,:,:] - dt2/rad * q[2,:,:]
    qstar[1,:,:] = q[1,:,:] - dt2/rad * rho*u*v
    qstar[2,:,:] = q[2,:,:] - dt2/rad * rho*v*v
    qstar[3,:,:] = q[3,:,:] - dt2/rad * v * (q[3,:,:] + press)

    rho = qstar[0,:,:]
    u   = qstar[1,:,:]/rho
    v   = qstar[2,:,:]/rho
    press  = (gamma - 1.) * (qstar[3,:,:] - 0.5*rho*(u**2 + v**2))

    q[0,:,:] = q[0,:,:] - dt/rad * qstar[2,:,:]
    q[1,:,:] = q[1,:,:] - dt/rad * rho*u*v
    q[2,:,:] = q[2,:,:] - dt/rad * rho*v*v
    q[3,:,:] = q[3,:,:] - dt/rad * v * (qstar[3,:,:] + press)


def dq_Euler_radial(solver,state,dt):
    """
    Geometric source terms for Euler equations with radial symmetry.
    This is a SharpClaw-style source term routine, which returns
```

```
    the value of the source terms.
    """
    q   = state.q
    rad = state.aux[0,:,:]

    rho = q[0,:,:]
    u   = q[1,:,:]/rho
    v   = q[2,:,:]/rho
    press  = (gamma - 1.) * (q[3,:,:] - 0.5*rho*(u**2 + v**2))

    dq = np.empty(q.shape)

    dq[0,:,:] = -dt/rad * q[2,:,:]
    dq[1,:,:] = -dt/rad * rho*u*v
    dq[2,:,:] = -dt/rad * rho*v*v
    dq[3,:,:] = -dt/rad * v * (q[3,:,:] + press)
    dq[4,:,:] = 0

    return dq

def setup(use_petsc=False,solver_type='classic', outdir='_output', kernel_language='Fortran',
        disable_output=False, mx=320, my=80, tfinal=0.6, num_output_times = 10):
    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if solver_type=='sharpclaw':
        solver = pyclaw.SharpClawSolver2D(riemann.euler_5wave_2D)
        solver.dq_src = dq_Euler_radial
        solver.weno_order = 5
        solver.lim_type   = 2
    else:
        solver = pyclaw.ClawSolver2D(riemann.euler_5wave_2D)
        solver.step_source = step_Euler_radial
        solver.source_split = 1
        solver.limiters = [4,4,4,4,2]
        solver.cfl_max = 0.5
        solver.cfl_desired = 0.45

    x = pyclaw.Dimension('x',0.0,2.0,mx)
    y = pyclaw.Dimension('y',0.0,0.5,my)
    domain = pyclaw.Domain([x,y])

    num_aux=1
    state = pyclaw.State(domain,num_eqn,num_aux)
    state.problem_data['gamma']= gamma

    qinit(state)
    auxinit(state)

    solver.user_bc_lower = incoming_shock

    solver.bc_lower[0]=pyclaw.BC.custom
    solver.bc_upper[0]=pyclaw.BC.extrap
    solver.bc_lower[1]=pyclaw.BC.wall
    solver.bc_upper[1]=pyclaw.BC.extrap
    #Aux variable in ghost cells doesn't matter
```

```
    solver.aux_bc_lower[0]=pyclaw.BC.extrap
    solver.aux_bc_upper[0]=pyclaw.BC.extrap
    solver.aux_bc_lower[1]=pyclaw.BC.extrap
    solver.aux_bc_upper[1]=pyclaw.BC.extrap

    claw = pyclaw.Controller()
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver

    claw.keep_copy = True
    if disable_output:
        claw.output_format = None
    claw.tfinal = tfinal
    claw.num_output_times = num_output_times
    claw.outdir = outdir
    claw.setplot = setplot

    return claw


def setplot(plotdata):
    """
    Plot solution using VisClaw.
    """
    from clawpack.visclaw import colormaps

    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Pressure plot
    plotfigure = plotdata.new_plotfigure(name='Density', figno=0)

    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'Density'
    plotaxes.scaled = True      # so aspect ratio is 1
    plotaxes.afteraxes = label_axes

    plotitem = plotaxes.new_plotitem(plot_type='2d_schlieren')
    plotitem.plot_var = 0
    plotitem.add_colorbar = False


    # Tracer plot
    plotfigure = plotdata.new_plotfigure(name='Tracer', figno=1)

    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'Tracer'
    plotaxes.scaled = True      # so aspect ratio is 1
    plotaxes.afteraxes = label_axes

    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.pcolor_cmin = 0.
    plotitem.pcolor_cmax=1.0
    plotitem.plot_var = 4
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.add_colorbar = False


    # Energy plot
```

```
    plotfigure = plotdata.new_plotfigure(name='Energy', figno=2)

    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'Energy'
    plotaxes.scaled = True      # so aspect ratio is 1
    plotaxes.afteraxes = label_axes

    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.pcolor_cmin = 2.
    plotitem.pcolor_cmax=18.0
    plotitem.plot_var = 3
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.add_colorbar = False

    return plotdata

def label_axes(current_data):
    import matplotlib.pyplot as plt
    plt.xlabel('z')
    plt.ylabel('r')


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Two-dimensional p-system Solve the two-dimensional generalization of the p-system:

$$\begin{aligned}\epsilon_t - u_x - v_y &= 0\\ \rho(x,y)u_t - \sigma(\epsilon,x,y)_x &= 0\\ \rho(x,y)v_t - \sigma(\epsilon,x,y)_y &= 0.\end{aligned}$$

We take $\sigma = e^{K(x,y)\epsilon} - 1$, and the material coefficients ρ, K vary in a checkerboard pattern. The resulting dynamics lead to solitary waves, though much more resolution is needed in order to see them.

This example shows how to set an aux array, use a b4step function, use gauges, compute output functionals, and restart a simulation from a checkpoint.

```
#!/usr/bin/env python
# encoding: utf-8
r"""
Two-dimensional p-system
============================

Solve the two-dimensional generalization of the p-system:

.. math::
    \epsilon_t - u_x - v_y & = 0 \\
    \rho(x,y) u_t - \sigma(\epsilon,x,y)_x & = 0 \\
    \rho(x,y) v_t - \sigma(\epsilon,x,y)_y & = 0.

We take :math:'\sigma = e^{K(x,y)\epsilon} - 1', and the
material coefficients :math:'\rho,K' vary in a checkerboard
pattern.  The resulting dynamics lead to solitary waves,
though much more resolution is needed in order to see them.

This example shows how to set an aux array, use a b4step function,
```

```
use gauges, compute output functionals, and restart a simulation
from a checkpoint.
"""

import numpy as np
from clawpack import riemann

def qinit(state,A,x0,y0,varx,vary):
    r""" Set initial conditions:
         Gaussian stress, zero velocities."""
    yy,xx = state.grid.c_centers
    stress=A*np.exp(-(xx-x0)**2/(2*varx)-(yy-y0)**2/(2*vary)) #sigma(@t=0)
    stress_rel=state.aux[2,:]
    K=state.aux[1,:]

    state.q[0,:,:]=np.where(stress_rel==1,1,0)*stress/K+np.where(stress_rel==2,1,0)*np.log(stress+1),
    state.q[1,:,:]=0; state.q[2,:,:]=0

def setaux(x,y, KA=1, KB=4, rhoA=1, rhoB=4, stress_rel=2):
    r"""Return an array containing the values of the material
        coefficients.

        aux[0,i,j] = rho(x_i, y_j)              (material density)
        aux[1,i,j] = K(x_i, y_j)                (bulk modulus)
        aux[2,i,j] = stress-strain relation type at (x_i, y_j)
    """
    alphax=0.5; deltax=1.
    alphay=0.5; deltay=1.

    medium_type = 'checkerboard'

    aux = np.empty((4,len(x),len(y)), order='F')
    if medium_type == 'checkerboard':
        # xfrac and yfrac are x and y relative to deltax and deltay resp.
        xfrac=x-np.floor(x/deltax)*deltax
        yfrac=y-np.floor(y/deltay)*deltay
        # create a meshgrid out of xfrac and yfrac
        [yf,xf]=np.meshgrid(yfrac,xfrac)
        # density
        aux[0,:,:]=rhoA*(xf<=alphax*deltax)*(yf<=alphay*deltay)\
                   +rhoA*(xf >alphax*deltax)*(yf >alphay*deltay)\
                   +rhoB*(xf >alphax*deltax)*(yf<=alphay*deltay)\
                   +rhoB*(xf<=alphax*deltax)*(yf >alphay*deltay)
        #Young's modulus
        aux[1,:,:]=KA*(xf<=alphax*deltax)*(yf<=alphay*deltay)\
                   +KA*(xf >alphax*deltax)*(yf >alphay*deltay)\
                   +KB*(xf >alphax*deltax)*(yf<=alphay*deltay)\
                   +KB*(xf<=alphax*deltax)*(yf >alphay*deltay)
        # linearity of material
        aux[2,:,:]=stress_rel
    elif medium_type == 'sinusoidal' or medium_type == 'smooth_checkerboard':
        [yy,xx]=np.meshgrid(y,x)
        Amp_rho=np.abs(rhoA-rhoB)/2; offset_p=(rhoA+rhoB)/2
        Amp_K=np.abs(KA-KB)/2; offset_E=(KA+KB)/2
        if medium_type == 'sinusoidal':
            frec_x=2*np.pi/deltax; frec_y=2*np.pi/deltay
            fun=np.sin(frec_x*xx)*np.sin(frec_y*yy)
        else:
```

```
            sharpness=10
            fun_x=xx*0; fun_y=yy*0
            for i in xrange(0,1+int(np.ceil((x[-1]-x[0])/(deltax*0.5)))):
                fun_x=fun_x+(-1)**i*np.tanh(sharpness*(xx-deltax*i*0.5))
            for i in xrange(0,1+int(np.ceil((y[-1]-y[0])/(deltay*0.5)))):
                fun_y=fun_y+(-1)**i*np.tanh(sharpness*(yy-deltay*i*0.5))
            fun=fun_x*fun_y
        aux[0,:,:]=Amp_rho*fun+offset_p
        aux[1,:,:]=Amp_K*fun+offset_E
        aux[2,:,:]=stress_rel
    return aux

def b4step(solver,state):
    r"""Put in aux[3,:,:] the value of q[0,:,:] (eps).
        This is required in rptpv.f.
        Only used by classic (not SharpClaw).
    """
    state.aux[3,:,:] = state.q[0,:,:]

def compute_stress(state):
    """ Compute stress from strain and store in state.p."""
    K=state.aux[1,:,:]
    stress_rel=state.aux[2,:,:]
    eps=state.q[0,:,:]
    state.p[0,:,:] = np.where(stress_rel==1,1,0) * K*eps \
                    +np.where(stress_rel==2,1,0) * (np.exp(eps*K)-1) \


def total_energy(state):
    rho = state.aux[0,:,:]; K = state.aux[1,:,:]

    u = state.q[1,:,:]/rho
    v = state.q[2,:,:]/rho
    kinetic=rho * (u**2 + v**2)/2.

    eps = state.q[0,:,:]
    sigma = np.exp(K*eps) - 1.
    potential = (sigma-np.log(sigma+1.))/K

    dx=state.grid.delta[0]; dy=state.grid.delta[1]

    state.F[0,:,:] = (potential+kinetic)*dx*dy

def gauge_stress(q,aux):
    p = np.exp(q[0]*aux[1])-1
    return [p,10*p]


def setup(kernel_language='Fortran',
              use_petsc=False,outdir='./_output',solver_type='classic',
              disable_output=False, cells_per_layer=30, tfinal=18.):

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    # material parameters
```

```
KA=1.;   rhoA=1.
KB=4.;   rhoB=4.
stress_rel=2;

# Domain
x_lower=0.25; x_upper=20.25
y_lower=0.25; y_upper=20.25
# cells per layer
mx=(x_upper-x_lower)*cells_per_layer;
my=(y_upper-y_lower)*cells_per_layer
# Initial condition parameters
initial_amplitude=10.
x0=0.25 # Center of initial perturbation
y0=0.25 # Center of initial perturbation
varx=0.5; vary=0.5 # Width of initial perturbation

num_output_times = 10

if solver_type == 'classic':
    solver = pyclaw.ClawSolver2D(riemann.psystem_2D)
    solver.dimensional_split=False
    solver.cfl_max = 0.9
    solver.cfl_desired = 0.8
    solver.limiters = pyclaw.limiters.tvd.superbee
elif solver_type=='sharpclaw':
    solver = pyclaw.SharpClawSolver2D(riemann.psystem_2D)

if kernel_language != 'Fortran':
    raise Exception('Unrecognized value of kernel_language for 2D psystem')

# Boundary conditions
solver.bc_lower[0] = pyclaw.BC.wall
solver.bc_upper[0] = pyclaw.BC.extrap
solver.bc_lower[1] = pyclaw.BC.wall
solver.bc_upper[1] = pyclaw.BC.extrap
solver.aux_bc_lower[0] = pyclaw.BC.wall
solver.aux_bc_upper[0] = pyclaw.BC.extrap
solver.aux_bc_lower[1] = pyclaw.BC.wall
solver.aux_bc_upper[1] = pyclaw.BC.extrap

solver.fwave = True
solver.before_step = b4step

#controller
claw = pyclaw.Controller()
claw.tfinal = tfinal
claw.solver = solver
claw.outdir = outdir

# restart options
restart_from_frame = None

if restart_from_frame is None:
    x = pyclaw.Dimension('x',x_lower,x_upper,mx)
    y = pyclaw.Dimension('y',y_lower,y_upper,my)
    domain = pyclaw.Domain([x,y])
    num_eqn = 3
    num_aux = 4
```

```
        state = pyclaw.State(domain,num_eqn,num_aux)
        state.mF = 1
        state.mp = 1

        grid = state.grid
        state.aux = setaux(grid.x.centers,grid.y.centers,KA,KB,rhoA,rhoB,stress_rel)
        #Initial condition
        qinit(state,initial_amplitude,x0,y0,varx,vary)

        claw.solution = pyclaw.Solution(state,domain)
        claw.num_output_times = num_output_times

    else:
        claw.solution = pyclaw.Solution(restart_from_frame, format='petsc',read_aux=False)
        claw.solution.state.mp = 1
        grid = claw.solution.domain.grid
        claw.solution.state.aux = setaux(grid.x.centers,grid.y.centers)
        claw.num_output_times = num_output_times - restart_from_frame
        claw.start_frame = restart_from_frame

    #claw.p_function = p_function
    if disable_output:
        claw.output_format = None
    claw.compute_F = total_energy
    claw.compute_p = compute_stress
    claw.write_aux_init = False

    grid.add_gauges([[0.25,0.25],[17.85,1.25],[3.25,18.75],[11.75,11.75]])
    solver.compute_gauge_values = gauge_stress
    state.keep_gauges = True
    claw.setplot = setplot
    claw.keep_copy = True

    return claw

#--------------------------
def setplot(plotdata):
#--------------------------
    """
    Specify what is to be plotted at each frame.
    Input:  plotdata, an instance of visclaw.data.ClawPlotData.
    Output: a modified version of plotdata.
    """
    from clawpack.visclaw import colormaps

    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Figure for strain
    plotfigure = plotdata.new_plotfigure(name='Stress', figno=0)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'Strain'
    plotaxes.xlimits = [0.,20.]
    plotaxes.ylimits = [0.,20.]
    plotaxes.scaled = True

    # Set up for item on these axes:
```

```
    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = stress
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.add_colorbar = True

    return plotdata

def stress(current_data):
    import numpy as np
    from psystem_2d import setaux
    aux = setaux(current_data.x[:,0],current_data.y[0,:])
    q = current_data.q
    return np.exp(aux[1,...]*q[0,...])-1.


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Shallow water flow on the sphere 2D shallow water equations on a spherical surface. The approximation of the three-dimensional equations is restricted to the surface of the sphere. Therefore only the solution on the surface is updated.

Reference: Logically Rectangular Grids and Finite Volume Methods for PDEs in Circular and Spherical Domains. By Donna A. Calhoun, Christiane Helzel, and Randall J. LeVeque SIAM Review 50 (2008), 723-752.

```
#!/usr/bin/env python
# encoding: utf-8
"""
Shallow water flow on the sphere
================================

2D shallow water equations on a spherical surface. The approximation of the
three-dimensional equations is restricted to the surface of the sphere.
Therefore only the solution on the surface is updated.

Reference: Logically Rectangular Grids and Finite Volume Methods for PDEs in
           Circular and Spherical Domains.
           By Donna A. Calhoun, Christiane Helzel, and Randall J. LeVeque
           SIAM Review 50 (2008), 723-752.
"""

import math
import os
import sys

import numpy as np

from clawpack import pyclaw
from clawpack import riemann
from clawpack.pyclaw.util import inplace_build

try:
    import problem
    import classic2

except ImportError:
```

```
    this_dir = os.path.dirname(__file__)
    if this_dir == '':
        this_dir = os.path.abspath('.')
    inplace_build(this_dir)

    try:
        # Now try to import again
        import problem
        import classic2
    except ImportError:
        print >> sys.stderr, "***\nUnable to import problem module or automatically build, try runnir
        raise


# Nondimensionalized radius of the earth
Rsphere = 1.0


def fortran_src_wrapper(solver,state,dt):
    """
    Wraps Fortran src2.f routine.
    src2.f contains the discretization of the source term.
    """
    # Some simplifications
    grid = state.grid

    # Get parameters and variables that have to be passed to the fortran src2
    # routine.
    mx, my = grid.num_cells[0], grid.num_cells[1]
    num_ghost = solver.num_ghost
    xlower, ylower = grid.lower[0], grid.lower[1]
    dx, dy = grid.delta[0], grid.delta[1]
    q = state.q
    aux = state.aux
    t = state.t

    # Call src2 function
    state.q = problem.src2(mx,my,num_ghost,xlower,ylower,dx,dy,q,aux,t,dt,Rsphere)


def mapc2p_sphere_nonvectorized(grid,mC):
    """
    Maps to points on a sphere of radius Rsphere. Nonvectorized version (slow).

    Takes as input: array_list made by x_coordinates, y_ccordinates in the map
                    space.

    Returns as output: array_list made by x_coordinates, y_ccordinates in the
                       physical space.

    Inputs: mC = list composed by two arrays
                 [array ([xc1, xc2, ...]), array([yc1, yc2, ...])]

    Output: pC = list composed by three arrays
                 [array ([xp1, xp2, ...]), array([yp1, yp2, ...]), array([zp1, zp2, ...])]

    NOTE: this function is not used in the standard script.
    """
```

```
    # Get number of cells in both directions
    mx, my = grid.num_cells[0], grid.num_cells[1]

    # Define new list of numpy array, pC = physical coordinates
    pC = []

    for i in range(mx):
        for j in range(my):
            xc = mC[0][i][j]
            yc = mC[1][i][j]

            # Ghost cell values outside of [-3,1]x[-1,1] get mapped to other
            # hemisphere:
            if (xc >= 1.0):
                xc = xc - 4.0
            if (xc <= -3.0):
                xc = xc + 4.0

            if (yc >= 1.0):
                yc = 2.0 - yc
                xc = -2.0 - xc

            if (yc <= -1.0):
                yc = -2.0 - yc
                xc = -2.0 - xc

            if (xc <= -1.0):
                # Points in [-3,-1] map to lower hemisphere - reflect about x=-1
                # to compute x,y mapping and set sgnz appropriately:
                xc = -2.0 - xc
                sgnz = -1.0
            else:
                sgnz = 1.0

            sgnxc = math.copysign(1.0,xc)
            sgnyc = math.copysign(1.0,yc)

            xc1 = np.abs(xc)
            yc1 = np.abs(yc)
            d = np.maximum(np.maximum(xc1,yc1), 1.0e-10)

            DD = Rsphere*d*(2.0 - d) / np.sqrt(2.0)
            R = Rsphere
            centers = DD - np.sqrt(np.maximum(R**2 - DD**2, 0.0))

            xp = DD/d * xc1
            yp = DD/d * yc1

            if (yc1 >= xc1):
                yp = centers + np.sqrt(np.maximum(R**2 - xp**2, 0.0))
            else:
                xp = centers + np.sqrt(np.maximum(R**2 - yp**2, 0.0))

            # Compute physical coordinates
            zp = np.sqrt(np.maximum(Rsphere**2 - (xp**2 + yp**2), 0.0))
            pC.append(xp*sgnxc)
            pC.append(yp*sgnyc)
            pC.append(zp*sgnz)
```

```
    return pC


def mapc2p_sphere_vectorized(grid,mC):
    """
    Maps to points on a sphere of radius Rsphere. Vectorized version (fast).

    Takes as input: array_list made by x_coordinates, y_ccordinates in the map
                    space.

    Returns as output: array_list made by x_coordinates, y_ccordinates in the
                       physical space.

    Inputs: mC = list composed by two arrays
                 [array ([xc1, xc2, ...]), array([yc1, yc2, ...])]

    Output: pC = list composed by three arrays
                 [array ([xp1, xp2, ...]), array([yp1, yp2, ...]), array([zp1, zp2, ...])]

    NOTE: this function is used in the standard script.
    """

    # Get number of cells in both directions
    mx, my = grid.num_cells[0], grid.num_cells[1]

    # 2D array useful for the vectorization of the function
    sgnz = np.ones((mx,my))

    # 2D coordinates in the computational domain
    xc = mC[0][:][:]
    yc = mC[1][:][:]

    # Compute 3D coordinates in the physical domain
    # ===============================================

    # Note: yc < -1 => second copy of sphere:
    ij2 = np.where(yc < -1.0)
    xc[ij2] = -xc[ij2] - 2.0;
    yc[ij2] = -yc[ij2] - 2.0;

    ij = np.where(xc < -1.0)
    xc[ij] = -2.0 - xc[ij]
    sgnz[ij] = -1.0;
    xc1 = np.abs(xc)
    yc1 = np.abs(yc)
    d = np.maximum(xc1,yc1)
    d = np.maximum(d, 1e-10)
    D = Rsphere*d*(2-d) / np.sqrt(2)
    R = Rsphere*np.ones((np.shape(d)))

    centers = D - np.sqrt(R**2 - D**2)
    xp = D/d * xc1
    yp = D/d * yc1

    ij = np.where(yc1==d)
    yp[ij] = centers[ij] + np.sqrt(R[ij]**2 - xp[ij]**2)
    ij = np.where(xc1==d)
    xp[ij] = centers[ij] + np.sqrt(R[ij]**2 - yp[ij]**2)
```

```
    # Define new list of numpy array, pC = physical coordinates
    pC = []

    xp = np.sign(xc) * xp
    yp = np.sign(yc) * yp
    zp = sgnz * np.sqrt(Rsphere**2 - (xp**2 + yp**2))

    pC.append(xp)
    pC.append(yp)
    pC.append(zp)

    return pC


def qinit(state,mx,my):
    r"""
    Initialize solution with 4-Rossby-Haurwitz wave.

    NOTE: this function is not used in the standard script.
    """
    # Parameters
    a = 6.37122e6     # Radius of the earth
    Omega = 7.292e-5  # Rotation rate
    G = 9.80616       # Gravitational acceleration

    K = 7.848e-6
    t0 = 86400.0
    h0 = 8.e3         # Minimum fluid height at the poles
    R = 4.0

    # Compute the the physical coordinates of the cells' centerss
    state.grid.compute_p_centers(recompute=True)

    for i in range(mx):
        for j in range(my):
            xp = state.grid.p_centers[0][i][j]
            yp = state.grid.p_centers[1][i][j]
            zp = state.grid.p_centers[2][i][j]

            rad = np.maximum(np.sqrt(xp**2 + yp**2),1.e-6)

            if (xp >= 0.0 and yp >= 0.0):
                theta = np.arcsin(yp/rad)
            elif (xp <= 0.0 and yp >= 0.0):
                theta = np.pi - np.arcsin(yp/rad)
            elif (xp <= 0.0 and yp <= 0.0):
                 theta = -np.pi + np.arcsin(-yp/rad)
            elif (xp >= 0.0 and yp <= 0.0):
                theta = -np.arcsin(-yp/rad)

            # Compute phi, at north pole: pi/2 at south pool: -pi/2
            if (zp >= 0.0):
                phi =  np.arcsin(zp/Rsphere)
            else:
                phi = -np.arcsin(-zp/Rsphere)

            xp = theta
            yp = phi
```

```
            bigA = 0.5*K*(2.0*Omega + K)*np.cos(yp)**2.0 + \
                   0.25*K*K*np.cos(yp)**(2.0*R)*((1.0*R+1.0)*np.cos(yp)**2.0 + \
                   (2.0*R*R - 1.0*R - 2.0) - 2.0*R*R*(np.cos(yp))**(-2.0))
            bigB = (2.0*(Omega + K)*K)/((1.0*R + 1.0)*(1.0*R + 2.0)) * \
                   np.cos(yp)**R*( (1.0*R*R + 2.0*R + 2.0) - \
                   (1.0*R + 1.0)**(2)*np.cos(yp)**2 )
            bigC = 0.25*K*K*np.cos(yp)**(2*R)*( (1.0*R + 1.0)* \
                   np.cos(yp)**2 - (1.0*R + 2.0))


            # Calculate local longitude-latitude velocity vector
            # ==================================================
            Uin = np.zeros(3)

            # Longitude (angular) velocity component
            Uin[0] = (K*np.cos(yp)+K*np.cos(yp)**(R-1.)*( R*np.sin(yp)**2.0 - \
                     np.cos(yp)**2.0)*np.cos(R*xp))*t0

            # Latitude (angular) velocity component
            Uin[1] = (-K*R*np.cos(yp)**(R-1.0)*np.sin(yp)*np.sin(R*xp))*t0

            # Radial velocity component
            Uin[2] = 0.0 # The fluid does not enter in the sphere


            # Calculate velocity vetor in cartesian coordinates
            # =================================================
            Uout = np.zeros(3)

            Uout[0] = (-np.sin(xp)*Uin[0]-np.sin(yp)*np.cos(xp)*Uin[1])
            Uout[1] = (np.cos(xp)*Uin[0]-np.sin(yp)*np.sin(xp)*Uin[1])
            Uout[2] = np.cos(yp)*Uin[1]

            # Set the initial condition
            # =========================
            state.q[0,i,j] =  h0/a + (a/G)*( bigA + bigB*np.cos(R*xp) + \
                              bigC*np.cos(2.0*R*xp))
            state.q[1,i,j] = state.q[0,i,j]*Uout[0]
            state.q[2,i,j] = state.q[0,i,j]*Uout[1]
            state.q[3,i,j] = state.q[0,i,j]*Uout[2]


def qbc_lower_y(state,dim,t,qbc,auxbc,num_ghost):
    """
    Impose periodic boundary condition to q at the bottom boundary for the
    sphere. This function does not work in parallel.
    """
    for j in range(num_ghost):
        qbc1D = np.copy(qbc[:,:,2*num_ghost-1-j])
        qbc[:,:,j] = qbc1D[:,::-1]


def qbc_upper_y(state,dim,t,qbc,auxbc,num_ghost):
    """
    Impose periodic boundary condition to q at the top boundary for the sphere.
    This function does not work in parallel.
    """
```

```
    my = state.grid.num_cells[1]
    for j in range(num_ghost):
        qbc1D = np.copy(qbc[:,:,my+num_ghost-1-j])
        qbc[:,:,my+num_ghost+j] = qbc1D[:,::-1]


def auxbc_lower_y(state,dim,t,qbc,auxbc,num_ghost):
    """
    Impose periodic boundary condition to aux at the bottom boundary for the
    sphere.
    """
    grid=state.grid

    # Get parameters and variables that have to be passed to the fortran src2
    # routine.
    mx, my = grid.num_cells[0], grid.num_cells[1]
    xlower, ylower = grid.lower[0], grid.lower[1]
    dx, dy = grid.delta[0],grid.delta[1]

    # Impose BC
    auxtemp = auxbc.copy()
    auxtemp = problem.setaux(mx,my,num_ghost,mx,my,xlower,ylower,dx,dy,auxtemp,Rsphere)
    auxbc[:,:,:num_ghost] = auxtemp[:,:,:num_ghost]

def auxbc_upper_y(state,dim,t,qbc,auxbc,num_ghost):
    """
    Impose periodic boundary condition to aux at the top boundary for the
    sphere.
    """
    grid=state.grid

    # Get parameters and variables that have to be passed to the fortran src2
    # routine.
    mx, my = grid.num_cells[0], grid.num_cells[1]
    xlower, ylower = grid.lower[0], grid.lower[1]
    dx, dy = grid.delta[0],grid.delta[1]

    # Impose BC
    auxtemp = auxbc.copy()
    auxtemp = problem.setaux(mx,my,num_ghost,mx,my,xlower,ylower,dx,dy,auxtemp,Rsphere)
    auxbc[:,:,-num_ghost:] = auxtemp[:,:,-num_ghost:]


def setup(use_petsc=False,solver_type='classic',outdir='./_output', disable_output=False):
    if use_petsc:
        raise Exception("petclaw does not currently support mapped grids (go bug Lisandro who promise

    if solver_type != 'classic':
        raise Exception("Only Classic-style solvers (solver_type='classic') are supported on mapped g

    solver = pyclaw.ClawSolver2D(riemann.shallow_sphere_2D)
    solver.fmod = classic2

    # Set boundary conditions
    # ======================
    solver.bc_lower[0] = pyclaw.BC.periodic
    solver.bc_upper[0] = pyclaw.BC.periodic
    solver.bc_lower[1] = pyclaw.BC.custom  # Custom BC for sphere
```

```
    solver.bc_upper[1] = pyclaw.BC.custom  # Custom BC for sphere

    solver.user_bc_lower = qbc_lower_y
    solver.user_bc_upper = qbc_upper_y

    # Auxiliary array
    solver.aux_bc_lower[0] = pyclaw.BC.periodic
    solver.aux_bc_upper[0] = pyclaw.BC.periodic
    solver.aux_bc_lower[1] = pyclaw.BC.custom  # Custom BC for sphere
    solver.aux_bc_upper[1] = pyclaw.BC.custom  # Custom BC for sphere

    solver.user_aux_bc_lower = auxbc_lower_y
    solver.user_aux_bc_upper = auxbc_upper_y


    # Dimensional splitting ?
    # =======================
    solver.dimensional_split = 0

    # Transverse increment waves and transverse correction waves are computed
    # and propagated.
    # ============================================================================
    solver.transverse_waves = 2

    # Use source splitting method
    # ===========================
    solver.source_split = 2

    # Set source function
    # ===================
    solver.step_source = fortran_src_wrapper

    # Set the limiter for the waves
    # =============================
    solver.limiters = pyclaw.limiters.tvd.MC


    #============================================================================
    # Initialize domain and state, then initialize the solution associated to the
    # state and finally initialize aux array
    #============================================================================
    # Domain:
    xlower = -3.0
    xupper = 1.0
    mx = 40

    ylower = -1.0
    yupper = 1.0
    my = 20

    # Check whether or not the even number of cells are used in in both
    # directions. If odd numbers are used a message is print at screen and the
    # simulation is interrputed.
    if(mx % 2 != 0 or my % 2 != 0):
        message = 'Please, use even numbers of cells in both direction. ' \
                  'Only even numbers allow to impose correctly the boundary ' \
                  'conditions!'
        raise ValueError(message)
```

```
x = pyclaw.Dimension('x',xlower,xupper,mx)
y = pyclaw.Dimension('y',ylower,yupper,my)
domain = pyclaw.Domain([x,y])
dx = domain.grid.delta[0]
dy = domain.grid.delta[1]

# Define some parameters used in Fortran common blocks
solver.fmod.comxyt.dxcom = dx
solver.fmod.comxyt.dycom = dy
solver.fmod.sw.g = 11489.57219
solver.rp.comxyt.dxcom = dx
solver.rp.comxyt.dycom = dy
solver.rp.sw.g = 11489.57219

# Define state object
# ====================
num_aux = 16 # Number of auxiliary variables
state = pyclaw.State(domain,solver.num_eqn,num_aux)

# Override default mapc2p function
# ================================
state.grid.mapc2p = mapc2p_sphere_vectorized


# Set auxiliary variables
# =======================

# Get lower left corner coordinates
xlower,ylower = state.grid.lower[0],state.grid.lower[1]

num_ghost = 2
auxtmp = np.ndarray(shape=(num_aux,mx+2*num_ghost,my+2*num_ghost), dtype=float, order='F')
auxtmp = problem.setaux(mx,my,num_ghost,mx,my,xlower,ylower,dx,dy,auxtmp,Rsphere)
state.aux[:,:,:] = auxtmp[:,num_ghost:-num_ghost,num_ghost:-num_ghost]

# Set index for capa
state.index_capa = 0

# Set initial conditions
# ======================
# 1) Call fortran function
qtmp = np.ndarray(shape=(solver.num_eqn,mx+2*num_ghost,my+2*num_ghost), dtype=float, order='F')
qtmp = problem.qinit(mx,my,num_ghost,mx,my,xlower,ylower,dx,dy,qtmp,auxtmp,Rsphere)
state.q[:,:,:] = qtmp[:,num_ghost:-num_ghost,num_ghost:-num_ghost]

# 2) call python function define above
#qinit(state,mx,my)


#============================================================================
# Set up controller and controller parameters
#============================================================================
claw = pyclaw.Controller()
claw.keep_copy = True
if disable_output:
    claw.output_format = None
claw.output_style = 1
```

```
    claw.num_output_times = 10
    claw.tfinal = 10
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.outdir = outdir

    return claw


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup)


def plot_on_sphere():
    """
    Plots the solution of the shallow water on a sphere in the
    rectangular computational domain. The user can specify the name of the solution
    file and its path. If these are not given, the script checks
    whether the solution fort.q0000 in ./_output exists and plots it. If it it does
    not exist an error message is printed at screen.
    The file must be ascii and clawpack format.

    To use this, you must first install the basemap toolkit; see
    http://matplotlib.org/basemap/users/installing.html.

    This function also shows how to manually read and plot the solution stored in
    an ascii file written by pyclaw, without using the pyclaw.io.ascii routines.
    """

    import matplotlib.pyplot as plt

    # Nondimensionalized radius of the earth
    Rsphere = 1.0

    def contourLineSphere(fileName='fort.q0000',path='./_output'):
        """
        This function plots the contour lines on a spherical surface for the shallow
        water equations solved on a sphere.
        """

        # Open file
        # =========

        # Concatenate path and file name
        pathFileName = path + "/" + fileName

        f = file(pathFileName,"r")

        # Read file header
        # ================
        # The information contained in the first two lines are not used.
        unused = f.readline()  # patch_number
        unused = f.readline() # AMR_level

        # Read mx, my, xlow, ylow, dx and dy
        line = f.readline()
        sline = line.split()
```

```
mx = int(sline[0])

line = f.readline()
sline = line.split()
my = int(sline[0])

line = f.readline()
sline = line.split()
xlower = float(sline[0])

line = f.readline()
sline = line.split()
ylower = float(sline[0])

line = f.readline()
sline = line.split()
dx = float(sline[0])

line = f.readline()
sline = line.split()
dy = float(sline[0])


# Patch:
# ====
xupper = xlower + mx * dx
yupper = ylower + my * dy

x = pyclaw.Dimension('x',xlower,xupper,mx)
y = pyclaw.Dimension('y',ylower,yupper,my)
patch = pyclaw.Patch([x,y])


# Override default mapc2p function
# ================================
patch.mapc2p = mapc2p_sphere_vectorized


# Compute the physical coordinates of each cell's centers
# =======================================================
patch.compute_p_centers(recompute=True)
xp = patch._p_centers[0]
yp = patch._p_centers[1]
zp = patch._p_centers[2]

patch.compute_c_centers(recompute=True)
xc = patch._c_centers[0]
yc = patch._c_centers[1]

# Define arrays of conserved variables
h = np.zeros((mx,my))
hu = np.zeros((mx,my))
hv = np.zeros((mx,my))
hw = np.zeros((mx,my))

# Read solution
for j in range(my):
    tmp = np.fromfile(f,dtype='float',sep=" ",count=4*mx)
```

```
            tmp = tmp.reshape((mx,4))
            h[:,j] = tmp[:,0]
            hu[:,j] = tmp[:,1]
            hv[:,j] = tmp[:,2]
            hw[:,j] = tmp[:,3]


        # Plot solution in the computational domain
        # =========================================

        # Fluid height
        plt.figure()
        CS = plt.contour(xc,yc,h)
        plt.title('Fluid height (computational domain)')
        plt.xlabel('xc')
        plt.ylabel('yc')
        plt.clabel(CS, inline=1, fontsize=10)
        plt.show()
```

2D shallow water: radial dam break Solve the 2D shallow water equations:

The initial condition is a circular area with high depth surrounded by lower-depth water. The top and right boundary conditions reflect, while the bottom and left boundaries are outflow.

```
#!/usr/bin/env python
# encoding: utf-8
r"""
2D shallow water: radial dam break
==================================

Solve the 2D shallow water equations:

.. :math:
    h_t + (hu)_x + (hv)_y & = 0 \\
    (hu)_t + (hu^2 + \frac{1}{2}gh^2)_x + (huv)_y & = 0 \\
    (hv)_t + (huv)_x + (hv^2 + \frac{1}{2}gh^2)_y & = 0.

The initial condition is a circular area with high depth surrounded by lower-depth water.
The top and right boundary conditions reflect, while the bottom and left boundaries
are outflow.
"""

import numpy as np
from clawpack import riemann
from clawpack.riemann.shallow_roe_with_efix_2D_constants import depth, x_momentum, y_momentum, num_ec

def qinit(state,h_in=2.,h_out=1.,dam_radius=0.5):
    x0=0.
    y0=0.
    X, Y = state.p_centers
    r = np.sqrt((X-x0)**2 + (Y-y0)**2)

    state.q[depth     ,:,:] = h_in*(r<=dam_radius) + h_out*(r>dam_radius)
    state.q[x_momentum,:,:] = 0.
    state.q[y_momentum,:,:] = 0.


def setup(use_petsc=False,outdir='./_output',solver_type='classic'):
```

```
    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if solver_type == 'classic':
        solver = pyclaw.ClawSolver2D(riemann.shallow_roe_with_efix_2D)
        solver.limiters = pyclaw.limiters.tvd.MC
        solver.dimensional_split=1
    elif solver_type == 'sharpclaw':
        solver = pyclaw.SharpClawSolver2D(riemann.shallow_roe_with_efix_2D)

    solver.bc_lower[0] = pyclaw.BC.extrap
    solver.bc_upper[0] = pyclaw.BC.wall
    solver.bc_lower[1] = pyclaw.BC.extrap
    solver.bc_upper[1] = pyclaw.BC.wall

    # Domain:
    xlower = -2.5
    xupper = 2.5
    mx = 150
    ylower = -2.5
    yupper = 2.5
    my = 150
    x = pyclaw.Dimension('x',xlower,xupper,mx)
    y = pyclaw.Dimension('y',ylower,yupper,my)
    domain = pyclaw.Domain([x,y])

    state = pyclaw.State(domain,num_eqn)

    # Gravitational constant
    state.problem_data['grav'] = 1.0

    qinit(state)

    claw = pyclaw.Controller()
    claw.tfinal = 2.5
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.outdir = outdir
    claw.num_output_times = 10
    claw.setplot = setplot
    claw.keep_copy = True

    return claw

#--------------------------
def setplot(plotdata):
#--------------------------
    """
    Specify what is to be plotted at each frame.
    Input:  plotdata, an instance of visclaw.data.ClawPlotData.
    Output: a modified version of plotdata.
    """
    from clawpack.visclaw import colormaps

    plotdata.clearfigures()  # clear any old figures,axes,items data
```

```
    # Figure for depth
    plotfigure = plotdata.new_plotfigure(name='Water height', figno=0)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.xlimits = [-2.5, 2.5]
    plotaxes.ylimits = [-2.5, 2.5]
    plotaxes.title = 'Water height'
    plotaxes.scaled = True

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = 0
    plotitem.pcolor_cmap = colormaps.red_yellow_blue
    plotitem.pcolor_cmin = 0.5
    plotitem.pcolor_cmax = 1.5
    plotitem.add_colorbar = True

    # Scatter plot of depth
    plotfigure = plotdata.new_plotfigure(name='Scatter plot of h', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.xlimits = [0., 2.5]
    plotaxes.ylimits = [0., 2.1]
    plotaxes.title = 'Scatter plot of h'

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d_from_2d_data')
    plotitem.plot_var = depth
    def q_vs_radius(current_data):
        from numpy import sqrt
        x = current_data.x
        y = current_data.y
        r = sqrt(x**2 + y**2)
        q = current_data.q[depth,:,:]
        return r,q
    plotitem.map_2d_to_1d = q_vs_radius
    plotitem.plotstyle = 'o'


    # Figure for x-momentum
    plotfigure = plotdata.new_plotfigure(name='Momentum in x direction', figno=2)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.xlimits = [-2.5, 2.5]
    plotaxes.ylimits = [-2.5, 2.5]
    plotaxes.title = 'Momentum in x direction'
    plotaxes.scaled = True

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = x_momentum
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.add_colorbar = True
    plotitem.show = False       # show on plot?
```

```
    # Figure for y-momentum
    plotfigure = plotdata.new_plotfigure(name='Momentum in y direction', figno=3)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.xlimits = [-2.5, 2.5]
    plotaxes.ylimits = [-2.5, 2.5]
    plotaxes.title = 'Momentum in y direction'
    plotaxes.scaled = True

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = y_momentum
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.add_colorbar = True
    plotitem.show = False       # show on plot?

    return plotdata


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

A non-convex flux scalar model Solve the KPP equation:

$$q_t + (\sin(q))_x + (\cos(q))_y = 0$$

first proposed by Kurganov, Petrova, and Popov. It is challenging for schemes with low numerical viscosity to capture the solution accurately.

```
#!/usr/bin/env python
# encoding: utf-8
r"""
A non-convex flux scalar model
==============================

Solve the KPP equation:

.. math::
    q_t + (\sin(q))_x + (\cos(q))_y & = 0

first proposed by Kurganov, Petrova, and Popov.  It is challenging for schemes
with low numerical viscosity to capture the solution accurately.
"""
import numpy as np
from clawpack import riemann

def setup(use_petsc=False,outdir='./_output',solver_type='classic'):

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if solver_type=='sharpclaw':
        solver = pyclaw.SharpClawSolver2D(riemann.kpp_2D)
```

```
    else:
        solver = pyclaw.ClawSolver2D(riemann.kpp_2D)
        solver.dimensional_split = 1
        solver.cfl_max = 1.0
        solver.cfl_desired = 0.9
        solver.limiters = pyclaw.limiters.tvd.minmod

    solver.bc_lower[0]=pyclaw.BC.extrap
    solver.bc_upper[0]=pyclaw.BC.extrap
    solver.bc_lower[1]=pyclaw.BC.extrap
    solver.bc_upper[1]=pyclaw.BC.extrap

    # Initialize domain
    mx=200; my=200
    x = pyclaw.Dimension('x',-2.0,2.0,mx)
    y = pyclaw.Dimension('y',-2.0,2.0,my)
    domain = pyclaw.Domain([x,y])
    state = pyclaw.State(domain,solver.num_eqn)

    # Initial data
    X, Y = state.grid.p_centers
    r = np.sqrt(X**2 + Y**2)
    state.q[0,:,:] = 0.25*np.pi + 3.25*np.pi*(r<=1.0)

    claw = pyclaw.Controller()
    claw.tfinal = 1.0
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.setplot = setplot
    claw.keep_copy = True

    return claw

#--------------------------
def setplot(plotdata):
#--------------------------
    """
    Specify what is to be plotted at each frame.
    Input:  plotdata, an instance of visclaw.data.ClawPlotData.
    Output: a modified version of plotdata.
    """
    from clawpack.visclaw import colormaps

    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Figure for pcolor plot
    plotfigure = plotdata.new_plotfigure(name='q[0]', figno=0)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'q[0]'
    plotaxes.afteraxes = "pylab.axis('scaled')"

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_pcolor')
    plotitem.plot_var = 0
    plotitem.pcolor_cmap = colormaps.yellow_red_blue
    plotitem.pcolor_cmin = 0.0
```

```
    plotitem.pcolor_cmax = 3.5*3.14
    plotitem.add_colorbar = True

    # Figure for contour plot
    plotfigure = plotdata.new_plotfigure(name='contour', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'q[0]'
    plotaxes.afteraxes = "pylab.axis('scaled')"

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='2d_contour')
    plotitem.plot_var = 0
    plotitem.contour_nlevels = 20
    plotitem.contour_min = 0.01
    plotitem.contour_max = 3.5*3.15
    plotitem.amr_contour_colors = ['b','k','r']

    return plotdata


if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Solitary wave formation in periodic nonlinear elastic media Solve a one-dimensional nonlinear elasticity system:

$$\epsilon_t + u_x = 0$$
$$(\rho(x)u)_t + \sigma(\epsilon, x)_x = 0.$$

Here ϵ is the strain, σ is the stress, u is the velocity, and $\rho(x)$ is the density. We take the stress-strain relation $\sigma = e^{K(x)\epsilon} - 1$; $K(x)$ is the linearized bulk modulus. Note that the density and bulk modulus may depend explicitly on x.

The problem solved here is based on [LeVYon03] (page 293). An initial hump evolves into two trains of solitary waves.

```
#!/usr/bin/env python
# encoding: utf-8
r"""
Solitary wave formation in periodic nonlinear elastic media
=============================================================

Solve a one-dimensional nonlinear elasticity system:

.. math::
    \epsilon_t + u_x & = 0 \\
    (\rho(x) u)_t + \sigma(\epsilon,x)_x & = 0.

Here :math:`\epsilon` is the strain, :math:`\sigma` is the stress,
u is the velocity, and :math:`\rho(x)` is the density.
We take the stress-strain relation :math:`\sigma = e^{K(x)\epsilon}-1`;
:math:`K(x)` is the linearized bulk modulus.
Note that the density and bulk modulus may depend explicitly on x.

The problem solved here is based on [LeVYon03]_.  An initial hump
```

```
evolves into two trains of solitary waves.

"""
import numpy as np


def qinit(state,ic=2,a2=1.0,xupper=600.):
    x = state.grid.x.centers

    if ic==1: #Zero ic
        state.q[:,:] = 0.
    elif ic==2:
        # Gaussian
        sigma = a2*np.exp(-((x-xupper/2.)/10.)**2.)
        state.q[0,:] = np.log(sigma+1.)/state.aux[1,:]
        state.q[1,:] = 0.


def setaux(x,rhoB=4,KB=4,rhoA=1,KA=1,alpha=0.5,xlower=0.,xupper=600.,bc=2):
    aux = np.empty([3,len(x)],order='F')
    xfrac = x-np.floor(x)
    #Density:
    aux[0,:] = rhoA*(xfrac<alpha)+rhoB*(xfrac>=alpha)
    #Bulk modulus:
    aux[1,:] = KA  *(xfrac<alpha)+KB  *(xfrac>=alpha)
    aux[2,:] = 0. # not used
    return aux


def b4step(solver,state):
    #Reverse velocity at trtime
    #Note that trtime should be an output point
    if state.t>=state.problem_data['trtime']-1.e-10 and not state.problem_data['trdone']:
        #print 'Time reversing'
        state.q[1,:]=-state.q[1,:]
        state.q=state.q
        state.problem_data['trdone']=True
        if state.t>state.problem_data['trtime']:
            print 'WARNING: trtime is '+str(state.problem_data['trtime'])+\
                ' but velocities reversed at time '+str(state.t)
    #Change to periodic BCs after initial pulse
    if state.t>5*state.problem_data['tw1'] and solver.bc_lower[0]==0:
        solver.bc_lower[0]=2
        solver.bc_upper[0]=2
        solver.aux_bc_lower[0]=2
        solver.aux_bc_upper[0]=2


def zero_bc(state,dim,t,qbc,auxbc,num_ghost):
    """Set everything to zero"""
    if dim.on_upper_boundary:
        qbc[:,-num_ghost:]=0.

def moving_wall_bc(state,dim,t,qbc,auxbc,num_ghost):
    """Initial pulse generated at left boundary by prescribed motion"""
    if dim.on_lower_boundary:
        qbc[0,:num_ghost]=qbc[0,num_ghost]
        t=state.t; t1=state.problem_data['t1']; tw1=state.problem_data['tw1']
```

```
        a1=state.problem_data['a1'];
        t0 = (t-t1)/tw1
        if abs(t0)<=1.: vwall = -a1*(1.+np.cos(t0*np.pi))
        else: vwall=0.
        for ibc in xrange(num_ghost-1):
            qbc[1,num_ghost-ibc-1] = 2*vwall*state.aux[1,ibc] - qbc[1,num_ghost+ibc]


def setup(use_petsc=0,kernel_language='Fortran',solver_type='classic',outdir='./_output'):
    from clawpack import riemann

    if use_petsc:
        import clawpack.petclaw as pyclaw
    else:
        from clawpack import pyclaw

    if kernel_language=='Python':
        rs = riemann.nonlinear_elasticity_1D_py.nonlinear_elasticity_1D
    elif kernel_language=='Fortran':
        rs = riemann.nonlinear_elasticity_fwave_1D

    if solver_type=='sharpclaw':
        solver = pyclaw.SharpClawSolver1D(rs)
        solver.char_decomp=0
    else:
        solver = pyclaw.ClawSolver1D(rs)

    solver.kernel_language = kernel_language

    solver.bc_lower[0] = pyclaw.BC.custom
    solver.bc_upper[0] = pyclaw.BC.extrap

    #Use the same BCs for the aux array
    solver.aux_bc_lower[0] = pyclaw.BC.extrap
    solver.aux_bc_upper[0] = pyclaw.BC.extrap

    xlower=0.0; xupper=300.0
    cells_per_layer=12; mx=int(round(xupper-xlower))*cells_per_layer
    x = pyclaw.Dimension('x',xlower,xupper,mx)
    domain = pyclaw.Domain(x)
    state = pyclaw.State(domain,solver.num_eqn)

    #Set global parameters
    alpha = 0.5
    KA    = 1.0
    KB    = 4.0
    rhoA  = 1.0
    rhoB  = 4.0
    state.problem_data = {}
    state.problem_data['t1']    = 10.0
    state.problem_data['tw1']   = 10.0
    state.problem_data['a1']    = 0.1
    state.problem_data['alpha'] = alpha
    state.problem_data['KA'] = KA
    state.problem_data['KB'] = KB
    state.problem_data['rhoA'] = rhoA
    state.problem_data['rhoB'] = rhoB
```

```
    state.problem_data['trtime'] = 999999999.0
    state.problem_data['trdone'] = False

    #Initialize q and aux
    xc=state.grid.x.centers
    state.aux=setaux(xc,rhoB,KB,rhoA,KA,alpha,xlower=xlower,xupper=xupper)
    qinit(state,ic=1,a2=1.0,xupper=xupper)

    tfinal=500.; num_output_times = 20;

    solver.max_steps = 5000000
    solver.fwave = True
    solver.before_step = b4step
    solver.user_bc_lower=moving_wall_bc
    solver.user_bc_upper=zero_bc

    claw = pyclaw.Controller()
    claw.output_style = 1
    claw.num_output_times = num_output_times
    claw.tfinal = tfinal
    claw.solution = pyclaw.Solution(state,domain)
    claw.solver = solver
    claw.setplot = setplot
    claw.keep_copy = True

    return claw


#--------------------------
def setplot(plotdata):
#--------------------------
    """
    Specify what is to be plotted at each frame.
    Input:  plotdata, an instance of visclaw.data.ClawPlotData.
    Output: a modified version of plotdata.
    """
    plotdata.clearfigures()  # clear any old figures,axes,items data

    # Figure for q[0]
    plotfigure = plotdata.new_plotfigure(name='Stress', figno=1)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.title = 'Stress'
    plotaxes.ylimits = [-0.1,1.0]

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
    plotitem.plot_var = stress
    plotitem.kwargs = {'linewidth':2}

    # Figure for q[1]
    plotfigure = plotdata.new_plotfigure(name='Velocity', figno=2)

    # Set up for axes in this figure:
    plotaxes = plotfigure.new_plotaxes()
    plotaxes.xlimits = 'auto'
    plotaxes.ylimits = [-.5,0.1]
```

```
    plotaxes.title = 'Velocity'

    # Set up for item on these axes:
    plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
    plotitem.plot_var = velocity
    plotitem.kwargs = {'linewidth':2}

    return plotdata


def velocity(current_data):
    """Compute velocity from strain and momentum"""
    from stegoton import setaux
    aux=setaux(current_data.x,rhoB=4,KB=4)
    velocity = current_data.q[1,:]/aux[0,:]
    return velocity

def stress(current_data):
    """Compute stress from strain and momentum"""
    from stegoton import setaux
    from clawpack.riemann.nonlinear_elasticity_1D_py import sigma
    aux=setaux(current_data.x)
    epsilon = current_data.q[0,:]
    stress = sigma(epsilon,aux[1,:])
    return stress

if __name__=="__main__":
    from clawpack.pyclaw.util import run_app_from_main
    output = run_app_from_main(setup,setplot)
```

Adding new examples

If you have used PyClaw, we'd love to add your application to the built-in scripts. Please contact us on the claw-users Google group (http://http://groups.google.com/group/claw-users) or just issue a pull request on Github (http://github.com/clawpack/pyclaw/pulls).

Plotting PyClaw results

PyClaw relies on the VisClaw package (http://github.com/clawpack/visclaw/) for easy plotting, although it is of course possible to load the output into other visualization packages. VisClaw supports 1D and 2D plotting; for 3D plotting, we recommend using the old Clawpack MATLAB routines (http://depts.washington.edu/clawpack/users-4.6/matlab_plotting.html) for now.

This page gives some very basic information; for more detail, see *Visclaw Plotting options* (page 247) in VisClaw's documentation.

Basics

VisClaw includes routines for creating HTML and LaTex plot pages or plotting interactively. These require a *setplot.py* file that defines the plotting parameters; see *Plotting options in Python* (page 248). for more information. Once you have an appropriate *setplot.py* file, there are some convenience functions in *$PYCLAW/src/petclaw/plot.py* for generating these plots. Assuming you have output files in *./_output* (which is the default), you can generate HTML pages with plots from Python via

```
>>> from clawpack.pyclaw import plot
>>> plot.html_plot()
```

This will generate HTML pages with plots and print out a message with the location of the HTML file. To launch an interactive plotting session from within Python, do

```
>>> from clawpack.pyclaw import plot
>>> plot.interactive_plot()
```

To see a list of commands available in the resulting interactive environment, type "?" or see *Interactive plotting with Iplotclaw* (page 248).

Plotting result from parallel runs

By default, when running in parallel, PyClaw outputs data in a binary format. In order to plot from such files, just replace `pyclaw` with `petclaw` in the commands above; e.g.

```
>>> from clawpack.petclaw import plot
>>> plot.interactive_plot()
```

4.2.2 Going Further

Setting up your own problem

The best way to set up a new problem is to find an existing problem setup that is similar. The main steps are:

- Write the initialization script
- Write routines for source terms, custom boundary conditions, or other customizations
- Write a Riemann solver (if solving a new system of equations)
- Write a Makefile if using any custom Fortran code
- Write a setplot.py file for visualization

If needed for your problem, custom Riemann solvers, boundary condition routines, source term routines, and other functions can all be written in Python but you may prefer to write some of them in Fortran for performance reasons. The latter approach requires direct use of f2py (http://www.scipy.org/F2py). See *Porting a problem from Clawpack 4.6.x to PyClaw* (page 176) for more details.

Writing the initialization script

This script should:

- Import the appropriate package (pyclaw or petclaw)
- Instantiate a `Solver` and specify the Riemann solver to use
- Set the boundary conditions
- Define the domain through a `Domain` (page 230) object
- Define a `Solution` (page 221) object
- Set the initial condition

Usually the script then instantiates a `Controller` (page 206), sets the initial solution and solver, and calls `run()` (page 207).

Setting initial conditions Once you have initialized a Solution object, it contains a member state.q whose first dimension is num_eqn and whose remaining dimensions are those of the grid. Now you must set the initial condition. For instance

```
>>> import numpy as np
>>> Y,X = np.meshgrid(state.grid.y.centers,state.grid.x.centers)
>>> r = np.sqrt(X**2 + Y**2)
>>> width = 0.2
>>> state.q[0,:,:] = (np.abs(r-0.5)<=width)*(1.+np.cos(np.pi*(r-0.5)/width))
>>> state.q[1,:,:] = 0.
>>> state.q[2,:,:] = 0.
```

Note that in a parallel run we only wish to set the local values of the state so the appropriate geometry object to use here is the `Grid` (page 231) class.

Setting auxiliary variables If the problem involves coefficients that vary in space or a mapped grid, the required fields are stored in state.aux. In order to use such fields, you must pass the num_aux argument to the State initialization

```
>>> state = pyclaw.State(domain,solver.num_eqn,num_aux)
```

The number of fields in state.aux (i.e., the length of its first dimension) is set equal to num_aux. The values of state.aux are set in the same way as those of state.q.

Setting boundary conditions The boundary conditions are specified through solver.bc_lower and solver.bc_upper, each of which is a list of length `solver.num_dim`. The ordering of the boundary conditions in each list is the same as the ordering of the Dimensions in the Grid; typically (x, y). Thus `solver.bc_lower[0]` specifies the boundary condition at the left boundary and `solver.bc_upper[0]` specifies the condition at the right boundary. Similarly, `solver.bc_lower[1]` and `solver.bc_upper[1]` specify the boundary conditions at the top and bottom of the domain.

PyClaw includes the following built-in boundary condition implementations:

- `pyclaw.BC.periodic` - periodic
- `pyclaw.BC.extrap` - zero-order extrapolation
- `pyclaw.BC.wall` - solid wall conditions, assuming that the 2nd/3rd component of q is the normal velocity in x/y.

Other boundary conditions can be implemented by using `pyclaw.BC.custom`, and providing a custom BC function. The attribute solver.user_bc_lower/upper must be set to the corresponding function handle. For instance

```
>>> def custom_bc(state,dim,t,qbc,num_ghost):
...     for i in xrange(num_ghost):
...         qbc[0,i,:] = q0

>>> solver.bc_lower[0] = pyclaw.BC.custom
>>> solver.user_bc_lower = custom_bc
```

If the `state.aux` array is used, boundary conditions must be set for it in a similar way, using `solver.aux_bc_lower` and `solver.aux_bc_upper`. Note that although state is passed to the BC routines, they should NEVER modify state. Rather, they should modify qbc/auxbc.

Setting solver options

Using your own Riemann solver

The Riemann package has solvers for many hyperbolic systems. If your problem involves a new system, you will need to write your own Riemann solver. A nice example of how to compile and import your own Riemann solver can be seen **'here https://github.com/damiansra/empyclaw/tree/master/maxwell_1d_homogeneous'_**. You will need to:

- Put the Riemann solver in the same directory as your Python script
- Write a short makefile calling f2py
- import the Riemann solver module in your Python script

Here are some tips if you are converting an old Clawpack 4.5 or earlier Riemann solver:

- Rename the file from .f to .f90 and switch to free-format Fortran
- Move the spatial index (i) to the last place in all array indexing

Please do contribute your solver to the package by sending a pull request on Github or e-mailing one of the developers. To add your Riemann solver to the Clawpack Riemann package, you will need to:

- Place the .f90 file(s) in clawpack/riemann/src.
- Add the solver to the list in clawpack/riemann/setup.py
- Add the solver to the list in clawpack/riemann/src/python/riemann/setup.py
- Add the solver to the list in clawpack/riemann/src/python/riemann/Makefile
- Add the solver to the list in clawpack/riemann/src/python/riemann/__init__.py

For very simple problems in one dimension, it may be worthwhile to write the Riemann solver in Python, especially if you are more comfortable with Python than with Fortran. For two-dimensional problems, or one-dimensional problems requiring fine grids (or if you are impatient) the solver should be written in Fortran. The best approach is generally to find a similar solver in the Riemann package and modify it to solve your system.

Adding source terms

Non-hyperbolic terms (representing, e.g., reaction or diffusion) can be included in a PyClaw simulation by providing an appropriate function handle to

- solver.step_source if using Classic Clawpack. In this case, the function specified should modify q by taking a step on the equation $q_t = \psi(q)$.
- solver.dq_src if using SharpClaw. In this case, the function should return $\Delta t \cdot \psi(q)$.

For an example, see pyclaw/examples/euler_2d/shockbubble.py.

Setting up the Makefile

Generally you can just copy the Makefile from an example in pyclaw/examples and replace the value of *RP_SOURCES*. Make sure the example you choose has the same dimensionality. Also be sure to use the f-wave targets if your Riemann solver is an f-wave solver.

Porting a problem from Clawpack 4.6.x to PyClaw

The script *pyclaw/development/clawdata2pyclaw.py* is intended to aid in converting a Clawpack 4.6 problem setup to PyClaw. However, some manual conversion is necessary, especially if the problem includes custom fortran routines.

In PyClaw, the high-level portions of the Fortran routines are reorganized in an object-oriented Python framework, while the low-level ones are bound through the Fortran to Python interface generator f2py (http://www.scipy.org/F2py). Therefore, for simple problems you won't need to call f2py directly. However, if you want to reutilize some problem-specific fortran routines that were set up and tested in a Clawpack problem, you can easily do it. Indeed, if those routines are complicated and/or computationally intensive, you should consider directly using the f2py interface in the initialization script (see *Setting up your own problem* (page 174)). The example in *clawpack/pyclaw/examples/shallow_sphere*, which solves the shallow water equations on the surface of a sphere, is a complete example that relies heavily on the use of problem-specific Fortran routines. In that problem setup, a few Fortran routines have been used to provide the following functionality:

- Initialize the solution `state.q[:,:,:]`
- Provide the mapping from a uniform Cartesian domain to the desired physical domain, i.e. the mapc2p function
- Setup the auxiliary variables `state.aux[:,:,:]`
- Compute the (non-hyperbolic) contribution of a source term
- Impose custom boundary conditions to both solution and auxiliary variables

The first step to succesfully interface the Fortran functions with PyClaw is to automate the extension module generation of these routines through f2py. You can use *clawpack/pyclaw/examples/shallow_sphere/Makefile* as a template:

```
# Problem's source Fortran files
INITIALIZE_SOURCE = mapc2p.f setaux.f qinit.f src2.f
problem.so: $(INITIALIZE_SOURCE)
    $(F2PY) -m problem -c $^
```

The code above, calls f2py to compile a set of Fortran routines and build a module (`problem.so`) which can then be imported as a function in Python. The argument following the ''-m" flag is the name the python module should have (i.e. the name of the target). f2py uses the `numpy.distutils` module from NumPy that supports a number of major Fortran compilers. For more information see http://www.scipy.org/F2py.

After compilation, it is useful to check the signature of each function contained in `problem.so`, which may be different than that of the original Fortran function, since f2py eliminates dummy variables. One can easily achieve that by using the following commands:

```
$ ipython
>>> import problem
>>> problem?
```

The last command queries the content of the module and outputs the functions' signature that must be used in the initialization script to correctly call the fortran functions. In the shallow water equations on a sphere example, we get the following output:

```
>>> Type:           module
>>> Base Class:     <type 'module'>
>>> String Form:    <module 'problem' from 'problem.so'>
>>> Namespace:      Interactive
>>> File:           /Users/../../../clawpack/pyclaw/examples/shallow-sphere/problem.so
>>> Docstring:
    This module 'problem' is auto-generated with f2py (version:1).
    Functions:
    mapc2p(x1,y1,xp,yp,zp,rsphere)
    aux = setaux(maxmx,maxmy,num_ghost,mx,my,xlower,ylower,dxc,dyc,aux,rsphere,num_aux=shape(aux,0))
    q = qinit(maxmx,maxmy,num_ghost,mx,my,xlower,ylower,dx,dy,q,aux,rsphere,num_eqn=shape(q,0),num_au
    q = src2(maxmx,maxmy,num_ghost,xlower,ylower,dx,dy,q,aux,t,dt,rsphere,num_eqn=shape(q,0),mx=shape
```

For instance, the function `src2`, which computes the contribution of the (non-hyperbolic) source term, has the following intent variables:

```
>>> cf2py integer intent(in) maxmx
>>> cf2py integer intent(in) maxmy
>>> cf2py integer optional, intent(in) num_eqn
>>> cf2py integer intent(in) num_ghost
>>> cf2py integer intent(in) mx
>>> cf2py integer intent(in) my
>>> cf2py double precision intent(in) xlower
>>> cf2py double precision intent(in) ylower
>>> cf2py double precision intent(in) dx
>>> cf2py double precision intent(in) dy
>>> cf2py intent(in,out) q
>>> cf2py integer optional, intent(in)  num_aux
>>> cf2py intent(in) aux
>>> cf2py double precision intent(in) t
>>> cf2py double precision intent(in) dt
>>> cf2py double precision intent(in) Rsphere
```

Note that `num_eqn`, `mx`, `my` `num_aux` are identified by f2py as optional arguments since their values can be retrieved by looking at the dimensions of other multidimensional arrays, i.e. `q` and `aux`.

We are now ready to call and use the Fortran functions in the initialization script. For instance, the `src2` function is called in the script (http://numerics.kaust.edu.sa/pyclaw/examples/shallow-sphere/shallow_4_Rossby_Haurwitz_wave.py) by using a fortran_src_wrapper function whose main part reads:

```
>>> # Call src2 function
>>> import problem
>>> state.q = problem.src2(mx,my,num_ghost,xlowerg,ylowerg,dx,dy,q,aux,t,dt,Rsphere)
```

A similar approach is used to call other wrapped Fortran functions like `qinit`, `setaux`, etc.

An important feature that makes PyClaw more flexible is the capability to replace the standard low-level Fortran routines whith some problem-specific routines. Binding new low-level functions and replacing the standard ones is very easy; the user just needs to modify the problem-specific Makefile. The shallow water equations on a sphere is again a typical example that uses this nice feature. Indeed, to run correctly the problem an ad-hoc `step2` function (i.e. the `step2qcor`) is required. For that problem the interesting part of the Makefile (http://numerics.kaust.edu.sa/pyclaw/examples/shallow-sphere/shallow_4_Rossby_Haurwitz_wave.py) reads:

```
# Override step2.f with a new function that contains a call to an additional
# function, i.e. qcor.f
# ======================================================================
override TWO_D_CLASSIC_SOURCES = step2qcor.f qcor.o flux2.o limiter.o philim.o

qcor.o: qcor.f
    $(FC) $(FFLAGS) -o qcor.o -c qcor.f
```

The user has just to override `step2.f` with the new function `step2qcor.f` and provide new:

```
output_filenames : input_filenames
    actions
```

rules to create the targets required by the new Fortran routine. Similar changes to the problem-specific Makefile can be used to replace other low-level Fortran routines.

Contents

Using PyClaw's solvers: Classic and SharpClaw

At present, PyClaw includes two types of solvers:

- Classic: the original Clawpack algorithms, in 1/2/3D
- SharpClaw: higher-order wave propagation using WENO reconstruction and Runge-Kutta integration, in 1/2D

Solver initialization takes one argument: a Riemann solver, usually from the Riemann repository. Typically, all that is needed to select a different solver is to specify it in the problem script, e.g.

```
>>> from clawpack import pyclaw
>>> from clawpack import riemann
>>> solver = pyclaw.ClawSolver2D(riemann.acoustics_2D)
```

for the 2D acoustics equations and the Classic Clawpack solver or

```
>>> solver = pyclaw.SharpClawSolver2D(riemann.acoustics_2D)
```

for the SharpClaw solver. Most of the applications distributed with PyClaw are set up to use either solver, depending on the value of the command line option *solver_type*, which should be set to *classic* or *sharpclaw*.

Typically, for a given grid resolution, the SharpClaw solvers are more accurate but also more computationally expensive. For typical problems involving shocks, the Classic solvers are recommended. For problems involving high-frequency waves, turbulence, or smooth solutions, the SharpClaw solvers may give more accurate solutions at less cost. This is an active area of research and you may wish to experiment with both solvers.

Future plans include incorporation of finite-difference and discontinuous Galerkin solvers.

Key differences between the Classic and SharpClaw solvers are:

- The source term routine for the Classic solver should return the integral of the source term over a step, while the source term routine for SharpClaw should return the instantaneous value of the source term.
- The solvers have different options. For a list of options and possible values, see the documentation of the `ClawSolver` and `SharpClawSolver` (page 209) classes.

SharpClaw Solvers

The SharpClaw solvers are a collection of solvers that contain the functionality of the Fortran code SharpClaw, developed in David Ketcheson's thesis. The 1D SharpClaw solver contains a pure Python implementation as well as a wrapped Fortran version. The 2D solver is in progress but not available yet. The SharpClaw solvers provide an interface similar to that of the classic Clawpack solvers, but with a few different options. The superclass solvers are not meant to be used separately but are there to provide common routines for all the Clawpack solvers. Please refer to each of the inherited classes for more info about the methods and attributes they provide each class. .. The inheritance structure is:

Example This is a simple example of how to instantiate and evolve a solution to a later time

```
>>> from clawpack import pyclaw
>>> solver = pyclaw.SharpClawSolver1D()            # Instantiate a default, 1d solver

>>> solver.evolve_to_time(solution,t_end)   # Evolve the solution to t_end
```

pyclaw.sharpclaw

class pyclaw.sharpclaw.solver.**SharpClawSolver**(*riemann_solver=None*, *claw_package=None*)

Superclass for all SharpClawND solvers.

Implements Runge-Kutta time stepping and the basic form of a semi-discrete step (the dq() function). If another method-of-lines solver is implemented in the future, it should be based on this class, which then ought to be renamed to something like "MOLSolver".

before_step

Function called before each time step is taken. The required signature for this function is:

def before_step(solver,solution)

lim_type

Limiter(s) to be used. 0: No limiting. 1: TVD reconstruction. 2: WENO reconstruction. `Default = 2`

weno_order

Order of the WENO reconstruction. From 1st to 17th order (PyWENO) `Default = 5`

time_integrator

Time integrator to be used. Currently implemented methods:

'Euler' : 1st-order Forward Euler integration 'SSP33' : 3rd-order strong stability preserving method of Shu & Osher 'SSP104' : 4th-order strong stability preserving method Ketcheson 'SSPMS32': 2nd-order strong stability preserving 3-step linear multistep method,

Unexpected indentation.

using Euler for starting values

Block quote ends without a blank line; unexpected unindent.

'SSPMS43': 3rd-order strong stability preserving 4-step linear multistep method using SSPRK22 for starting values

'RK' [Arbitrary Runge-Kutta method, specified by setting *solver.a*] and *solver.b* to the Butcher arrays of the method.

'LMM' [Arbitrary linear multistep method, specified by setting the] coefficient arrays *solver.alpha* and *solver.beta*.

`Default = 'SSP104'`

char_decomp

Type of WENO reconstruction. 0: conservative variables WENO reconstruction (standard). 1: Wave-slope reconstruction. 2: characteristic-wise WENO reconstruction. 3: transmission-based WENO reconstruction. `Default = 0`

tfluct_solver

Whether a total fluctuation solver have to be used. If True the function that calculates the total fluctuation must be provided. `Default = False`

tfluct

Pointer to Fortran routine to calculate total fluctuation `Default = default_tfluct (None)`

aux_time_dep
Whether the auxiliary array is time dependent. `Default = False`

kernel_language
Specifies whether to use wrapped Fortran routines ('Fortran') or pure Python ('Python'). `Default = 'Fortran'`.

num_ghost
Number of ghost cells. `Default = 3`

fwave
Whether to split the flux jump (rather than the jump in Q) into waves; requires that the Riemann solver performs the splitting. `Default = False`

cfl_desired
Desired CFL number. `Default = 2.45`

cfl_max
Maximum CFL number. `Default = 2.50`

dq_src
Whether a source term is present. If it is present the function that computes its contribution must be provided. `Default = None`

call_before_step_each_stage
Whether to call the method *self.before_step* before each RK stage. `Default = False`

dq(*state*)
Evaluate dq/dt * (delta t)

dqdt(*state*)
Evaluate dq/dt. This routine is used for implicit time stepping.

get_cfl_max()
Set maximum CFL number for current step depending on time integrator

get_dt_new()
Set time-step for next step depending on time integrator

setup(*solution*)
Allocate RK stage arrays or previous step solutions and fortran routine work arrays.

step(*solution*)
Evolve q over one time step.

Take one step with a Runge-Kutta or multistep method as specified by *solver.time_integrator*.

Pyclaw Classic Clawpack Solvers

The pyclaw classic clawpack solvers are a collection of solvers that represent the functionality of the older versions of clawpack. It comes in two forms, a pure python version and a python wrapping of the fortran libraries. All of the solvers available provide the same basic interface and provide the same options as the old versions of clawpack. The superclass solvers are not meant to be used separately but there to provide common routines for all the Clawpack solvers. Please refer to each of the inherited classes for more info about the methods and attributes they provide each class. .. The inheritance structure is:

Example This is a simple example of how to instantiate and evolve a solution to a later time

```
>>> from clawpack import pyclaw
>>> solver = pyclaw.ClawSolver1D()                    # Instantiate a default, 1d solver
>>> solver.limiters = pyclaw.limiters.tvd.vanleer  # Use the van Leer limiter
```

```
>>> solver.dt = 0.0001                                    # Set the initial time step
>>> solver.max_steps = 500                                # Set the maximum number of time steps

>>> solver.evolve_to_time(solution,t_end)   # Evolve the solution to t_end
```

pyclaw.classic.solver

class clawpack.pyclaw.classic.solver.**ClawSolver**(*riemann_solver=None*, *claw_package=None*)

Generic classic Clawpack solver

All Clawpack solvers inherit from this base class.

mthlim
Limiter(s) to be used. Specified either as one value or a list. If one value, the specified limiter is used for all wave families. If a list, the specified values indicate which limiter to apply to each wave family. Take a look at pyclaw.limiters.tvd for an enumeration. `Default = limiters.tvd.minmod`

order
Order of the solver, either 1 for first order (i.e., Godunov's method) or 2 for second order (Lax-Wendroff-LeVeque). `Default = 2`

source_split
Which source splitting method to use: 1 for first order Godunov splitting and 2 for second order Strang splitting. `Default = 1`

fwave
Whether to split the flux jump (rather than the jump in Q) into waves; requires that the Riemann solver performs the splitting. `Default = False`

step_source
Handle for function that evaluates the source term. The required signature for this function is:

def step_source(solver,state,dt)

before_step
Function called before each time step is taken. The required signature for this function is:

def before_step(solver,solution)

kernel_language
Specifies whether to use wrapped Fortran routines ('Fortran') or pure Python ('Python'). `Default = 'Fortran'`.

verbosity
The level of detail of logged messages from the Fortran solver. `Default = 0`.

setup(*solution*)
Perform essential solver setup. This routine must be called before solver.step() may be called.

step(*solution*)
Evolve solution one time step

The elements of the algorithm for taking one step are:

1. The `before_step()` (page 212) function is called
2. A half step on the source term `step_source()` (page 212) if Strang splitting is being used (`source_split` (page 212) = 2)
3. A step on the homogeneous problem $q_t + f(q)_x = 0$ is taken

4. A second half step or a full step is taken on the source term `step_source()` (page 212) depending on whether Strang splitting was used (`source_split` (page 212) = 2) or Godunov splitting (`source_split` (page 212) = 1)

This routine is called from the method evolve_to_time defined in the pyclaw.solver.Solver superclass.

Input

- *solution* - (`Solution` (page 221)) solution to be evolved

Output

- (bool) - True if full step succeeded, False otherwise

step_hyperbolic(*solution*)

Take one homogeneous step on the solution.

This is a dummy routine and must be overridden.

Change to Custom BC Function Signatures

To allow better access to aux array data in the boundary condition functions both the *qbc* and *auxbc* arrays are now passed to the custom boundary condition functions. The new signature is

def my_custom_BC(state, dim, t, qbc, auxbc, num_ghost): ...

and should be adopted as soon as possible. The old signature

def my_custom_BC(state, dim, t, bc_array, num_ghost): ...

can still be used but a warning will be issued and the old signature will not be supported when version 6.0 is released. This addition is available in versions > 5.2.0.

Running in parallel

PyClaw can be run in parallel on your desktop or on large supercomputers using the PETSc library. Running your PyClaw script in parallel is usually very easy; it mainly consists of replacing:

```
from clawpack import pyclaw
```

with:

```
import clawpack.petclaw as pyclaw
```

Also, most of the provided scripts in *pyclaw/examples* are set up to run in parallel simply by passing the command-line option *use_petsc=True* (of course, you will need to launch them with *mpirun*.

Installing PETSc

First make sure you have a working install of PyClaw. For PyClaw installation instructions, see *installation*.

To run in parallel you'll need:

- PETSc (http://www.mcs.anl.gov/petsc/petsc-as/) a toolkit for parallel scientific computing. The current recommended version is 3.3 with latest patch.
- petsc4py (http://code.google.com/p/petsc4py/): Python bindings for PETSc. The current recommended version is 3.3.

Obtaining PETSc:

PETSc 3.3 can be obtained using three approaches. Here we use Mercurial (http://mercurial.selenic.com/) to get it. Look at http://www.mcs.anl.gov/petsc/petsc-as/download/index.html for more information.

Do:

```
$ cd path/to/the/dir/where/you/want/download/Petsc-3.3
$ hg clone https://bitbucket.org/petsc/petsc-3.3 petsc-3.3
$ hg clone https://bitbucket.org/petsc/buildsystem-3.3 petsc-3.3/config/BuildSystem
```

For sh, bash, or zsh shells add the following lines to your shell start-up file:

```
$ export PETSC_DIR=path/to/the/dir/where/you/downloaded/Petsc-3.3/petsc-3.3
$ export PETSC_ARCH=your/architecture
```

whereas for csh/tcsh shells add the following lines to your shell start-up file:

```
$ setenv PETSC_DIR path/to/the/dir/where/you/downloaded/Petsc-3.3/petsc-3.3
$ setenv PETSC_ARCH your/architecture
```

For more information about PETSC_DIR and PETSC_ARCH, i.e. the variables that control the configuration and build process of PETSc, please look at http://www.mcs.anl.gov/petsc/petsc-as/documentation/installation.html.

Then, if you want PETSc-3.3 configure for 32-bit use the following command:

```
$ ./config/configure.py --with-cc=gcc --with-cxx=g++ --with-python=1 --download-mpich=1 --with-shared
```

whereas, if you want PETSc-3.3 64-bit do:

```
$ ./config/configure.py --with-cc=gcc --with-cxx=g++ --with-python=1 --download-mpich=1 --with-shared
```

Note that one of the option is –download-mpich=1. This means that mpich is downloaded. If you do not need/want mpich, remove this option. Note that you need MPI when using PETSc. Therefore, if the option –download-mpich=1 is removed you should have MPI installed on your system or in your user account.

Once the configuration phase is completed, build PETSc libraries with

```
$ make PETSC_DIR=path/to/the/dir/where/you/have/Petsc-3.3/petsc-3.3 PETSC_ARCH=your/architecture all
```

Check if the libraries are working by running

```
$ make PETSC_DIR=path/to/the/dir/where/you/have/Petsc-3.3/petsc-3.3 PETSC_ARCH=your/architecture test
```

Obtaining petsc4py:

petsc4py is a python binding for PETSc. We recommend installing petsc4py 3.3 because it is compatible with PETSc 3.3 and 3.2. To install this binding correctly make sure that the PETSC_DIR and PETSC_ARCH are part of your shell start-up file.

Obtain petsc4py-3.3 with mercurial:

```
$ cd path/to/the/dir/where/you/want/download/petsc4py
$ hg clone https://petsc4py.googlecode.com/hg/petsc4py-3.3 -r 3.3
```

The prefered method for the petsc4py iinstallation is pip (http://pypi.python.org/pypi/pip)

```
$ cd petsc4py-3.3
$ pip install . --user
```

Indeed, pip removes the old petsc4py installation, downloads the new version of cython (http://cython.org/) (if needed) and installs petsc4py.

To check petsc4py-3.3 installation do:

```
$ cd petsc4py-3.3/test
$ python runtests.py
```

All the tests cases should pass, i.e. OK should be printed at the screen.

NOTE: To run a python code that uses petsc4py in parallel you will need to use mpiexec or mpirun commands. It is important to remember to use the mpiexec or mpirun executables that come with the MPI installation that was used for configuring PETSc installation. If you have used the option –download-mpich=1 while installing PETSc, then the correct mpiexec to use is the one in ${PETSC_DIR}/${PETSC_ARCH}/bin. You can set this mpiexec to be your default by adding this line to your sh, bash, or zsh shell start-up file:

```
$ export PATH="${PETSC_DIR}/${PETSC_ARCH}/bin:${PATH}"
```

or this line in case you are using csh or tcsh shells:

```
$ setenv PATH "${PETSC_DIR}/${PETSC_ARCH}/bin:${PATH}"
```

You can test that you are using the right mpiexec by running a demonstration script that can be found in $PYCLAW/demo as follows:

```
$ cd $PYCLAW
$ mpiexec -n 4 python demo/petsc_hello_world.py
```

and you should get an output that looks like follows:

```
Hello World! From process 3 out of 4 process(es).
Hello World! From process 1 out of 4 process(es).
Hello World! From process 0 out of 4 process(es).
Hello World! From process 2 out of 4 process(es).
```

NOTE: An alternative way to install petsc4py is simply using the python script setup.py inside petsc4py, i.e.

```
$ cd petsc4py-dev
$ python setup.py build
$ python setup.py install --user
```

Testing your installation

If you don't have it already, install nose

```
$ easy_install nose
```

Now simply execute

```
$ cd $PYCLAW
$ nosetests
```

If everything is set up correctly, this will run all the regression tests (which include pure python code and python/Fortran code) and inform you that the tests passed. If any fail, please post the output and details of your platform on the claw-users Google group (http://http://groups.google.com/group/claw-users).

Running and plotting an example

Next

```
$ cd $PYCLAW/examples/advection/1d/constant
$ python advection.py use_PETSc=True iplot=1
```

This will run the code and then place you in an interactive plotting shell. To view the simulation output frames in sequence, simply press 'enter' repeatedly. To exit the shell, type 'q'. For help, type '?' or see this Clawpack interactive python plotting help page (http://kingkong.amath.washington.edu/clawpack/users/plotting.html#interactive-plotting-with-iplotclaw).

Tips for making your application run correctly in parallel

Generally serial PyClaw code should "just work" in parallel, but if you are not reasonably careful it is certainly possible to write serial code that will fail in parallel.

Most importantly, use the appropriate grid attributes. In serial, both *grid.n* and *grid.ng* give you the dimensions of the grid (i.e., the number of cells in each dimension). In parallel, *grid.n* contains the size of the whole grid, while *grid.ng* contains just the size of the part that a given process deals with. You should typically use only *grid.ng* (you can also use *q.shape[1:]*, which is equal to *grid.ng*).

Similarly, *grid.lower* contains the lower bounds of the problem domain in the computational coordinates, whereas *grid.lowerg* contains the lower bounds of the part of the grid belonging to the current process. Typically you should use *grid.lowerg*.

Additionally, be aware that when a Grid object is instantiated in a parallel run, it is not instantiated as a parallel object. A typical code excerpt looks like

```
>>> import clawpack.petclaw as pyclaw
>>> from clawpack import pyclaw
>>> mx = 320; my = 80
>>> x = pyclaw.Dimension('x',0.0,2.0,mx)
>>> y = pyclaw.Dimension('y',0.0,0.5,my)
>>> grid = pyclaw.Domain([x,y])
```

At this point, *grid.ng* is identically equal to *grid.n*, rather than containing the size of the grid partition on the current process. Before using it, you should instantiate a State object

```
>>> num_eqn = 5
>>> num_aux=1
>>> state = pyclaw.State(grid,num_eqn,num_aux)
```

Now *state.grid.ng* contains appropriate information.

Passing options to PETSc

The built-in applications (see *Working with PyClaw's built-in examples* (page 117)) are set up to automatically pass command-line options starting with a dash ("-") to PETSc.

Contents

PyClaw output

PyClaw supports options to output more than just the solution q. It can provide:

- Output of derived quantities computed from q; for instance, pressure (not a conserved quantity) could be computed from density and energy.
- Output of scalar functionals, such as the total mass summed over the whole grid.
- Output of gauge values, which are time traces of the solution at a single point.

Derived quantities and functionals are written out at the same times that the solution q is written. While these could be computed in postprocessing, it is more efficient to compute them at run-time for large parallel runs.

Gauge output is written at every timestep. In order to get this data without a gauge, one would otherwise have to write the full solution out at every timestep, which might be very slow.

Outputting derived quantities

It is sometimes desirable to output quantities other than those in the vector q. To do so, just add a function *p_function* to the controller that accepts the state and sets the derived quantities in state.p

```
>>> def stress(state):
...     state.p[0,:,:] = np.exp(state.q[0,:,:]*state.aux[1,:,:]) - 1.

>>> state.mp = 1
>>> claw.p_function = stress
```

Outputting functionals

In PyClaw a functional is a scalar quantity computed from q that is written to file at each output time. For now, only functionals of the form

$$F(q) = \int |f(q)| dx dy \quad (4.-13)$$

are supported. In other words, the functional must be the absolute integral of some function of q. To enable writing functionals, simply set state.mf to the number of functionals and point the controller to a function that computes $f(q)$

```
>>> def compute_f(state):
...     state.F[0,:,:] = state.q[0,:,:]*state.q[1,:,:]

>>> state.mf = 1
```

Using gauges

A gauge in PyClaw is a single grid location for which output is written at every time step. This can be very useful in some applications, like comparing with data from tidal gauges (from whence the name is derived) in tsunami modeling. The gauges are managed by the grid object, and a grid at location (x, y) may be added simply by calling *grid.add_gauges((x,y))*. Multiple gauges can be set at once by providing a list of coordinate tuples

```
>>> state.grid.add_gauges([(x1,y1),(x2,y2),(x3,y3)])
```

By default, the solution values are written out at each gauge location. To write some other quantity, simply provide a function $f(q, aux)$ and point the solver to it

```
>>> def f(q,aux):
...     return q[1,:,:]/q[0,:,:]

>>> solver.compute_gauge_values = f
```

Logging

By default, PyClaw prints a message to the screen each time it writes an output file. This message is also writing to the file *pyclaw.log* in the working directory. There are additional warning or error messages that may be sent to the screen or to file. You can adjust the logger levels in order to turn these messages off or to get more detailed debugging information.

The controller provides one means to managing the logging with the `verbosity` parameter and is provided as an easy interace to control the console output (that which is shown on screen). Valid values for `verbosity` are:

Malformed table. Column span alignment problem in table line 5.

```
===========  ================
Verbosity     Message Level
-----------  ----------------
0             Critical - This effectively silences the logger, since there are
              no logging messages in PyClaw that correspond to this level.  May
              be useful in an IPython notebook for instance if you want the
              plots to appear immediately below your code.
1             Error - These are logged by the IO system to indicate that
              something has gone wrong with either reading or writing a file.
2             Warning - There are no warning level logger messages.
3             Info - Additional IO messages are printed and some minor messages
              dealing with hitting the end time requested.
4             Debug - If this level is set all logger output is displayed.  This
              includes the above and detailed time step information for every
              time step (includes CFL, current dt and whether a time step is
              rejected).
===========  ================
```

When running on a supercomputer, logging to file can be problematic because the associated I/O can slow down the entire computation (this is true on Shaheen). To turn off all logging (both to screen and to file), you need to change the level of the root logger:

```
import logging
logger = logging.getLogger('pyclaw')
logger.setLevel(logging.CRITICAL)
```

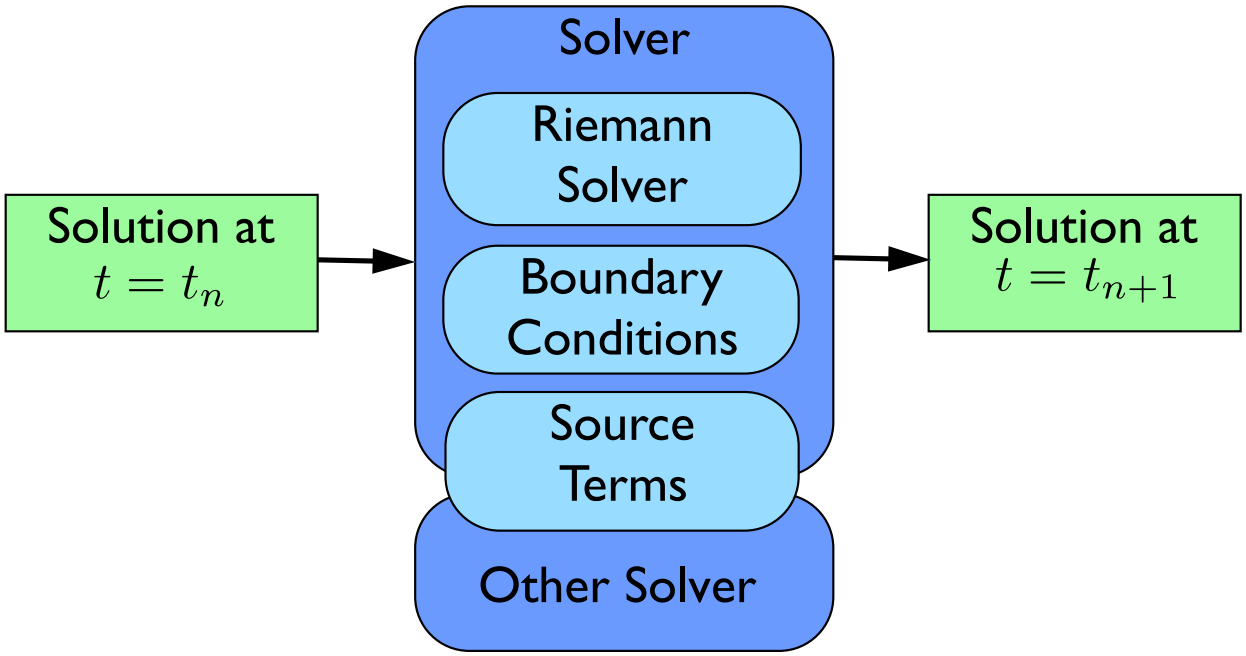
Again since we don't use *CRITICAL* logger messages in PyClaw, this has the effect of turning the loggers off.

4.2.3 Understanding Pyclaw Classes

Contents

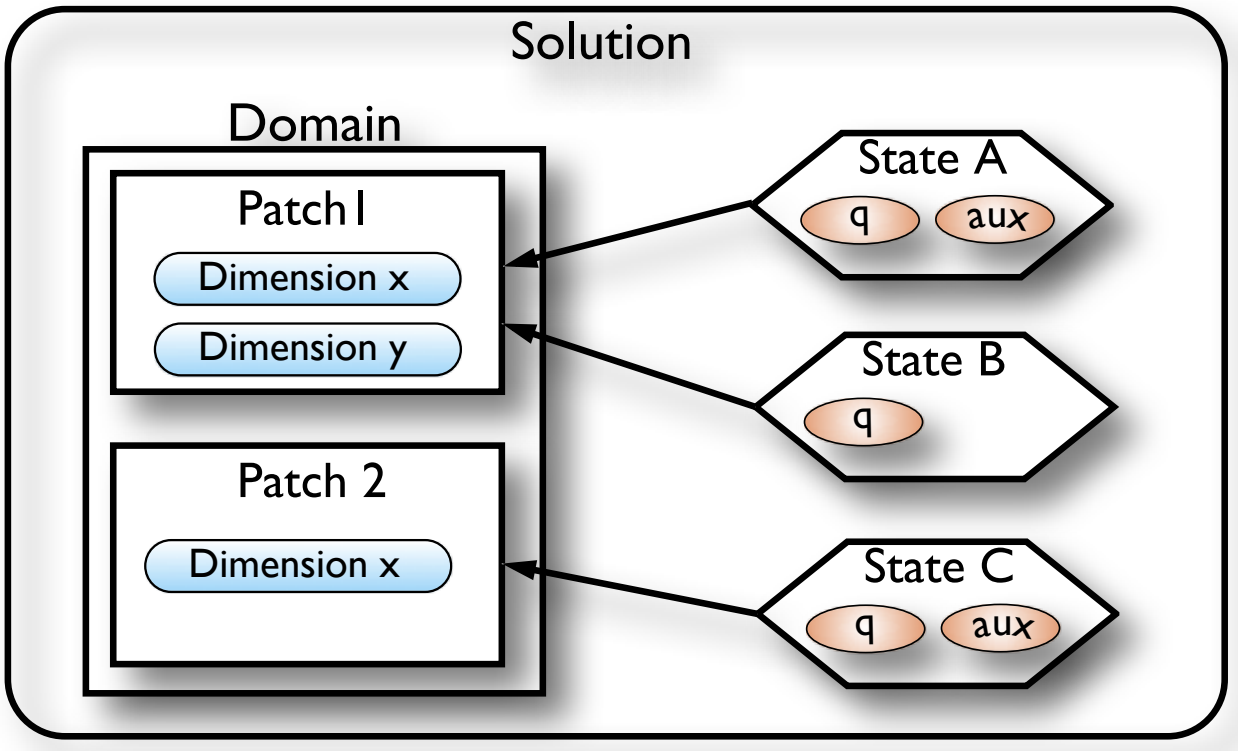
Flow of a Pyclaw Simulation

The basic idea of a pyclaw simulation is to construct a `Solution` (page 221) object, hand it to a `Solver` object, and request a solution at a new time. The solver will take whatever steps are necessary to evolve the solution to the requested time.



The bulk of the work in order to run a simulation then is the creation and setup of the appropriate `Domain` (page 230), `State` (page 224), `Solution` (page 221), and `Solver` objects needed to evolve the solution to the requested time.

Creation of a Pyclaw `Solution`



A Pyclaw `Solution` (page 221) is a container for a collection of `Domain` (page 230) and `State` (page 224) designed with a view to future support of adaptive mesh refinement and multi-block simulations. The `Solution` (page 221) object keeps track of a list of `State` (page 224) objects and controls the overall input and output of the entire collection of `State` (page 224) objects. Each `State` (page 224) object inhabits a `Grid` (page 231), composed of `Dimension` (page 234) objects that define the extents of the `Domain`. Multiple states can inhabit the same grid, but each `State` (page 224) inhabits a single grid.

The process needed to create a `Solution` (page 221) object then follows from the bottom up.

```
>>> from clawpack import pyclaw
>>> x = pyclaw.Dimension('x', -1.0, 1.0, 200)
>>> y = pyclaw.Dimension('y', 0.0, 1.0, 100)
```

This code creates two dimensions, a dimension `x` on the interval `[-1.0, 1.0]` with 200 grid points and a dimension `y` on the interval `[0.0, 1.0]` with 100 grid points.

Note: Many of the attributes of a `Dimension` (page 234) object are set automatically so make sure that the values you want are set by default. Please refer to the `Dimension` (page 234) classes definition for what the default values are.

Next we have to create a `Domain` (page 230) object that will contain our `dimensions` objects.

```
>>> domain = pyclaw.Domain([x,y])
>>> num_eqn = 2
>>> state = pyclaw.State(domain,num_eqn)
```

Here we create a `domain` with the dimensions we created earlier to make a single 2D `Domain` (page 230) object. Then we set the number of equations the State will represent to 2. Finally, we create a `State` (page 224) that inhabits this domain. As before, many of the attributes of the `Domain` (page 230) and State objects are set automatically.

We now need to set the initial condition q and possibly aux to the correct values.

```
>>> import numpy as np
>>> sigma = 0.2
>>> omega = np.pi
>>> Y,X = np.meshgrid(state.grid.y.centers,state.grid.x.centers)
>>> r = np.sqrt(X**2 + Y**2)
>>> state.q[0,:] = np.cos(omega * r)
>>> state.q[1,:] = np.exp(-r**2 / sigma**2)
```

We now have initialized the first entry of q to a cosine function evaluated at the cell centers and the second entry of q to a gaussian, again evaluated at the grid cell centers.

Many Riemann solvers also require information about the problem we are going to run which happen to be grid properties such as the impedence Z and speed of sound c for linear acoustics. We can set these values in the problem_data dictionary in one of two ways. The first way is to set them directly as in:

```
>>> state.problem_data['c'] = 1.0
>>> state.problem_data['Z'] = 0.25
```

If you're using a Fortran Riemann solver, these values will automatically get copied to the corresponding variables in the cparam common block of the Riemann solver. This is done in solver.setup(), which calls state.set_cparam().

Last we have to put our State (page 224) object into a Solution (page 221) object to complete the process. In this case, since we are not using adaptive mesh refinement or a multi-block algorithm, we do not have multiple grids.

```
>>> sol = pyclaw.Solution(state,domain)
```

We now have a solution ready to be evolved in a Solver object.

Creation of a Pyclaw Solver

A Pyclaw Solver can represent many different types of solvers; here we will use a 1D, classic Clawpack type of solver. This solver is defined in the solver module.

First we import the particular solver we want and create it with the default configuration.

```
>>> solver = pyclaw.ClawSolver1D()
>>> solver.bc_lower[0] = pyclaw.BC.periodic
>>> solver.bc_upper[0] = pyclaw.BC.periodic
```

Next we need to tell the solver which Riemann solver to use from the *Riemann Solver Package* (page 240). We can always check what Riemann solvers are available to use via the riemann module. Once we have picked one out, we pass it to the solver via:

```
>>> from clawpack import riemann
>>> solver.rp = riemann.acoustics_1D
```

In this case we have decided to use the 1D linear acoustics Riemann solver. You can also set your own solver by importing the module that contains it and setting it directly to the *rp* attribute of the particular object in the class ClawSolver1D.

```
>>> import my_rp_module
>>> solver.rp = my_rp_module.my_acoustics_rp
```

Last we finish up by specifying solver options, if we want to override the defaults. For instance, we might want to specify a particular limiter

```
>>> solver.limiters = pyclaw.limiters.tvd.vanleer
```

If we wanted to control the simulation we could at this point by issuing the following commands:

```
>>> solver.evolve_to_time(sol,1.0)
```

This would evolve our solution `sol` to `t = 1.0` but we are then responsible for all output and other setup considerations.

Creating and Running a Simulation with `Controller`

The `Controller` (page 206) coordinates the output and setup of a run with the same parameters as the classic Clawpack. In order to have it control a run, we need only to create the controller, assign it a solver and initial condition, and call the `run()` (page 207) method.

```
>>> from pyclaw.controller import Controller

>>> claw = Controller()
>>> claw.solver = solver
>>> claw.solutions = sol
```

Here we have imported and created the `Controller` (page 206) class, assigned the `Solver` and `Solution` (page 221).

These next commands setup the type of output the controller will output. The parameters are similar to the ones found in the classic clawpack claw.data format.

```
>>> claw.output_style = 1
>>> claw.num_output_times = 10
>>> claw.tfinal = 1.0
```

When we are ready to run the simulation, we can call the `run()` (page 207) method. It will then run the simulation and output the appropriate time points. If the `keep_copy` (page 207) is set to *True* the controller will keep a copy of each solution output in memory in the frames array. For instance, you can then immediately plot the solutions output into the *frames* array.

Restarting a simulation

To restart a simulation, simply initialize a Solution object using an output frame from a previous run; for example, to restart from frame 3

```
>>> claw.solution = pyclaw.Solution(3)
```

By default, the `Controller` (page 206) will number your output frames starting from the frame number used for initializing the `Solution` (page 221) object. If you want to change the default behaviour and start counting frames from zero, you will need to pass the keyword argument `count_from_zero=True` to the solution initializer.

Note: It is necessary to specify the output format ('petsc' or 'ascii').

If your simulation includes aux variables, you will need to either recompute them or output the aux values at every step, following the instructions below.

Outputting aux values

To write aux values to disk at the initial time:

```
>>> claw.write_aux_init = True
```

To write aux values at every step:

```
>>> claw.write_aux_always = True
```

Outputting derived quantities

It is sometimes desirable to output quantities other than those in the vector q. To do so, just add a function *compute_p* to the controller that accepts the state and sets the derived quantities in state.p

```
>>> def stress(state):
...     state.p[0,:,:] = np.exp(state.q[0,:,:]*state.aux[1,:,:]) - 1.

>>> state.mp = 1
>>> claw.compute_p = stress
```

4.2.4 Developers' Guide

Contents

Guidelines for contributing

When preparing contributions, please follow the guidelines in *contribution*. Also:

- If the planned changes are substantial or will be backward-incompatible, it's best to discuss them on the claw-dev Google group (http://groups.google.com/group/claw-dev) before starting.
- Make sure all tests pass and all the built-in examples run correctly.
- Be verbose and detailed in your commit messages and your pull request.
- It may be wise to have one of the maintainers look at your changes before they are complete (especially if the changes will necessitate modifications of tests and/or examples).
- If your changes are not backward-compatible, your pull request should include instructions for users to update their own application codes.

Reporting and fixing bugs

If you find a bug, post an issue with as much explanation as possible on the appropriate issue tracker (for instance, the PyClaw issue tracker is at https://github.com/clawpack/pyclaw/issues. If you're looking for something useful to do, try tackling one of the issues listed there.

Developer communication

Developer communication takes place on the google group at http://groups.google.com/group/claw-dev/, and (increasingly) within the issue trackers on Github.

Installation instructions for developers

Cloning the most recent code from Github

You can create a read-only development version of Clawpack via:

```
git clone git://github.com/clawpack/clawpack.git
cd clawpack
python setup.py git-dev
```

This downloads the following clawpack modules as subrepositories checked out at specific commits (as opposed to the tip of a branch).

- https://github.com/clawpack/pyclaw (Python code, some of which is needed also for Fortran version)
- https://github.com/clawpack/clawutil (Utility functions, Makefile.common used in multiple repositories)
- https://github.com/clawpack/classic (Classic single-grid code)
- https://github.com/clawpack/amrclaw (AMR version of Fortran code)
- https://github.com/clawpack/riemann (Riemann solvers)
- https://github.com/clawpack/visclaw (Python graphics and visualization tools)
- https://github.com/clawpack/geoclaw (GeoClaw)

This should give a snapshot of the repositories that work well together. (Note that there are many inter-dependencies between code in the repositories and checking out a different commit in one repository may break things in a different repository.)

If you want to also install the PyClaw Python components, you can then do:

```
python setup.py install
```

If you plan to work on the Python parts of Clawpack as a developer, you may instead wish to do:

```
pip install -e .
```

The advantage of this is that when you edit Python code in your clawpack directly, it will immediately take effect, without the need to install again. However, the (potential) danger of this approach is that the path to your clawpack directory will be stored in the file site-packages/easy-install.pth and prepended to your PYTHONPATH whenever you run Python. This path will take precedence over any manually added paths, unless you delete the .pth file.

If you want to use the Fortran versions in *classic*, *amrclaw*, *geoclaw*, etc., you need to set environment variables and proceed as described at *Set environment variables* (page 9).

Updating to the latest development version

The repositories will each be checked out to a specific commit and will probably be in a detached-head state. You will need to checkout *master* in each repository to see the current head of the master branch.

You should never commit to *master*, only to a feature branch, so the *master* branch should always reflect what's in the main *clawpack* repository. You can update it to reflect any changes via:

```
git checkout master
git fetch origin
git merge origin/master
```

or simply:

```
git pull origin master:master
```

Remember that you need to do this in each repository before running anything to make sure everything is up to date with *master*.

Adding your fork as a remote

If you plan to make changes and issue pull requests to one or more repositories, you will need to do the following steps for each such repository:

1. Go to http://github.com/clawpack and fork the repository to your own Github account. (Click on the repository name and then the *Fork* button at the top of the screen.)
2. Add a *remote* pointing to your repository. For example, if you have forked the *amrclaw* repository to account *username*, you would do:

   ```
   cd amrclaw
   git remote add username git@github.com:username/amrclaw.git
   ```

   You should push only to this remote, not to *origin*, e.g.:

   ```
   git push username
   ```

You might also want to clone some or all of the following repositories:

- https://github.com/clawpack/doc (documentation)
- https://github.com/clawpack/apps (To collect applications)
- https://github.com/clawpack/regression (Regression tests)
- https://github.com/clawpack/clawpack-4.x (Previous versions, 4.6)

These are not brought over by cloning the top *clawpack* super-repository. You can get one of these in read-only mode by doing, e.g.:

```
git clone git://github.com/clawpack/doc.git
```

Then go through the above steps to add your own fork as a remote if you plan to modify code and issue pull requests.

Modifying code

Before making changes, make sure *master* is up to date:

```
git checkout master
git pull
```

Then create a new branch based on *master* for any new commits:

```
git checkout -b new_feature master
```

Now make changes, add and commit them, and then push to your own fork:

```
# make some changes
# git add the modified files
git commit -m "describe the changes"

git push username new_feature
```

If you want these changes pulled into *master*, you can issue a pull request from the github page for your fork of this repository (make sure to select the correct branch of your repository).

Note: If you accidentally commit to *master* rather than creating a feature branch first, you can easily recover:

```
git checkout -b new_feature
```

will create a new branch based on the current state and history (including your commits to *master*) and you can just continue adding additional commits.

The only problem is your *master* branch no longer agrees with the history on Github and you want to throw away the commits you made to *master*. The easiest way to do this is just to make sure you're on a different branch, e.g.,

```
git checkout new_feature
```

and then:

```
git branch -D master
git checkout -b master origin/master
```

This deletes your local branch named *master* and recreates a branch with the same name based on *origin/master*, which is what you want.

Issuing a pull request

Before issuing a pull request, you should make sure you have not broken anything:

1. Make sure you are up to date with *master*:

   ```
   git checkout master
   git pull
   ```

   If this does not say "Already up-to-date" then you might want to rebase your modified code onto the updated master. With your feature branch checked out, you can see what newer commits have been added to *master* via:

   ```
   git checkout new_feature
   git log HEAD..master
   ```

   If your new feature can be added on to the updated master, you can rebase:

   ```
   git rebase master
   ```

   which gives a cleaner history than merging the branches.

2. Run the appropriate regression tests. If you have modified code in pyclaw or riemann, then you should run the pyclaw tests. First, if you have modified any Fortran code, you need to recompile:

   ```
   cd clawpack/
   pip install -e .
   ```

   Then run the tests:

   ```
   cd pyclaw
   nosetests
   ```

   If any tests fail, you should fix them before issuing a pull request.

To issue a pull request (PR), go to the Github page for your fork of the repository in question, select the branch from which you want the pull request to originate, and then click the *Pull Request* button.

Testing a pull request

To test out someone else's pull request, follow these instructions: For example, if you want to try out a pull request coming from a branch named *bug-fix* from user *rjleveque* to the *master* branch of the *amrclaw* repository, you would do:

```
cd $CLAW/amrclaw   # (and make sure you don't have uncommitted changes)
git checkout master
git pull  # to make sure you are up to date

git checkout -b rjleveque-bug-fix master
git pull https://github.com/rjleveque/amrclaw.git bug-fix
```

This puts you on a new branch of your own repository named *rjleveque-bug-fix* that has the proposed changes pulled into it.

Once you are done testing, you can get rid of this branch via:

```
git checkout master
git branch -D rjleveque-bug-fix
```

Top-level pull requests

The top level *clawpack* repository keeps track of what versions of the subrepositories work well together.

If you make pull requests in two different repositories that are linked, say to both *pyclaw* and *riemann*, then you should also push these changes to the top-level *clawpack* repository and issue a PR for this change:

```
cd $CLAW   # top-level clawpack repository
git checkout master
git pull
git checkout -b pyclaw-riemann-changes
git add pyclaw riemann
git commit -m "Cross-update pyclaw and riemann."
git push username pyclaw-riemann-changes
```

Git workflow

See *git-resources* for useful links.

Catching errors with Pyflakes and Pylint

Pyflakes and Pylint are Python packages designed to help you catch errors or poor coding practices. To run pylint on the whole PyClaw package, do:

```
cd $PYCLAW
pylint -d C pyclaw
```

The *-d* option suppresses a lot of style warnings, since PyClaw doesn't generally conform to PEP8. To run pylint on just one module, use something like:

```
pylint -d C pyclaw.state
```

Since pylint output can be long, it's helpful to write it to an html file and open that in a web browser:

```
pylint -d C pyclaw.state -f html > pylint.html
```

Pyflakes is similar to pylint but aims only to catch errors. If you use Vim, there is a nice extension package pyflakes.vim (https://github.com/kevinw/pyflakes-vim) that will catch errors as you code and underline them in red.

Checking test coverage

You can use nose to see how much of the code is covered by the current suite of tests and track progress if you add more tests

```
nosetests --with-coverage --cover-package=pyclaw --cover-html
```

This creates a set of html files in *./cover*, showing exactly which lines of code have been tested.

4.2.5 Troubleshooting

This page lists some of the most common difficulties encountered in installing and running PyClaw. If you do not find a solution for your problem here, please e-mail the claw-users Google group (http://http://groups.google.com/group/claw-users). You may also wish to consult the list of known issues (https://github.com/clawpack/pyclaw/issues).

Compilation errors

Two frequent sources of compilation error are:

- Your environment variable FC is set to g77 or another Fortran 77 compiler. FC should be undefined or set to a Fortran 90 compiler. If you have installed gfortran, you could set:

  ```
  $ export FC = gfortran
  ```

  in your .bash_profile (in mac) or .bashrc (in linux).

- Conflicts between 32-bit and 64-bit files. This has been encountered on Mac OS X with 32-bit Enthought Python. We recommend using a 64-bit Python install, such as that available from Enthought (free for academics). The 32-bit EPD has also been known to cause a plotting issue with PyClaw in which plotting becomes extremely slow.

Use Fortran-ordered arrays

By default, Numpy arrays use C-ordering. But the arrays that store the solution and coefficients in PyClaw (i.e., q and aux) must be initialized using Fortran ordering, for compatibility with the Fortran routines and PETSc. Ordinarily, this is handled automatically when you create a State or Solution object. If you are manually creating arrays, be sure to pass the flag 'F' to specify Fortran ordering.

Installation

When installing Clawpack, if you get an error message saying that lblas or llapack is not found, please update your installation of Numpy to at least version 1.8. You can do this via:

```
pip install -U numpy
```

Then try the installation again.

4.2.6 About PyClaw

PyClaw is part of Clawpack – an open-source, free project. If you find PyClaw useful, please let us know (claw-users@googlegroups.com).

Contributors

Many people have contributed to PyClaw, some of them very substantial parts of the package. Their work is greatly appreciated: no open source project can survive without a community. The following people contributed major parts of the library (in alphabetical order)

- Aron Ahmadia: PETSc integration; I/O; programmatic testing framework; general design.
- Amal Alghamdi: Initial development of PetClaw, many bug-fixes and enhancements.
- Jed Brown: Implicit time stepping (still experimental).
- Ondrej Certik: Installation and continuous integration bug-fixes.
- Lisandro Dalcin: Fortran wrapping; PETSc integration; general efficiency.
- Matthew Emmett: PyWENO integration.
- Yiannis Hadjimichael: Documentation testing.
- David Ketcheson: General maintenance and development; incorporation of SharpClaw routines.
- Matthew Knepley: General design; PETSc integration.
- Grady Lemoine: Interleaving and cache-optimization of 3D Classic routines.
- Kyle Mandli: Initial design and implementation of the PyClaw framework.
- Matteo Parsani: Mapped grids; Python-Fortran interfacing; implicit time stepping.
- Manuel Quezada de Luna: 2D P-system Riemann solvers and example script.
- Kristof Unterweger: PeanoClaw (AMR); still experimental.

Contributions to the package are most welcome. If you have used PyClaw for research, chances are that others would find your code useful. See *develop* for more details.

License

PyClaw is distributed under a Berkeley Software Distribution (BSD) style license. The license is in the file pyclaw/LICENSE.txt and reprinted below.

See http://www.opensource.org/licenses/bsd-license.php for more details.

Copyright (c) 2008-2011 Kyle Mandli and David I. Ketcheson. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of King Abdullah University of Science & Technology nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Funding

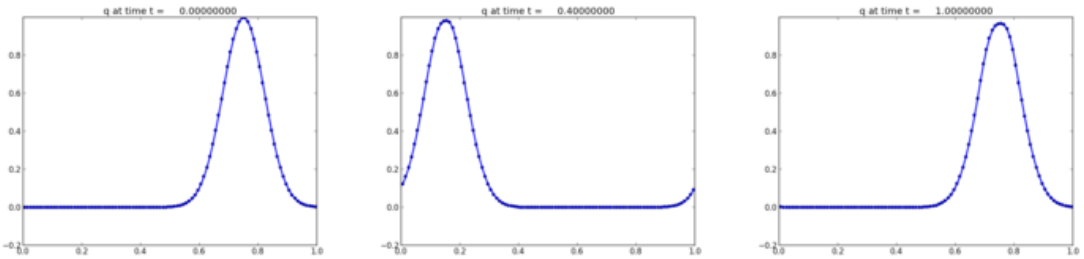
PyClaw development has been supported by grants from King Abdullah University of Science & Technology.

4.2.7 Gallery of all PyClaw applications

Contents

1-dimensional advection

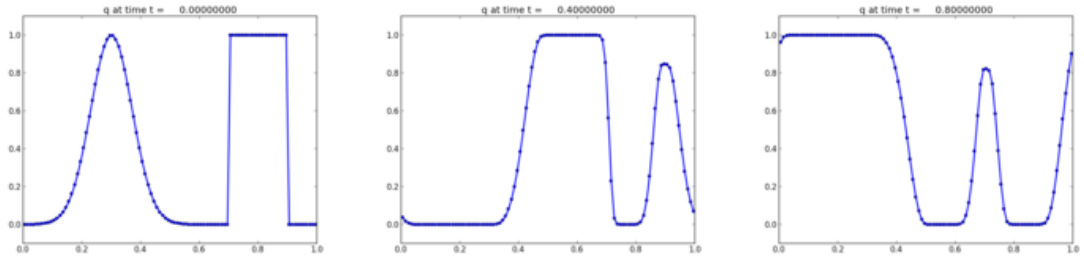
Advecting Gaussian with periodic boundary.



Source code ... Plots

1-dimensional variable-velocity advection

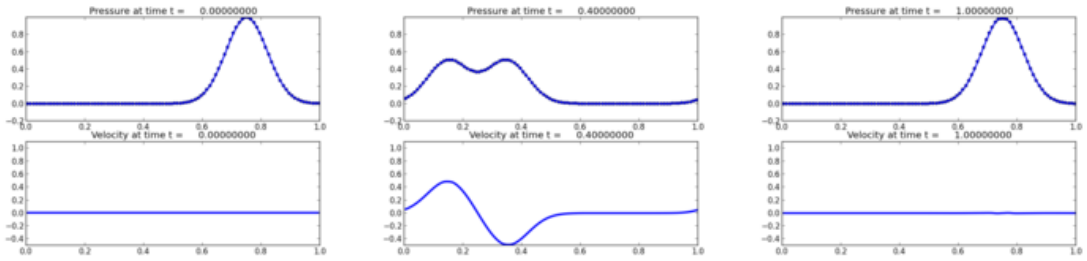
Advecting Gaussian and square wave with periodic boundary.



Source code ... Plots

1-dimensional acoustics

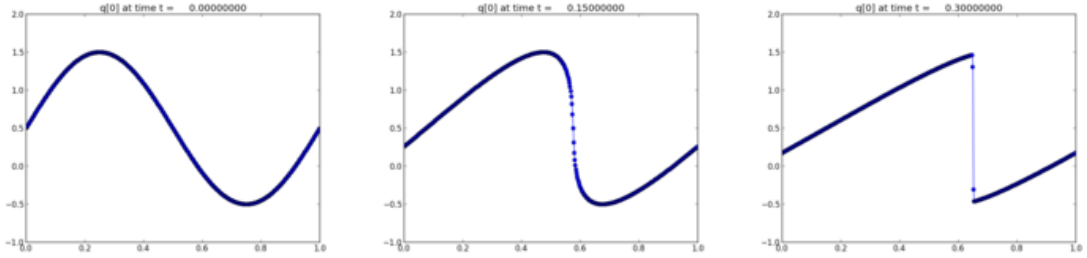
Acoustics equations with wall boundary at left and extrap at right.



Source code ... Plots

1-dimensional Burgers' equation

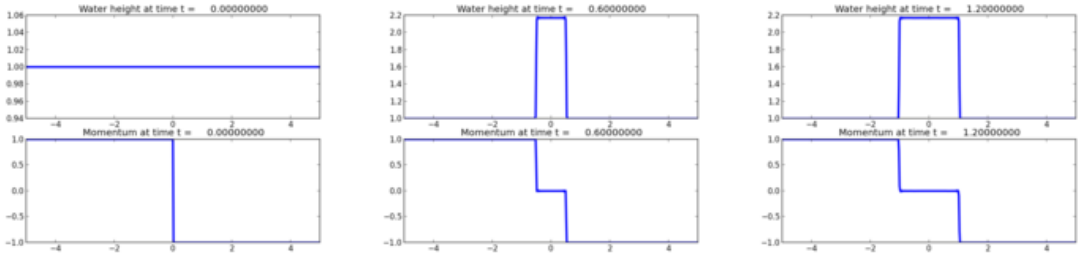
Burgers' equation with sinusoidal initial data, steepening to N-wave.



Source code ... Plots

1-dimensional shallow water equation

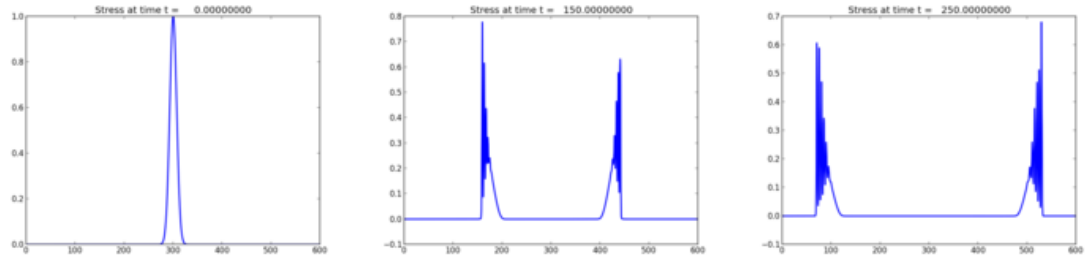
Shallow water shock tube.



Source code ... Plots

1-dimensional nonlinear elasticity

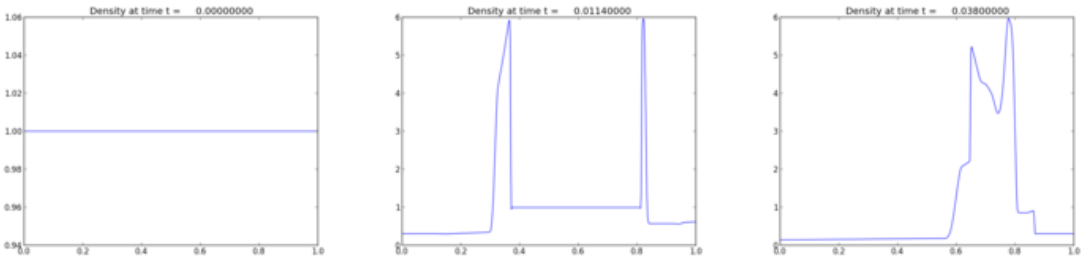
Evolution of two trains of solitary waves from an initial gaussian.



Source code ... Plots

1-dimensional Euler equations

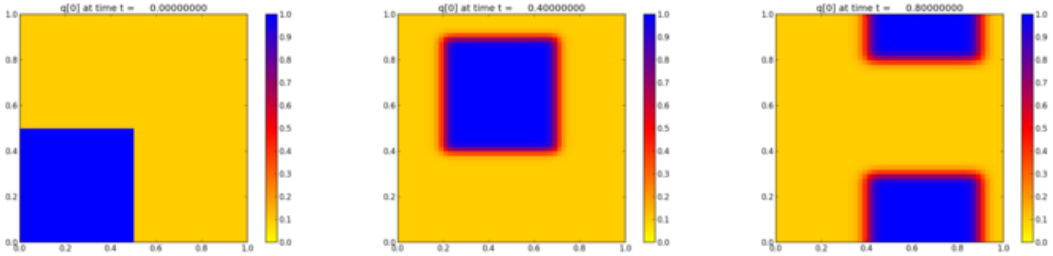
Woodward-Colella blast-wave interaction problem.



Source code ... Plots

2-dimensional advection

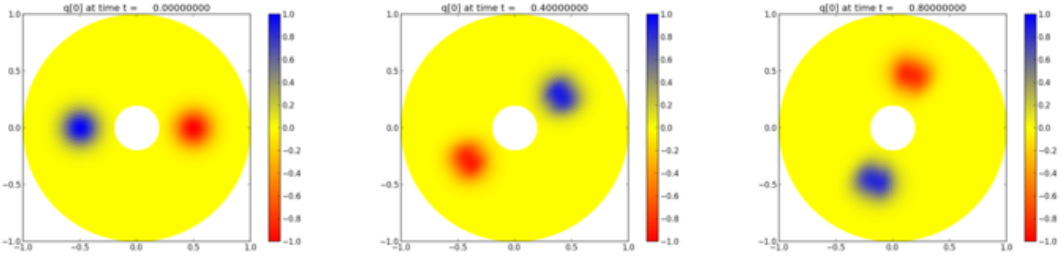
Advecting square with periodic boundary conditions.



Source code ... Plots

2-dimensional variable-coefficient advection

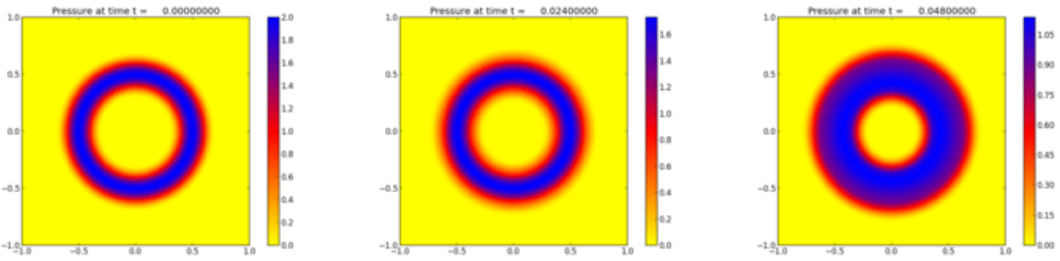
Advection in an annular region.



Source code ... Plots

2-dimensional acoustics

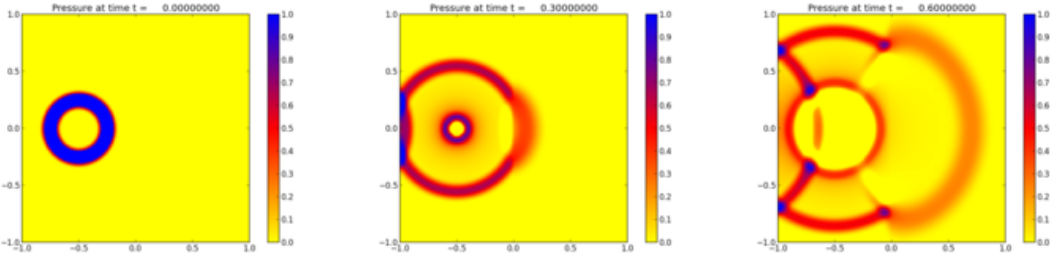
Expanding radial acoustic wave in a homogeneous medium.



Source code ... Plots

2-dimensional variable-coefficient acoustics

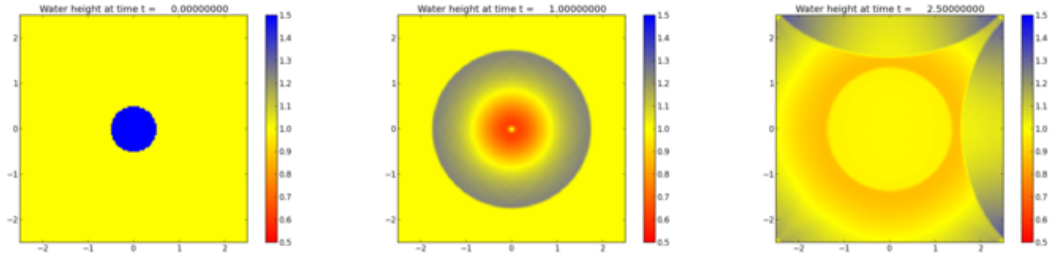
Expanding radial acoustic wave in a two-material medium with an interface.



Source code ... Plots

2-dimensional shallow water equations

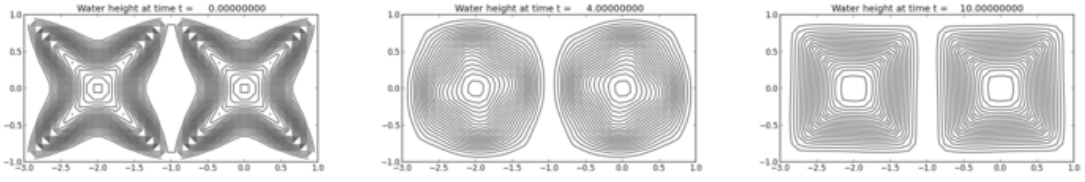
Radial dam-break.



Source code ... Plots

2-dimensional shallow water on the sphere

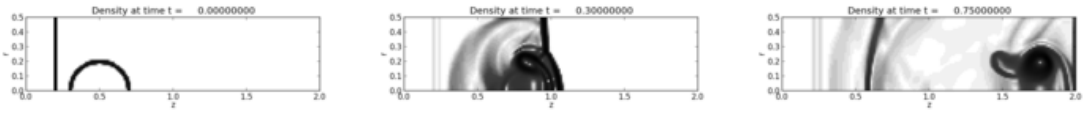
Wavenumber 4 Rossby-Haurwitz wave on a rotating sphere.



Source code ... Plots

2-dimensional Euler equations

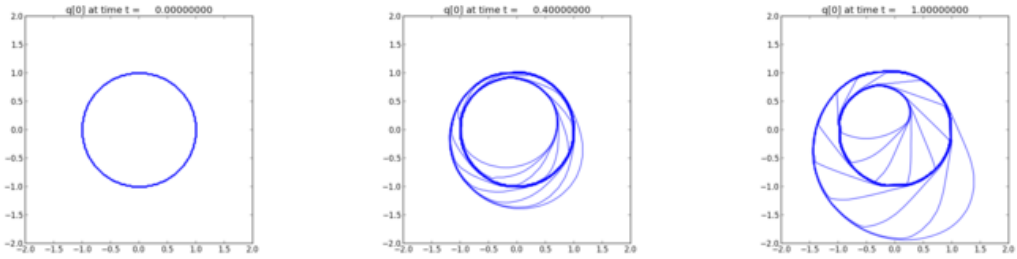
Shock-bubble interaction.



Source code ... Plots

2-dimensional KPP equation

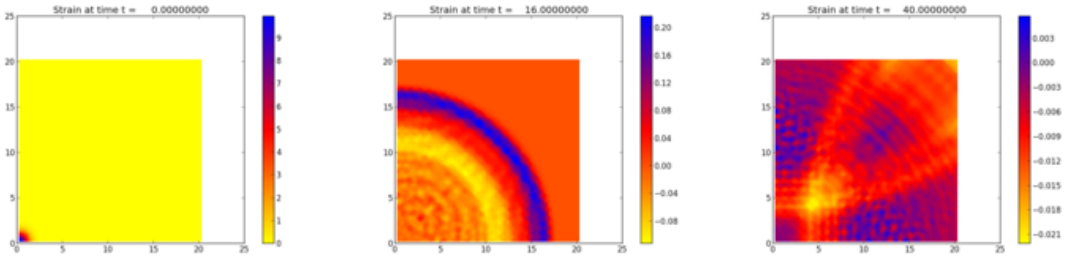
Non-convex flux example.



Source code ... Plots

2-dimensional p-system

Radial wave in a checkerboard-like medium.



Source code ... Plots

4.2.8 Building the PyClaw gallery locally

You can build a local copy of the PyClaw gallery as follows: first, you should clone the clawpack documentation repository:

```
git clone git://github.com/clawpack/doc
```

Then run all the examples:

```
cd doc/doc/pyclaw/gallery
python make_plots.py
```

Next generate the gallery itself:

```
python gallery.py
```

Finally, you need to call sphinx to convert all the .rst files to .html:

```
cd ../..
make html
```

4.3 PyClaw Modules reference documentation

4.3.1 Pyclaw Controller Class

The pyclaw controller object is a convenience class for running simulations based on the classic clawpack formats and output specifications. It allows for a variety of output time specifications, output styles and other ways to keep a simulation organized.

The main way to use a Controller object then is to provide it with an appropriate `Solver` and initial `Solution` (page 221) object. Then specify what kind of output you would like different than the defaults (see `Controller` (page 206) for details on what those are). Then simply call `run()` (page 207) in order to run the desired simulation.

```
>>> import pyclaw.controller as controller
>>> claw = controller.Controller()             # Instantiate a new controller
>>> claw.solver = my_solver                    # Assign a solver
>>> claw.solution = my_initial_solution        # Assign an initial condition
```

Here we would set a variety of run parameters such as `tfinal`, `keep_copy` if we wanted to plot the solutions immediately, or `output_format` to specify a format other than `ascii` or no output files if we are going to use `keep_copy = True`. After we are all set up we just need to call the controller's `run()` method and off we go.

```
>>> claw.run()
```

Please see the *PyClaw tutorial: Solve the acoustics equations* (page 115) for a detailed example of how this would work in its entirety.

pyclaw.controller.Controller

class pyclaw.controller.**Controller**

Controller for pyclaw simulation runs and plotting

Initialization Input: None

Examples

```
>>> import clawpack.pyclaw as pyclaw
>>> x = pyclaw.Dimension('x',0.,1.,100)
>>> domain = pyclaw.Domain((x))
>>> state = pyclaw.State(domain,3,2)
>>> claw = pyclaw.Controller()
>>> claw.solution = pyclaw.Solution(state,domain)
>>> claw.solver = pyclaw.ClawSolver1D()
```

check_validity()

Check that the controller has been properly set up and is ready to run.

Also checks validity of the solver, solution and states.

plot()
Plot from memory.

run()
Convenience routine that will evolve solution based on the traditional clawpack output and run parameters.

This function uses the run parameters and solver parameters to evolve the solution to the end time specified in run_data, outputting at the appropriate times.

Input None

Ouput (dict) - Return a dictionary of the status of the solver.

F_file_name = None
(string) - Name of text file containing functionals

F_path
(string) - Full path to output file for functionals

compute_F = None
(function) - Function that computes density of functional F

compute_p = None
(function) - function that computes derived quantities

file_prefix_p = None
(string) - File prefix to be prepended to derived quantity output files

frames = None
(list) - List of saved frames if `keep_copy` is set to `True`

keep_copy = None
(bool) - Keep a copy in memory of every output time, `default = False`

nstepout = None
(int) - Number of steps between output, only used with `output_style = 3`, `default = 1`

num_output_times = None
(int) - Number of output times, only used with `output_style = 1`, `default = 10`

out_times = None
(int) - Output time list, only used with `output_style = 2`, `default = numpy.linspace(0.0,tfinal,num_output_times)`

outdir = None
(string) - Output directory, directs output files to outdir

outdir_p
(string) - Directory to use for writing derived quantity files

output_file_prefix = None
(string) - File prefix to be appended to output files, `default = None`

output_format = None
(list of strings) - Format or list of formats to output the data, if this is None, no output is performed. See _pyclaw_io for more info on available formats. `default = 'ascii'`

output_options = None
(dict) - Output options passed to function writing and reading data in output_format's format. `default = {}`

output_style = None
(int) - Time output style, `default = 1`

overwrite = None
(bool) - Ok to overwrite old result in outdir, `default = True`

plotdata = None
(`ClawPlotData`) - An instance of a `ClawPlotData` object defining the objects plot parameters.

rundir = None
(string) - Directory to run from (containing *.data files), uses *.data from rundir

runmake = None
(bool) - Run make in xdir before xclawcmd

savecode = None
(bool) - Save a copy of *.f files in outdir

solver = None
(`Solver`) - Solver object

tfinal = None
(float) - Final time output, `default = 1.0`

verbosity
(int) - Level of output to screen; `default = 3`

viewable_attributes = None
(list) - Viewable attributes of the *:class:'~pyclaw.controller.Controller*

write_aux_always = None
(bool) - Write out auxiliary array at every time step, `default = False`

write_aux_init = None
(bool) - Write out initial auxiliary array, `default = False`

xclawcmd = None
(string) - Command to execute (if using fortran), defaults to xclaw or xclaw.exe if cygwin is being used (which it checks vis sys.platform)

xclawerr = None
(string) - Where to write error messages

xclawout = None
(string) - Where to write timestep messages

xdir = None
(string) - Executable path, executes xclawcmd in xdir

Contents

4.3.2 Using PyClaw's solvers: Classic and SharpClaw

At present, PyClaw includes two types of solvers:

- Classic: the original Clawpack algorithms, in 1/2/3D
- SharpClaw: higher-order wave propagation using WENO reconstruction and Runge-Kutta integration, in 1/2D

Solver initialization takes one argument: a Riemann solver, usually from the Riemann repository. Typically, all that is needed to select a different solver is to specify it in the problem script, e.g.

```
>>> from clawpack import pyclaw
>>> from clawpack import riemann
>>> solver = pyclaw.ClawSolver2D(riemann.acoustics_2D)
```

for the 2D acoustics equations and the Classic Clawpack solver or

```
>>> solver = pyclaw.SharpClawSolver2D(riemann.acoustics_2D)
```

for the SharpClaw solver. Most of the applications distributed with PyClaw are set up to use either solver, depending on the value of the command line option *solver_type*, which should be set to *classic* or *sharpclaw*.

Typically, for a given grid resolution, the SharpClaw solvers are more accurate but also more computationally expensive. For typical problems involving shocks, the Classic solvers are recommended. For problems involving high-frequency waves, turbulence, or smooth solutions, the SharpClaw solvers may give more accurate solutions at less cost. This is an active area of research and you may wish to experiment with both solvers.

Future plans include incorporation of finite-difference and discontinuous Galerkin solvers.

Key differences between the Classic and SharpClaw solvers are:

- The source term routine for the Classic solver should return the integral of the source term over a step, while the source term routine for SharpClaw should return the instantaneous value of the source term.
- The solvers have different options. For a list of options and possible values, see the documentation of the `ClawSolver` and `SharpClawSolver` (page 209) classes.

SharpClaw Solvers

The SharpClaw solvers are a collection of solvers that contain the functionality of the Fortran code SharpClaw, developed in David Ketcheson's thesis. The 1D SharpClaw solver contains a pure Python implementation as well as a wrapped Fortran version. The 2D solver is in progress but not available yet. The SharpClaw solvers provide an interface similar to that of the classic Clawpack solvers, but with a few different options. The superclass solvers are not meant to be used separately but are there to provide common routines for all the Clawpack solvers. Please refer to each of the inherited classes for more info about the methods and attributes they provide each class. .. The inheritance structure is:

Example This is a simple example of how to instantiate and evolve a solution to a later time

```
>>> from clawpack import pyclaw
>>> solver = pyclaw.SharpClawSolver1D()            # Instantiate a default, 1d solver

>>> solver.evolve_to_time(solution,t_end)   # Evolve the solution to t_end
```

pyclaw.sharpclaw

class pyclaw.sharpclaw.solver.**SharpClawSolver**(*riemann_solver=None*, *claw_package=None*)

Superclass for all SharpClawND solvers.

Implements Runge-Kutta time stepping and the basic form of a semi-discrete step (the dq() function). If another method-of-lines solver is implemented in the future, it should be based on this class, which then ought to be renamed to something like "MOLSolver".

before_step
Function called before each time step is taken. The required signature for this function is:

def before_step(solver,solution)

lim_type
Limiter(s) to be used. 0: No limiting. 1: TVD reconstruction. 2: WENO reconstruction. `Default = 2`

weno_order
Order of the WENO reconstruction. From 1st to 17th order (PyWENO) `Default = 5`

time_integrator
Time integrator to be used. Currently implemented methods:

'Euler' : 1st-order Forward Euler integration 'SSP33' : 3rd-order strong stability preserving method of Shu & Osher 'SSP104' : 4th-order strong stability preserving method Ketcheson 'SSPMS32': 2nd-order strong stability preserving 3-step linear multistep method,

Unexpected indentation.

using Euler for starting values

Block quote ends without a blank line; unexpected unindent.

'SSPMS43': 3rd-order strong stability preserving 4-step linear multistep method using SSPRK22 for starting values

'RK' [Arbitrary Runge-Kutta method, specified by setting *solver.a*] and *solver.b* to the Butcher arrays of the method.

'LMM' [Arbitrary linear multistep method, specified by setting the] coefficient arrays *solver.alpha* and *solver.beta*.

`Default = 'SSP104'`

char_decomp
Type of WENO reconstruction. 0: conservative variables WENO reconstruction (standard). 1: Wave-slope reconstruction. 2: characteristic-wise WENO reconstruction. 3: transmission-based WENO reconstruction. `Default = 0`

tfluct_solver
Whether a total fluctuation solver have to be used. If True the function that calculates the total fluctuation must be provided. `Default = False`

tfluct
Pointer to Fortran routine to calculate total fluctuation `Default = default_tfluct (None)`

aux_time_dep
Whether the auxiliary array is time dependent. `Default = False`

kernel_language
Specifies whether to use wrapped Fortran routines ('Fortran') or pure Python ('Python'). `Default = 'Fortran'`.

num_ghost
Number of ghost cells. `Default = 3`

fwave
Whether to split the flux jump (rather than the jump in Q) into waves; requires that the Riemann solver performs the splitting. `Default = False`

cfl_desired
Desired CFL number. `Default = 2.45`

cfl_max
Maximum CFL number. `Default = 2.50`

dq_src
Whether a source term is present. If it is present the function that computes its contribution must be provided. `Default = None`

call_before_step_each_stage
Whether to call the method *self.before_step* before each RK stage. `Default = False`

dq(*state*)
Evaluate dq/dt * (delta t)

dqdt(*state*)
Evaluate dq/dt. This routine is used for implicit time stepping.

get_cfl_max()
Set maximum CFL number for current step depending on time integrator

get_dt_new()
Set time-step for next step depending on time integrator

setup(*solution*)
Allocate RK stage arrays or previous step solutions and fortran routine work arrays.

step(*solution*)
Evolve q over one time step.

Take one step with a Runge-Kutta or multistep method as specified by *solver.time_integrator*.

Pyclaw Classic Clawpack Solvers

The pyclaw classic clawpack solvers are a collection of solvers that represent the functionality of the older versions of clawpack. It comes in two forms, a pure python version and a python wrapping of the fortran libraries. All of the solvers available provide the same basic interface and provide the same options as the old versions of clawpack. The superclass solvers are not meant to be used separately but there to provide common routines for all the Clawpack solvers. Please refer to each of the inherited classes for more info about the methods and attributes they provide each class. .. The inheritance structure is:

Example This is a simple example of how to instantiate and evolve a solution to a later time

```
>>> from clawpack import pyclaw
>>> solver = pyclaw.ClawSolver1D()                    # Instantiate a default, 1d solver
>>> solver.limiters = pyclaw.limiters.tvd.vanleer  # Use the van Leer limiter
>>> solver.dt = 0.0001                                # Set the initial time step
>>> solver.max_steps = 500                            # Set the maximum number of time steps

>>> solver.evolve_to_time(solution,t_end)   # Evolve the solution to t_end
```

pyclaw.classic.solver

class clawpack.pyclaw.classic.solver.**ClawSolver**(*riemann_solver=None*, *claw_package=None*)

Generic classic Clawpack solver

All Clawpack solvers inherit from this base class.

mthlim
Limiter(s) to be used. Specified either as one value or a list. If one value, the specified limiter is used for all wave families. If a list, the specified values indicate which limiter to apply to each wave family. Take a look at pyclaw.limiters.tvd for an enumeration. `Default = limiters.tvd.minmod`

order
Order of the solver, either 1 for first order (i.e., Godunov's method) or 2 for second order (Lax-Wendroff-LeVeque). `Default = 2`

source_split
Which source splitting method to use: 1 for first order Godunov splitting and 2 for second order Strang splitting. `Default = 1`

fwave
Whether to split the flux jump (rather than the jump in Q) into waves; requires that the Riemann solver performs the splitting. `Default = False`

step_source
Handle for function that evaluates the source term. The required signature for this function is:

def step_source(solver,state,dt)

before_step
Function called before each time step is taken. The required signature for this function is:

def before_step(solver,solution)

kernel_language
Specifies whether to use wrapped Fortran routines ('Fortran') or pure Python ('Python'). `Default = 'Fortran'`.

verbosity
The level of detail of logged messages from the Fortran solver. `Default = 0`.

setup(*solution*)
Perform essential solver setup. This routine must be called before solver.step() may be called.

step(*solution*)
Evolve solution one time step

The elements of the algorithm for taking one step are:

1. The `before_step()` (page 212) function is called
2. A half step on the source term `step_source()` (page 212) if Strang splitting is being used (`source_split` (page 212) = 2)
3. A step on the homogeneous problem $q_t + f(q)_x = 0$ is taken
4. A second half step or a full step is taken on the source term `step_source()` (page 212) depending on whether Strang splitting was used (`source_split` (page 212) = 2) or Godunov splitting (`source_split` (page 212) = 1)

This routine is called from the method evolve_to_time defined in the pyclaw.solver.Solver superclass.

Input

- *solution* - (`Solution` (page 221)) solution to be evolved

Output

- (bool) - True if full step succeeded, False otherwise

step_hyperbolic(*solution*)
Take one homogeneous step on the solution.

This is a dummy routine and must be overridden.

Change to Custom BC Function Signatures

To allow better access to aux array data in the boundary condition functions both the *qbc* and *auxbc* arrays are now passed to the custom boundary condition functions. The new signature is

def my_custom_BC(state, dim, t, qbc, auxbc, num_ghost): ...

and should be adopted as soon as possible. The old signature

def my_custom_BC(state, dim, t, bc_array, num_ghost): ...

can still be used but a warning will be issued and the old signature will not be supported when version 6.0 is released. This addition is available in versions > 5.2.0.

4.3.3 Pyclaw Limiters

Note: Need to provide short explanation of limiters here.

pyclaw.limiters.tvd.**limit**(*num_eqn*, *wave*, *s*, *limiter*, *dtdx*)
Apply a limiter to the waves

Function that limits the given waves using the methods contained in limiter. This is the vectorized version of the function acting on a row of waves at a time.

Input

- *wave* - (ndarray(:,num_eqn,num_waves)) The waves at each interface
- *s* - (ndarray(:,num_waves)) Speeds for each wave
- ***limiter*** **- (int list) Array of type int determining which** limiter to use
- ***dtdx*** **- (ndarray(:))** $\Delta t/\Delta x$ **ratio, used for CFL** dependent limiters

Output

- (ndarray(:,num_eqn,num_waves)) - Returns the limited waves

Version 1.1 (2009-07-05)

pyclaw.limiters.tvd

Library of limiter functions to be applied to waves

This module contains all of the standard limiters found in clawpack. To use any of the limiters, use the function limit to limit the appropriate waves. Refer to each limiter and the function limit's doc strings.

This is a list of the provided limiters and their corresponding method number, note that some of the limiters actually correspond to a more general function which can be controlled more directly. Refer to the limiter function and its corresponding documentation for details.

CFL Independent Limiters

1. minmod - minmod_limiter() (page 215)
2. superbee - superbee_limiter() (page 215)
3. van leer - $(r + |r|)/(1 + |r|)$
4. mc - mc_limiter() (page 215)
5. Beam-warming - r
6. Frommm - $1/2(1 + r)$
7. Albada 2 - $(r^2 + r)/(1 + r^2)$
8. Albada 3 - $1/2(1 + r)(1 - (|1 - r|^3)/(1 + |r|^3))$
9. van Leer with Klein sharpening, k=2 - van_leer_klein_sharpening_limiter() (page 215)

CFL Dependent Limiters

10. Roe's linear third order scheme - $1 + (r - 1)(1 + cfl)/3$
11. Arora-Roe (= limited version of the linear third order scheme) - arora_roe() (page 215)
12. Theta Limiter, theta=0.95 (safety on nonlinear waves) - theta_limiter() (page 215)
13. Theta Limiter, theta=0.75 - theta_limiter() (page 215)
14. Theta Limiter, theta=0.5 - theta_limiter() (page 215)
15. CFL-Superbee (Roe's Ultrabee) - cfl_superbee() (page 215)
16. CFL-Superbee (Roe's Ultrabee) with theta=0.95 (nonlinear waves) - cfl_superbee_theta() (page 215)
17. beta=2/3 limiter - beta_limiter() (page 215)
18. beta=2/3 limiter with theta=0.95 (nonlinear waves) - beta_limiter() (page 215)
19. Hyperbee - hyperbee_limiter() (page 215)
20. SuperPower - superpower_limiter() (page 215)
21. Cada-Torrilhon modified - cada_torrilhon_limiter() (page 215)
22. Cada-Torrilhon modified, version for nonlinear waves - cada_torrilhon_limiter_nonlinear() (page 215)
23. upper bound limiter (1st order) - upper_bound_limiter() (page 215)

All limiters have the same function call signature:

Input

- *r* - (ndarray(:))
- *cfl* - (ndarray(:)) Local CFL number

Output

- (ndarray(:)) -

Newer limiters are based on work done by Friedemann Kemm [kemm_2009], paper in review.

Authors Kyle Mandli and Randy LeVeque (2008-08-21) Initial version

Kyle Mandli (2009-07-05) Added CFL depdendent limiters

pyclaw.limiters.tvd.**arora_roe**(*r*, *cfl*)
Arora-Roe limiter, limited version of the linear third order scheme

pyclaw.limiters.tvd.**beta_limiter**(*r*, *cfl*, *theta=0.95*, *beta=0.6666666666666666*)
Modification of CFL Superbee limiter with theta and beta parameters

Additional Input:

- *theta*
- *beta*

pyclaw.limiters.tvd.**cada_torrilhon_limiter**(*r*, *cfl*, *epsilon=0.001*)
Cada-Torrilhon modified

Additional Input:

- *epsilon* =

pyclaw.limiters.tvd.**cada_torrilhon_limiter_nonlinear**(*r*, *cfl*)
Cada-Torrilhon modified, version for nonlinear waves

pyclaw.limiters.tvd.**cfl_superbee**(*r*, *cfl*)
CFL-Superbee (Roe's Ultrabee) without theta parameter

pyclaw.limiters.tvd.**cfl_superbee_theta**(*r*, *cfl*, *theta=0.95*)
CFL-Superbee (Roe's Ultrabee) with theta parameter

pyclaw.limiters.tvd.**hyperbee_limiter**(*r*, *cfl*)
Hyperbee

pyclaw.limiters.tvd.**mc_limiter**(*r*, *cfl*)
MC vectorized limiter

pyclaw.limiters.tvd.**minmod_limiter**(*r*, *cfl*)
Minmod vectorized limiter

pyclaw.limiters.tvd.**superbee_limiter**(*r*, *cfl*)
Superbee vectorized limiter

pyclaw.limiters.tvd.**superpower_limiter**(*r*, *cfl*, *caut=1.0*)
SuperPower limiter

Additional input:

- *caut* = Limiter parameter

pyclaw.limiters.tvd.**theta_limiter**(*r*, *cfl*, *theta=0.95*)
Theta limiter

Additional Input:

- *theta* =

pyclaw.limiters.tvd.**upper_bound_limiter**(*r*, *cfl*, *theta=1.0*)
Upper bound limiter (1st order)

Additional Input:

- *theta* =

pyclaw.limiters.tvd.**van_leer_klein_sharpening_limiter**(*r*, *cfl*)
van Leer with Klein sharpening, k=2

4.3.4 Pyclaw Input/Output Package

Pyclaw supports the following input and output formats:

- ASCII (page 217) - ASCII file I/O, supports traditional clawpack format files
- HDF5 (page 218) - HDF5 file I/O
- NetCDF (page 219) - NetCDF file I/O, support for NetCDF3 and NetCDF4 files

Each module contains two main routines `read_<format>` and `write_<format>` which `Solution` can call with the appropriate <format>. In order to create a new file I/O extension the calling signature must match

```
read_<format>(solution,frame,path,file_prefix,write_aux,options)
```

where the the inputs are

Input

- *solution* - (`Solution` (page 221)) Pyclaw object to be output
- *frame* - (int) Frame number
- *path* - (string) Root path
- *file_prefix* - (string) Prefix for the file name.
- *write_aux* - (bool) Boolean controlling whether the associated auxiliary array should be written out.
- *options* - (dict) Optional argument dictionary

and

```
write_<format>(solution,frame,path,file_prefix,write_aux,options)
```

where the inputs are

Input

- *solution* - (`Solution` (page 221)) Pyclaw object to be output
- *frame* - (int) Frame number
- *path* - (string) Root path
- *file_prefix* - (string) Prefix for the file name.
- *write_aux* - (bool) Boolean controlling whether the associated auxiliary array should be written out.
- *options* - (dict) Optional argument dictionary.

Note that both allow for an `options` dictionary that is format specific and should be documented thoroughly. For examples of this usage, see the HDF5 (page 218) and NetCDF (page 219) modules.

HDF5 (page 218) and NetCDF (page 219) support require installed libraries in order to work, please see the respective modules for details on how to obtain and install the libraries needed.

Note: Pyclaw automatically detects the availability of HDF5 and NetCDF file support and will warn you if you try and use them without the proper libraries.

---

pyclaw.io.ascii

Routines for reading and writing an ascii output file

pyclaw.io.ascii.**read**(*solution*, *frame*, *path='./'*, *file_prefix='fort'*, *read_aux=False*, *options={}*)
Read in a frame of ascii formatted files, and enter the data into the solution object.

This routine reads the ascii formatted files corresponding to the classic clawpack format 'fort.txxxx', 'fort.qxxxx', and 'fort.axxxx' or 'fort.aux' Note that the fort prefix can be changed.

Input

- *solution* - (Solution (page 221)) Solution object to read the data into.
- *frame* - (int) Frame number to be read in
- *path* - (string) Path to the current directory of the file
- *file_prefix* - (string) Prefix of the files to be read in. `default = 'fort'`
- *read_aux* (bool) Whether or not an auxillary file will try to be read in. `default = False`
- *options* - (dict) Dictionary of optional arguments dependent on the format being read in. `default = {}`

pyclaw.io.ascii.**read_array**(*f*, *state*, *num_var*)
Read in an array from an ASCII output file f.

The variable q here may in fact refer to q or to aux.

This routine supports the possibility that the values q[:,i,j,k] (for a fixed i,j,k) have been split over multiple lines, because some packages write just 4 values per line. For Clawpack 6.0, we plan to make all packages write q[:,i,j,k] on a single line. This routine can then be simplified.

pyclaw.io.ascii.**read_t**(*frame*, *path='./'*, *file_prefix='fort'*)
Read only the fort.t file and return the data

Input

- *frame* - (int) Frame number to be read in
- *path* - (string) Path to the current directory of the file
- *file_prefix* - (string) Prefix of the files to be read in. `default = 'fort'`

Output

- (list) List of output variables
- *t* - (int) Time of frame
- *num_eqn* - (int) Number of equations in the frame
- *nstates* - (int) Number of states
- *num_aux* - (int) Auxillary value in the frame
- *num_dim* - (int) Number of dimensions in q and aux

pyclaw.io.ascii.**write**(*solution*, *frame*, *path*, *file_prefix='fort'*, *write_aux=False*, *options={}*, *write_p=False*)
Write out ascii data file

Write out an ascii file formatted identical to the fortran clawpack files including writing out fort.t, fort.q, and fort.aux if necessary. Note that there are some parameters that assumed to be the same for every patch in this format which is not necessarily true for the actual data objects. Make sure that if you use this output format that

all of you patchs share the appropriate values of num_dim, num_eqn, num_aux, and t. Only supports up to 3 dimensions.

Input

- *solution* - (`Solution` (page 221)) Pyclaw object to be output.
- *frame* - (int) Frame number
- *path* - (string) Root path
- *file_prefix* - (string) Prefix for the file name. `default = 'fort'`
- *write_aux* - (bool) Boolean controlling whether the associated auxiliary array should be written out. `default = False`
- *options* - (dict) Dictionary of optional arguments dependent on the format being written. `default = {}`

`pyclaw.io.ascii.`**`write_array`**(*f*, *patch*, *q*)

Write a single array to output file f as ASCII text.

The variable q here may in fact refer to q or to aux.

pyclaw.io.hdf5

Routines for reading and writing a HDF5 output file

This module reads and writes hdf5 files via either of the following modules: h5py - http://code.google.com/p/h5py/ PyTables - http://www.pytables.org/moin

It will first try h5py and then PyTables and use the correct calls according to whichever is present on the system. We recommend that you use h5py as it is a minimal wrapper to the HDF5 library.

To install either, you must also install the hdf5 library from the website: http://www.hdfgroup.org/HDF5/release/obtain5.html

`pyclaw.io.hdf5.`**`read`**(*solution*, *frame*, *path='./'*, *file_prefix='claw'*, *read_aux=True*, *options={}*)

Read in a HDF5 file into a Solution

Input

- *solution* - (`Solution` (page 221)) Pyclaw object to be output
- *frame* - (int) Frame number
- *path* - (string) Root path
- *file_prefix* - (string) Prefix for the file name. `default = 'claw'`
- *write_aux* - (bool) Boolean controlling whether the associated auxiliary array should be written out. `default = False`
- *options* - (dict) Optional argument dictionary, unused for reading.

`pyclaw.io.hdf5.`**`write`**(*solution*, *frame*, *path*, *file_prefix='claw'*, *write_aux=False*, *options={}*, *write_p=False*)

Write out a Solution to a HDF5 file.

Input

- *solution* - (`Solution` (page 221)) Pyclaw solution object to input into
- *frame* - (int) Frame number
- *path* - (string) Root path

- *file_prefix* - (string) Prefix for the file name. `default = 'claw'`
- *write_aux* - (bool) Boolean controlling whether the associated auxiliary array should be written out. `default = False`
- *options* - (dict) Optional argument dictionary, see HDF5 Option Table (page 219)

Key	Value
compression	(None, string [”gzip” | “lzf” | “szip”] or int 0-9) Enable dataset compression. DEFLATE, LZF and (where available) SZIP are supported. An integer is interpreted as a GZIP level for backwards compatibility.
compression_opts	(None, or special value) Setting for compression filter; legal values for each filter type are: •*gzip* - (int) 0-9 •*lzf* - None allowed •***szip* - (tuple) 2-tuple (‘ec’|’nn’, even integer** 0-32) See the filters module for a detailed description of each of these filters.
chunks	(None, True or shape tuple) Store the dataset in chunked format. Automatically selected if any of the other keyword options are given. If you don’t provide a shape tuple, the library will guess one for you.
shuffle	(True/False) Enable/disable data shuffling, which can improve compression performance. Automatically enabled when compression is used.
fletcher32	(True/False) Enable Fletcher32 error detection; may be used with or without compression.

pyclaw.io.netcdf

Routines for reading and writing a NetCDF output file

Routines for reading and writing a NetCDF output file via either

- netcdf4-python - http://code.google.com/p/netcdf4-python/
- pupynere - http://pypi.python.org/pypi/pupynere/

These interfaces are very similar so if a different module needs to be used, it can more than likely be inserted with a minimal of effort.

This module will first try to import the netcdf4-python module which is based on the compiled libraries and failing that will attempt to import the pure python interface pupynere which requires no libraries.

To install the netCDF 4 library, please see: http://www.unidata.ucar.edu/software/netcdf/

Authors Kyle T. Mandli (2009-02-17) Initial version

`pyclaw.io.netcdf.`**`read`**(*solution*, *frame*, *path='./'*, *file_prefix='claw'*, *read_aux=True*, *options={}*)

Read in a NetCDF data files into solution

Input

- *solution* - (`Solution` (page 221)) Pyclaw object to be output
- *frame* - (int) Frame number
- *path* - (string) Root path

- *file_prefix* - (string) Prefix for the file name. `default = 'claw'`
- *write_aux* - (bool) Boolean controlling whether the associated auxiliary array should be written out. `default = False`
- *options* - (dict) Optional argument dictionary, unused for reading.

`pyclaw.io.netcdf.`**`write`**(*solution*, *frame*, *path*, *file_prefix='claw'*, *write_aux=False*, *options={}*, *write_p=False*)

Write out a NetCDF data file representation of solution

Input

- *solution* - (`Solution` (page 221)) Pyclaw object to be output
- *frame* - (int) Frame number
- *path* - (string) Root path
- *file_prefix* - (string) Prefix for the file name. `default = 'claw'`
- *write_aux* - (bool) Boolean controlling whether the associated auxiliary array should be written out. `default = False`
- *options* - (dict) Optional argument dictionary, see NetCDF Option Table (page 220)

Key	Value
description	Dictionary of key/value pairs that will be attached to the root group as attributes, i.e. {'time':3}
format	Can be one of the following netCDF flavors: NETCDF3_CLASSIC, NETCDF3_64BIT, NETCDF4_CLASSIC, and NETCDF4 `default = NETCDF4`
clobber	if True (Default), file will be overwritten, if False an exception will be raised
zlib	if True, data assigned to the Variable instance is compressed on disk. `default = False`
complevel	the level of zlib compression to use (1 is the fastest, but poorest compression, 9 is the slowest but best compression). Ignored if zlib=False. `default = 6`
shuffle	if True, the HDF5 shuffle filter is applied to improve compression. Ignored if zlib=False. `default = True`
fletcher32	if True (default False), the Fletcher32 checksum algorithm is used for error detection.
contiguous	if True (default False), the variable data is stored contiguously on disk. Setting to True for a variable with an unlimited dimension will trigger an error. `default = False`
chunksizes	Can be used to specify the HDF5 chunksizes for each dimension of the variable. A detailed discussion of HDF chunking and I/O performance is available here. Basically, you want the chunk size for each dimension to match as closely as possible the size of the data block that users will read from the file. chunksizes cannot be set if contiguous=True.
least_significant_digit	If specified, variable data will be truncated (quantized). In conjunction with zlib=True this produces 'lossy', but significantly more efficient compression. For example, if least_significant_digit=1, data will be quantized using around (scale*data)/scale, where scale = 2**bits, and bits is determined so that a precision of 0.1 is retained (in this case bits=4). `default = None`, or no quantization.
endian	Can be used to control whether the data is stored in little or big endian format on disk. Possible values are little, big or native (default). The library will automatically handle endian conversions when the data is read, but if the data is always going to be read on a computer with the opposite format as the one used to create the file, there may be some performance advantage to be gained by setting the endian-ness.
fill_value	If specified, the default netCDF _FillValue (the value that the variable gets filled with before any data is written to it) is replaced with this value. If fill_value is set to False, then the variable is not pre-filled.

Note: The zlib, complevel, shuffle, fletcher32, contiguous, chunksizes and endian keywords are silently ignored for netCDF 3 files that do not use HDF5.

4.3.5 PyClaw Solutions

PyClaw `Solution` (page 221) objects are containers for `State` (page 224) and `Domain` (page 230) objects that define an entire solution. The `State` (page 224) class is responsible for containing all the data of the solution on the given `Domain` (page 230). The `Domain` (page 230) is responsible for containing the geometry of the `Solution` (page 221). The structure of a solution may look something like the *figure* (page 221).

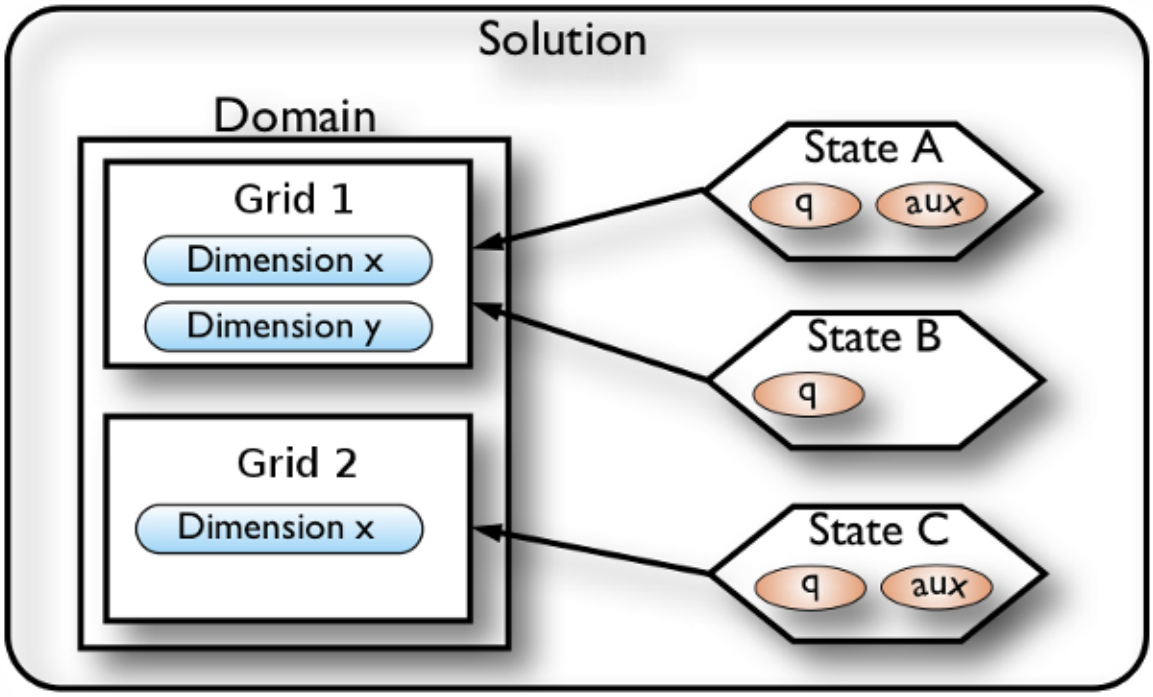


Figure 4.1: Pyclaw solution structure including a `Domain` (page 230), a set of `Patch` (page 230) objects and corresponding `Dimension` (page 234) objects defining the solution's geometry and three `State` (page 224) objects pointing to the appropriate `Patch` (page 230) with varying fields.

List of serial and parallel objects in a `Solution` (page 221) class:

Serial	Parallel
`pyclaw.state.State` (page 224)	`petclaw.state.State` (page 226)
`pyclaw.geometry.Domain` (page 230)	`petclaw.geometry.Domain` (page 236)
`pyclaw.geometry.Patch` (page 230)	`petclaw.geometry.Patch` (page 237)
`pyclaw.geometry.Grid` (page 231)	
`pyclaw.geometry.Dimension` (page 234)	

`pyclaw.solution.Solution`

class `pyclaw.solution.`**`Solution`**(**arg*, ***kargs*)

Pyclaw patch container class

Input and Output Input and output of solution objects is handle via the io package. Solution contains the generic methods `write()`, `read()` and `plot()` which then figure out the correct

method to call. Please see the io package for the particulars of each format and method and the methods in this class for general input and output information.

Properties If there is only one state and patch belonging to this solution, the solution will appear to have many of the attributes assigned to its one state and patch. Some parameters that have in the past been parameters for all patch,s are also reachable although Solution does not check to see if these parameters are truly universal.

Patch Attributes: 'dimensions'

State Attributes: 't','num_eqn','q','aux','capa','problem_data'

Initialization The initialization of a Solution can happen one of these ways

1. *args* is empty and an empty Solution is created
2. *args* is an integer (the number of components of q), a single State, or a list of States and is followed by the appropriate *geometry* (page 227) object which can be one of:
   - (`Domain` (page 230))
   - (`Patch` (page 230)) - A domain is created with the patch or list of patches provided.
   - (`Dimension` (page 234)) - A domain and patch is created with the dimensions or list of dimensions provided.
3. *args* is a variable number of arguments that describes the location of a file to be read in to initialize the object

Examples

```
>>> import clawpack.pyclaw as pyclaw
>>> x = pyclaw.Dimension('x',0.,1.,100)
>>> domain = pyclaw.Domain((x))
>>> state = pyclaw.State(domain,3,2)
>>> solution = pyclaw.Solution(state,domain)
```

is_valid()
Checks to see if this solution is valid

The Solution checks to make sure it is valid by checking each of its states. If an invalid state is found, a message is logged what specifically made this solution invalid.

Output

- (bool) - True if valid, false otherwise

plot()
Plot the solution

read(*frame*, *path='./_output'*, *file_format='ascii'*, *file_prefix=None*, *read_aux=True*, *options={}*, ***kargs*)
Reads in a Solution object from a file

Reads in and initializes this Solution with the data specified. This function will raise an IOError if it was unsuccessful.

Any format must conform to the following call signiture and return True if the file has been successfully read into the given solution or False otherwise. Options is a dictionary of parameters that each format can specify. See the ascii module for an example.:

```
read_<format>(solution,path,frame,file_prefix,options={})
```

`<format>` is the name of the format in question.

Input

- *frame* - (int) Frame number to be read in
- *path* - (string) Base path to the files to be read. `default = './_output'`
- *file_format* - (string) Format of the file, should match on of the modules inside of the io package. `default = 'ascii'`
- *file_prefix* - (string) Name prefix in front of all the files, defaults to whatever the format defaults to, e.g. fort for ascii
- *options* - (dict) Dictionary of optional arguments dependent on the format being read in. `default = {}`

Output

- (bool) - True if read was successful, False otherwise

set_all_states(*attr*, *value*, *overwrite=True*)
Sets all member states attribute 'attr' to value

Input

- *attr* - (string) Attribute name to be set
- *value* - (id) Value for attribute
- *overwrite* - (bool) Whether to overwrite the attribute if it already exists. `default = True`

write(*frame*, *path='./'*, *file_format='ascii'*, *file_prefix=None*, *write_aux=False*, *options={}*, *write_p=False*)
Write out a representation of the solution

Writes out a suitable representation of this solution object based on the format requested. The path is built from the optional path and file_prefix arguments. Will raise an IOError if unsuccessful.

Input

- *frame* - (int) Frame number to append to the file output
- *path* - (string) Root path, will try and create the path if it does not already exist. `default = './'`
- *format* - (string or list of strings) a string or list of strings containing the desired output formats. `default = 'ascii'`
- *file_prefix* - (string) Prefix for the file name. Defaults to the particular io modules default.
- *write_aux* - (book) Write the auxillary array out as well if present. `default = False`
- *options* - (dict) Dictionary of optional arguments dependent on which format is being used. `default = {}`

patch
(`Patch`) - Base state's patch is returned

start_frame
(int) - : Solution start frame number in case the *Solution* object is initialized by loading frame from file

state
(`State`) - Base state is returned

4.3.6 PyClaw State

The State (page 224) object records the fields that exist on a given Patch (page 230). These fields include q and aux. The State (page 224) also includes references to the Patch (page 230) that the state belongs to.

In parallel the State (page 226) object also handles some of the parallel communication required of the state on the given patch such that only the parts of the fields local to the process. If you are interested in the geometry of the local state you can find it through the Patch (page 237) object's reference to its own Grid.

Serial pyclaw.state.State

class pyclaw.state.**State**(*geom*, *num_eqn*, *num_aux=0*)

A PyClaw State object contains the current state on a particular patch, including the unkowns q, the time t, and the auxiliary coefficients aux.

The variables num_eqn and num_aux determine the length of the first dimension of the q and aux arrays.

State Data The arrays q, and aux have variable extents based on the patch dimensions and the values of num_eqn and num_aux.

A State object is automatically created upon instantiation of a Solution object from a Domain object:

```
>>> from clawpack import pyclaw
>>> x = pyclaw.Dimension('x',0.0,1.0,100)
>>> domain = pyclaw.Domain(x)
>>> num_eqn = 1
>>> solution = pyclaw.Solution(num_eqn,domain)
>>> print solution.state
PyClaw State object
Patch dimensions: [100]
Time  t=0.0
Number of conserved quantities: 1
```

A State lives on a Patch, and can be instantiated directly by first creating a Patch:

```
>>> x = pyclaw.Dimension('x',0.,1.,100)
>>> patch = pyclaw.Patch((x))
```

The arguments to the constructor are the patch, the number of equations, and the number of auxiliary fields:

```
>>> state = pyclaw.State(patch,3,2)
>>> state.q.shape
(3, 100)
>>> state.aux.shape
(2, 100)
>>> state.t
0.0
```

Note that state.q and state.aux are initialized as empty arrays (not zeroed). Additional parameters, such as scalar values that are used in the Riemann solver, can be set using the dictionary state.problem_data.

get_aux_global()

Returns a copy of state.aux.

get_auxbc_from_aux(*num_ghost*, *auxbc*)

Fills in the interior of auxbc by copying aux to it.

get_q_global()

Returns a copy of state.q.

get_qbc_from_q(*num_ghost*, *qbc*)
Fills in the interior of qbc by copying q to it.

is_valid()
Checks to see if this state is valid

The state is declared valid based on the following criteria:

- `q` is Fortran contiguous
- `aux` is Fortran contiguous

A debug logger message will be sent documenting exactly what was not valid.

Output

- (bool) - True if valid, false otherwise.

set_aux_from_auxbc(*num_ghost*, *auxbc*)
Set the value of aux using the array auxbc.

set_cparam(*fortran_module*)
Set the variables in fortran_module.cparam to the corresponding values in patch.problem_data. This is the mechanism for passing scalar variables to the Fortran Riemann solvers; cparam must be defined as a common block in the Riemann solver.

This function should be called from solver.setup(). This seems like a fragile interdependency between solver and state; perhaps problem_data should belong to solver instead of state.

This function also checks that the set of variables defined in cparam all appear in problem_data.

set_num_ghost(*num_ghost*)
Virtual routine (does nothing). Overridden in the petclaw.state class.

set_q_from_qbc(*num_ghost*, *qbc*)
Set the value of q using the array qbc. Typically this is called after qbc has been updated by the solver.

F = None
(ndarray(mF,...)) - Cell averages of output functional densities.

gauge_data = None
(list) - List of numpy.ndarray objects. Each element of the list stores the values of the corresponding gauge if `keep_gauges` is set to `True`

keep_gauges = None
(bool) - Keep gauge values in memory for every time step, `default = False`

mF
(int) - Number of output functionals

mp
(int) - Number of derived quantities

num_aux
(int) - Number of auxiliary fields

num_eqn
(int) - Number of unknowns (components of q)

p = None
(ndarray(mp,...)) - Cell averages of derived quantities.

problem_data = None
(dict) - Dictionary of global values for this patch, `default = {}`

t = None
(float) - Current time represented on this patch, `default = 0.0`

Parallel `petclaw.state.State`

class `petclaw.state.`**State**(*geom*, *num_eqn*, *num_aux=0*)

Parallel State class Unexpected section title.

```
Parent Class Documentation
==========================
```

Module containing all Pyclaw solution objects

Authors David I. Ketcheson – Initial version (June 2011)

get_aux_global()
Returns a copy of the global aux array on process 0, otherwise returns None

get_auxbc_from_aux(*num_ghost*, *auxbc*)
Returns aux with ghost cells attached, by accessing the local vector.

get_q_global()
Returns a copy of the global q array on process 0, otherwise returns None

get_qbc_from_q(*num_ghost*, *qbc*)
Returns q with ghost cells attached, by accessing the local vector.

set_num_ghost(*num_ghost*)
This is a hack to deal with the fact that petsc4py doesn't allow us to change the stencil_width (num_ghost).

Instead, we initially create DAs with stencil_width=0. Then, in solver.setup(), we call this function to replace those DAs with new ones that have the right stencil width.

This could be made more efficient using some PETSc calls, but it only happens once so it seems not to be worth it.

F
Array containing pointwise values (densities) of output functionals. This is just used as temporary workspace before summing.

aux
We never communicate aux values; every processor should set its own ghost cell values for the aux array. The global aux vector is used only for outputting the aux values to file; everywhere else we use the local vector.

fset
Array containing pointwise values (densities) of output functionals. This is just used as temporary workspace before summing.

gauge_data = None
(list) - List of numpy.ndarray objects. Each element of the list stores the values of the corresponding gauge if `keep_gauges` is set to `True`

keep_gauges = None
(bool) - Keep gauge values in memory for every time step, `default = False`

mF
(int) - Number of derived quantities (components of p)

mp
(int) - Number of derived quantities (components of p)

num_aux
(int) - Number of auxiliary fields

num_eqn
(int) - Number of unknowns (components of q)

p
Array containing values of derived quantities for output.

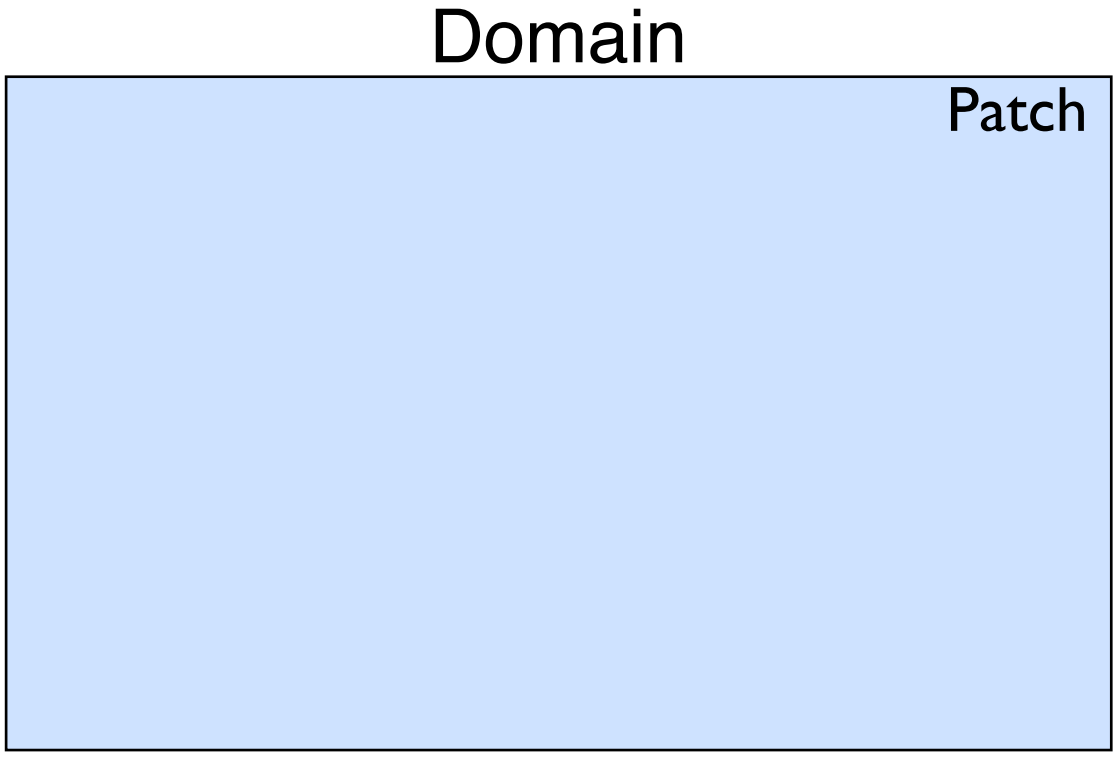
problem_data = None
(dict) - Dictionary of global values for this patch, `default = {}`

q
Array of solution values.

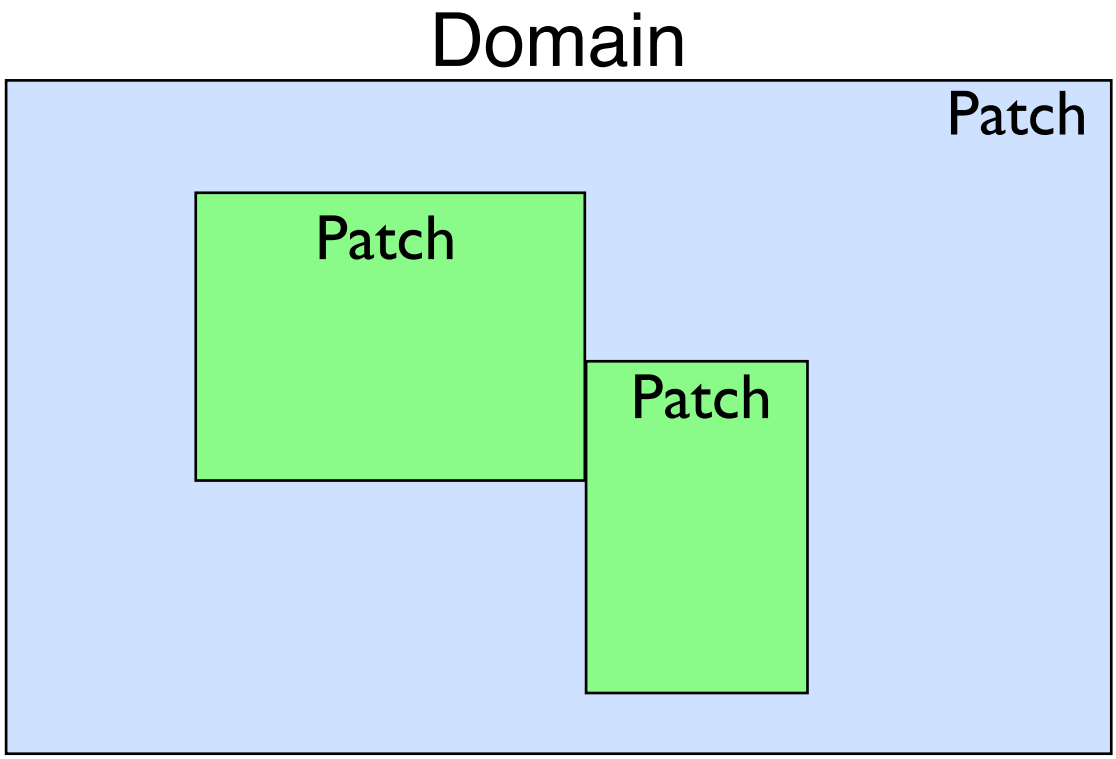
t = None
(float) - Current time represented on this patch, `default = 0.0`

4.3.7 PyClaw Geometry

The PyClaw geometry package contains the classes used to define the geometry of a `Solution` (page 221) object. The base container for all other geometry is the `Domain` (page 230) object. It contains a list of `Patch` (page 230) objects that reside inside of the `Domain` (page 230).

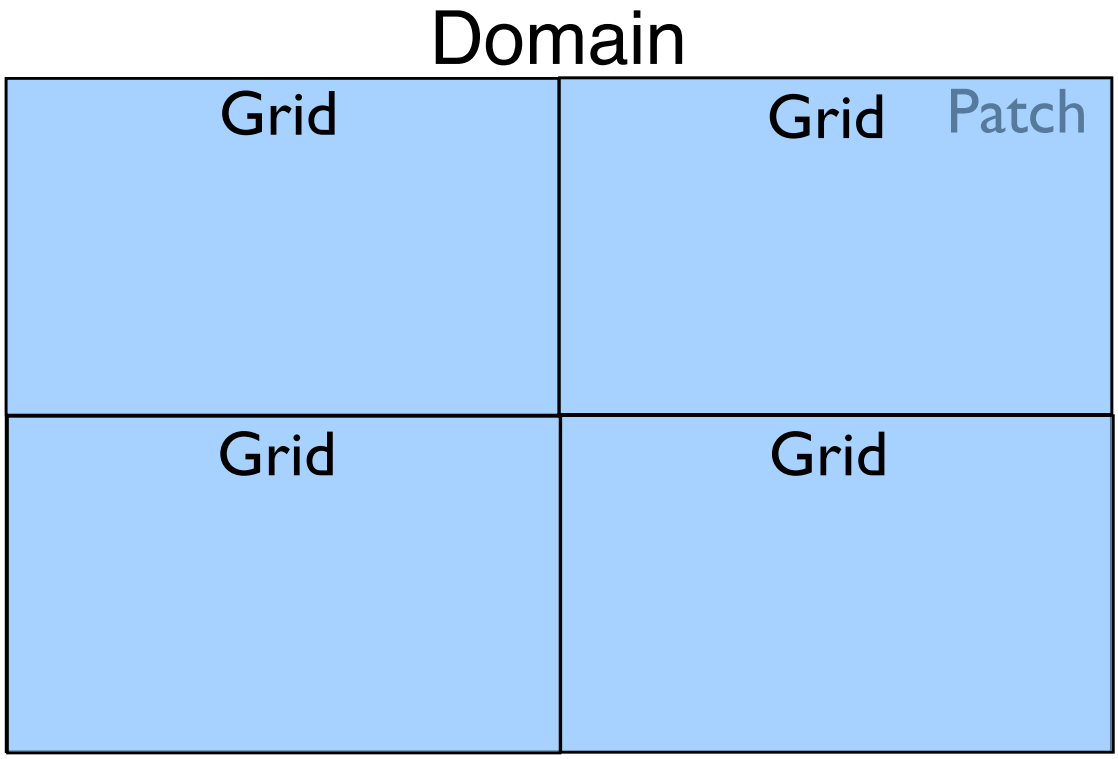


`Patch` (page 230) represents a piece of the domain that could be a different resolution than the others, have a different coordinate mapping, or be used to construct complex domain shapes.

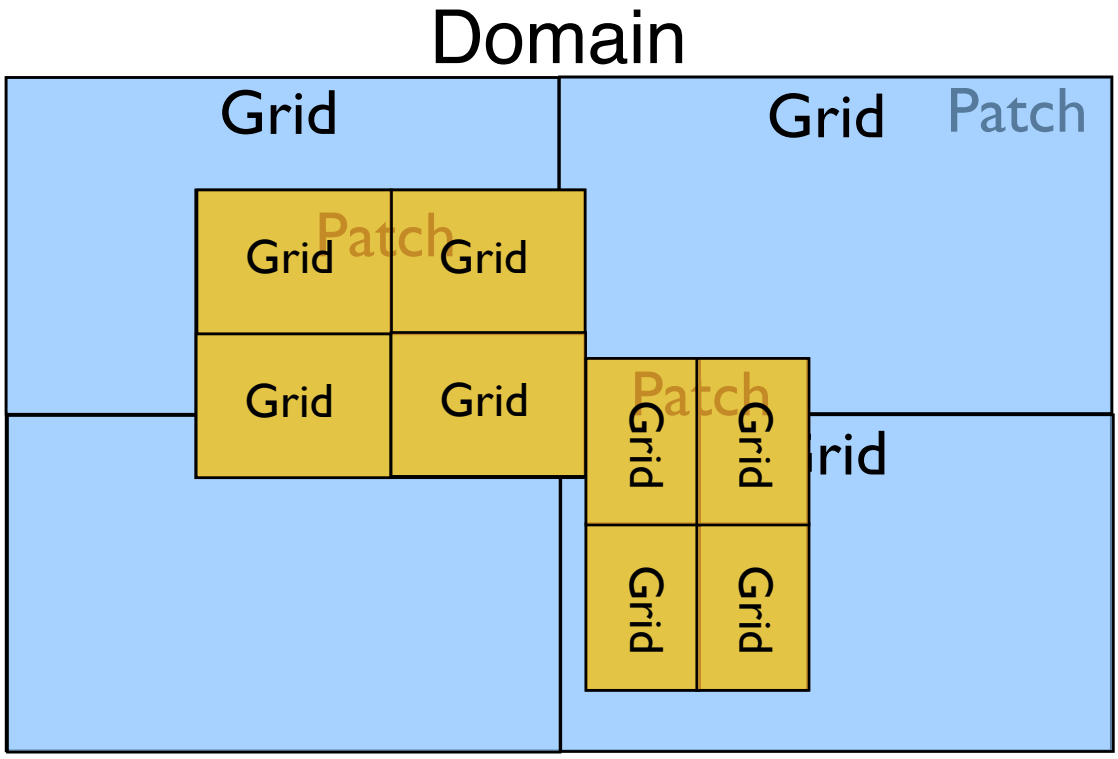


It contains `Dimension` (page 234) objects that define the extent of the `Patch` (page 230) and the number of grid cells in each dimension. `Patch` (page 230) also contains a reference to a nearly identical `Grid` (page 231) object. The `Grid` (page 231) object also contains a set of `Dimension` (page 234) objects describing its extent and number of grid cells. The `Grid` (page 231) is meant to represent the geometry of the data local to the process in the case of a parallel run. In a serial simulation the `Patch` (page 230) and `Grid` (page 231) share the same dimensions.

In the case where only one `Patch` (page 230) object exists in a `Domain` (page 230) but it is run with four processes in parallel, the `Domain` (page 230) hierarchy could look like:



In the most complex case with multiple patches and a parallel run we may have the following:



Serial Geometry Objects

pyclaw.geometry.Domain

class pyclaw.geometry.**Domain**(**arg*)

A Domain is a list of Patches.

A Domain may be initialized in the following ways:

1. **Using 3 arguments, which are in order**
   - A list of the lower boundaries in each dimension
   - A list of the upper boundaries in each dimension
   - A list of the number of cells to be used in each dimension
2. **Using a single argument, which is**
   - A list of dimensions; or
   - A list of patches.

Examples

```
>>> from clawpack import pyclaw
>>> domain = pyclaw.Domain( (0.,0.), (1.,1.), (100,100))
>>> print domain.num_dim
2
>>> print domain.grid.num_cells
[100, 100]
```

grid
(list) - Patch.grid of base patch

num_dim
(int) - Patch.num_dim of base patch

patch
(Patch) - First patch is returned

pyclaw.geometry.Patch

class pyclaw.geometry.**Patch**(*dimensions*)

Bases: object

Global Patch information Each patch has a value for level and patch_index.

add_dimension(*dimension*)
Add the specified dimension to this patch

Input

- *dimension* - (Dimension) Dimension to be added

get_dim_attribute(*attr*)
Returns a tuple of all dimensions' attribute attr

delta
(list) - List of computational cell widths

dimensions
(list) - List of `Dimension` objects defining the grid's extent and resolution

level = None
(int) - AMR level this patch belongs to, `default = 1`

lower_global
(list) - Lower coordinate extents of each dimension

name
(list) - List of names of each dimension

num_cells_global
(list) - List of the number of cells in each dimension

num_dim
(int) - Number of dimensions

patch_index = None
(int) - Patch number of current patch, `default = 0`

upper_global
(list) - Upper coordinate extends of each dimension

pyclaw.geometry.Grid

class `pyclaw.geometry.`**Grid**(*dimensions*)
Basic representation of a single grid in Pyclaw

Dimension information Each dimension has an associated name with it that can be accessed via that name such as `grid.x.num_cells` which would access the x dimension's number of cells.

Properties If the requested property has multiple values, a list will be returned with the corresponding property belonging to the dimensions in order.

Initialization

Input:

- *dimensions* - (list of `Dimension`) Dimensions that are to be associated with this grid

Output:

- (`grid`) Initialized grid object

A PyClaw grid is usually constructed from a tuple of PyClaw Dimension objects:

```
>>> from clawpack.pyclaw.geometry import Dimension, Grid
>>> x = Dimension('x',0.,1.,10)
>>> y = Dimension('y',-1.,1.,25)
>>> grid = Grid((x,y))
>>> print grid
Dimension x:  (num_cells,delta,[lower,upper]) = (10,0.1,[0.0,1.0])
Dimension y:  (num_cells,delta,[lower,upper]) = (25,0.08,[-1.0,1.0])
>>> grid.num_dim
2
>>> grid.num_cells
[10, 25]
>>> grid.lower
[0.0, -1.0]
```

```
>>> grid.delta # Returns [dx, dy]
[0.1, 0.08]
```

A grid can be extended to higher dimensions using the add_dimension() method:

```
>>> z=Dimension('z',-2.0,2.0,21)
>>> grid.add_dimension(z)
>>> grid.num_dim
3
>>> grid.num_cells
[10, 25, 21]
>>> grid.c_edges[0][0,0,0]
0.0
>>> grid.c_edges[1][0,0,0]
-1.0
>>> grid.c_edges[2][0,0,0]
-2.0
```

add_dimension(*dimension*)
Add the specified dimension to this patch

Input

- *dimension* - (Dimension) Dimension to be added

add_gauges(*gauge_coords*)
Determine the cell indices of each gauge and make a list of all gauges with their cell indices.

c_centers_with_ghost(*num_ghost*)
(list of ndarray(...)) - List containing the arrays locating the computational locations of cell centers, see compute_c_centers() for more info.

c_edges_with_ghost(*num_ghost*)
(list of ndarray(...)) - List containing the arrays locating the computational locations of cell edges, see compute_c_edges() for more info.

compute_c_centers(*recompute=False*)
Calculate the c_centers array

This array is computed only when requested and then stored for later use unless the recompute flag is set to True.

Access the resulting computational coodinate array via the corresponding dimensions or via the computational grid properties c_centers.

Input

- *recompute* - (bool) Whether to force a recompute of the arrays

compute_c_centers_with_ghost(*num_ghost*, *recompute=False*)
Calculate the c_centers_with_ghost array

This array is computed only when requested and then stored for later use unless the recompute flag is set to True.

Access the resulting computational coodinate array via the corresponding dimensions or via the computational grid properties c_centers_with_ghost.

Input

- *recompute* - (bool) Whether to force a recompute of the arrays

compute_c_edges(*recompute=False*)
Calculate the `c_edges` array

This array is computed only when requested and then stored for later use unless the recompute flag is set to True.

Access the resulting computational coodinate array via the corresponding dimensions or via the computational grid properties `c_edges`.

Input

- *recompute* - (bool) Whether to force a recompute of the arrays

compute_c_edges_with_ghost(*num_ghost*, *recompute=False*)
Calculate the `c_centers_with_ghost` array

This array is computed only when requested and then stored for later use unless the recompute flag is set to True.

Access the resulting computational coodinate array via the corresponding dimensions or via the computational grid properties `c_centers_with_ghost`.

Input

- *recompute* - (bool) Whether to force a recompute of the arrays

compute_p_centers(*recompute=False*)
Calculates the `p_centers` array, which contains the physical coordinates of the cell centers when a mapping is used.

grid._p_centers is a list of numpy arrays. Each array has shape equal to the shape of the grid; the number of arrays is equal to the dimension of the embedding space for the mapping.

This array is computed only when requested and then stored for later use unless the recompute flag is set to True (you may want to do this for time-dependent mappings).

Access the resulting physical coordinate array via the corresponding dimensions or via the computational grid properties `p_centers`.

Input

- *recompute* - (bool) Whether to force a recompute of the arrays

compute_p_edges(*recompute=False*)
Calculates the `p_edges` array

This array is computed only when requested and then stored for later use unless the recompute flag is set to True (you may want to do this for time dependent mappings).

Access the resulting physical coordinate array via the corresponding dimensions or via the computational grid properties `p_edges`.

Input

- *recompute* - (bool) Whether to force a recompute of the arrays

get_dim_attribute(*attr*)
Returns a tuple of all dimensions' attribute attr

setup_gauge_files(*outdir*)
Creates and opens file objects for gauges.

c_centers
(list of ndarray(...)) - List containing the arrays locating the computational locations of cell centers, see `compute_c_centers()` for more info.

c_edges
(list of ndarray(...)) - List containing the arrays locating the computational locations of cell edges, see `compute_c_edges()` for more info.

dimensions
(list) - List of `Dimension` objects defining the grid's extent and resolution

gauge_dir_name = None
(string) - Name of the output directory for gauges. If the *Controller* class is used to run the application, this directory by default will be created under the *Controller outdir* directory.

gauge_file_names = None
(list) - List of file names to write gauge values to

gauge_files = None
(list) - List of file objects to write gauge values to

gauges = None
(list) - List of gauges' indices to be filled by add_gauges method.

mapc2p = None
(func) - Coordinate mapping function

num_dim
(int) - Number of dimensions

p_centers
(list of ndarray(...)) - List containing the arrays locating the physical locations of cell centers, see `compute_p_centers()` for more info.

p_edges
(list of ndarray(...)) - List containing the arrays locating the physical locations of cell edges, see `compute_p_edges()` for more info.

pyclaw.geometry.Dimension

class pyclaw.geometry.**Dimension**(**args*, ***kargs*)
Basic class representing a dimension of a Patch object

Initialization

Input:

- *name* - (string) string Name of dimension
- *lower* - (float) Lower extent of dimension
- *upper* - (float) Upper extent of dimension
- *n* - (int) Number of cells
- *units* - (string) Type of units, used for informational purposes only

Output:

- (`Dimension`) - Initialized Dimension object

Example:

```
>>> from clawpack.pyclaw.geometry import Dimension
>>> x = Dimension('x',0.,1.,100)
>>> print x
Dimension x:  (num_cells,delta,[lower,upper]) = (100,0.01,[0.0,1.0])
>>> x.name
'x'
>>> x.num_cells
100
>>> x.delta
0.01
>>> x.edges[0]
0.0
>>> x.edges[1]
0.01
>>> x.edges[-1]
1.0
>>> x.centers[-1]
0.995
>>> len(x.centers)
100
>>> len(x.edges)
101
```

centers
(ndarrary(:)) - Location of all cell center coordinates for this dimension

centers_with_ghost
(ndarrary(:)) - Location of all cell center coordinates for this dimension, including centers of ghost cells.

delta
(float) - Size of an individual, computational cell

edges
(ndarrary(:)) - Location of all cell edge coordinates for this dimension

lower = None
(float) - Lower computational dimension extent

name = None
(string) Name of this coordinate dimension (e.g. 'x')

num_cells = None
(int) - Number of cells in this dimension `units`

on_lower_boundary = None
(bool) - Whether the dimension is crossing a lower boundary.

on_upper_boundary = None
(bool) - Whether the dimension is crossing an upper boundary.

units = None
(string) Corresponding physical units of this dimension (e.g. 'm/s'), `default = None`

upper = None
(float) - Upper computational dimension extent

Parallel Geometry Objects

petclaw.geometry.Domain

class petclaw.geometry.**Domain**(*geom*)

Bases: clawpack.pyclaw.geometry.Domain

Parallel Domain Class

Unexpected section title.

```
Parent Class Documentation
==========================
```

2D Classic (Clawpack) solver.

Solve using the wave propagation algorithms of Randy LeVeque's Clawpack code (www.clawpack.org).

In addition to the attributes of ClawSolver1D, ClawSolver2D also has the following options:

dimensional_split
If True, use dimensional splitting (Godunov splitting). Dimensional splitting with Strang splitting is not supported at present but could easily be enabled if necessary. If False, use unsplit Clawpack algorithms, possibly including transverse Riemann solves.

transverse_waves
If dimensional_split is True, this option has no effect. If dimensional_split is False, then transverse_waves should be one of the following values:

ClawSolver2D.no_trans: Transverse Riemann solver not used. The stable CFL for this algorithm is 0.5. Not recommended.

ClawSolver2D.trans_inc: Transverse increment waves are computed and propagated.

ClawSolver2D.trans_cor: Transverse increment waves and transverse correction waves are computed and propagated.

Note that only the fortran routines are supported for now in 2D.

Unexpected section title.

```
Parent Class Documentation
==========================
```

Generic classic Clawpack solver

All Clawpack solvers inherit from this base class.

mthlim
Limiter(s) to be used. Specified either as one value or a list. If one value, the specified limiter is used for all wave families. If a list, the specified values indicate which limiter to apply to each wave family. Take a look at pyclaw.limiters.tvd for an enumeration. `Default = limiters.tvd.minmod`

order
Order of the solver, either 1 for first order (i.e., Godunov's method) or 2 for second order (Lax-Wendroff-LeVeque). `Default = 2`

source_split
Which source splitting method to use: 1 for first order Godunov splitting and 2 for second order Strang splitting. `Default = 1`

fwave
Whether to split the flux jump (rather than the jump in Q) into waves; requires that the Riemann solver performs the splitting. `Default = False`

step_source
Handle for function that evaluates the source term. The required signature for this function is:

def step_source(solver,state,dt)

before_step
Function called before each time step is taken. The required signature for this function is:

def before_step(solver,solution)

kernel_language
Specifies whether to use wrapped Fortran routines ('Fortran') or pure Python ('Python'). `Default = 'Fortran'`.

verbosity
The level of detail of logged messages from the Fortran solver. `Default = 0`.

petclaw.geometry.Patch

class petclaw.geometry.**Patch**(*dimensions*)
Bases: clawpack.pyclaw.geometry.Patch

Parallel Patch class.

Unexpected section title.

```
Parent Class Documentation
==========================
```

Global Patch information Each patch has a value for `level` and `patch_index`.

4.3.8 Pyclaw Utility Module

pyclaw.util

Pyclaw utility methods

class pyclaw.util.**FrameCounter**
Simple frame counter

Simple frame counter to keep track of current frame number. This can also be used to keep multiple runs frames seperated by having multiple counters at once.

Initializes to 0

get_counter()
Get the current frame number

increment()
Increment the counter by one

reset_counter()
Reset the counter to 0

set_counter(*new_frame_num*)
Set the counter to new_frame_num

pyclaw.util.**add_parent_doc**(*parent*)
add parent documentation for a class

pyclaw.util.**check_diff**(*expected*, *test*, ***kwargs*)
Checks the difference between expected and test values, return None if ok

This function expects either the keyword argument 'abstol' or 'reltol'.

pyclaw.util.**compile_library**(*source_list*, *module_name*, *interface_functions=[]*, *local_path='./'*, *library_path='./'*, *f2py_flags=''*, *FC=None*, *FFLAGS=None*, *recompile=False*, *clean=False*)
Compiles and wraps fortran source into a callable module in python.

This function uses f2py to create an interface from python to the fortran sources in source_list. The source_list can either be a list of names of source files in which case compile_library will search for the file in local_path and then in library_path. If a path is given, the file will be checked to see if it exists, if not it will look for the file in the above resolution order. If any source file is not found, an IOException is raised.

The list interface_functions allows the user to specify which fortran functions are actually available to python. The interface functions are assumed to be in the file with their name, i.e. claw1 is located in 'claw1.f95' or 'claw1.f'.

The interface from fortran may be different than the original function call in fortran so the user should make sure to check the automatically created doc string for the fortran module for proper use.

Source files will not be recompiled if they have not been changed.

One set of options of note is for enabling OpenMP, it requires the usual fortran flags but the OpenMP library also must be compiled in, this is done with the flag -lgomp. The call to compile_library would then be:

compile_library(src,module_name,f2py_flags='-lgomp',FFLAGS='-fopenmp')

For complete optimization use:

FFLAGS='-O3 -fopenmp -funroll-loops -finline-functions -fdefault-real-8'

Input

- *source_list* - (list of strings) List of source files, if these are just names of the source files, i.e. 'bc1.f' then they will be searched for in the default source resolution order, if an explicit path is given, i.e. './bc1.f', then the function will use that source if it can find it.
- *module_name* - (string) Name of the resulting module
- *interface_functions* - (list of strings) List of function names to provide access to, if empty, all functions are accessible to python. Defaults to [].
- *local_path* - (string) The base path for source resolution, defaults to './'.
- *library_path* - (string) The library path for source resolution, defaults to './'.
- *f2py_flags* - (string) f2py flags to be passed
- *FC* - (string) Override the environment variable FC and use it to compile, note that this does not replace the compiler that f2py uses, only the object file compilation (functions that do not have interfaces)
- *FFLAGS* - (string) Override the environment variable FFLAGS and pass them to the fortran compiler
- *recompile* - (bool) Force recompilation of the library, defaults to False
- *clean* - (bool) Force a clean build of all source files

pyclaw.util.**construct_function_handle**(*path*, *function_name=None*)
Constructs a function handle from the file at path.

This function will attempt to construct a function handle from the python file at path.

Input

- *path* - (string) Path to the file containing the function
- *function_name* - (string) Name of the function defined in the file that the handle will point to. Defaults to the same name as the file without the extension.

Output

- (func) Function handle to the constructed function, None if this has failed.

pyclaw.util.**convert_fort_double_to_float**(*number*)
Converts a fortran format double to a float

Converts a fortran format double to a python float.

number: is a string representation of the double. Number should be of the form "1.0d0"

pyclaw.util.**gen_variants**(*application*, *verifier*, *kernel_languages=('Fortran',)*, ***kwargs*)
Generator of runnable variants of a test application given a verifier

Given an application, a script for verifying its output, and a list of kernel languages to try, generates all possible variants of the application to try by taking a product of the available kernel_languages and (petclaw/pyclaw). For many applications, this will generate 4 variants: the product of the two main kernel languages ('Fortran' and 'Python'), against the the two parallel modes (petclaw and pyclaw).

For more information on how the verifier function should be implemented, see util.test_app for a description, and util.check_diff for an example.

All unrecognized keyword arguments are passed through to the application.

pyclaw.util.**read_data_line**(*inputfile*, *num_entries=1*, *data_type=<type 'float'>*)
Read data a single line from an input file

Reads one line from an input file and returns an array of values

inputfile: a file pointer to an open file object num_entries: number of entries that should be read, defaults to only 1 type: Type of the values to be read in, they all must be the same type

This function will return either a single value or an array of values depending on if num_entries > 1

pyclaw.util.**run_app_from_main**(*application*, *setplot=None*)
Runs an application from pyclaw/examples/, automatically parsing command line keyword arguments (key=value) as parameters to the application, with positional arguments being passed to PETSc (if it is enabled).

Perhaps we should take the PETSc approach of having a database of PyClaw options that can be queried for options on specific objects within the PyClaw runtime instead of front-loading everything through the application main...

pyclaw.util.**run_serialized**(*fun*)
Decorates a function to only run serially, even if called in parallel.

In a parallel communicator, the first process will run while the remaining processes block on a barrier. In a serial run, the function will be called directly.

This currently assumes the global communicator is PETSc.COMM_WORLD, but is easily generalized.

pyclaw.util.**test_app**(*application*, *verifier*, *kwargs*)
Test the output of a given application against its verifier method.

This function performs the following two function calls:

```
output = application(**kwargs)
check_values = verifier(output)
```

The verifier method should return None if the output is correct, otherwise it should return an indexed sequence of three items:

```
0 - expected value
1 - test value
2 - string describing the tolerance type (abs/rel) and value.
```

This information is used to present descriptive help if an error is detected. For an example verifier method, see util.check_diff

4.4 Riemann Solvers reference documentation

The Riemann solvers now comprise a separate package. For convenience, documentation of the available pure python Riemann solvers is included here. Many other Fortran-based Riemann solvers are available.

4.4.1 Riemann Solver Package

This package contains all of the Python-based Riemann solvers. Each module solves the Riemann solver for a particular system of hyperbolic equations. The solvers all have a common function signature:

`rp_<name>_<dim>d(q_l,q_r,aux_l,aux_r,problem_data)`

with `<name>` replaced with the appropriate solver name and `<dim>` with the appropriate dimension.

Input

- *q_l* - (ndarray(...,num_eqn)) Contains the left states of the Riemann problem
- *q_r* - (ndarray(...,num_eqn)) Contains the right states of the Riemann problem
- *aux_l* - (ndarray(...,num_aux)) Contains the left values of the auxiliary array
- *aux_r* - (ndarray(...,num_aux)) Contains the right values oft he auxiliary array
- ***problem_data* - (dict) Dictionary containing miscellaneous data which is** usually problem dependent.

Output

- ***wave* - (ndarray(...,num_eqn,num_waves)) Contains the resulting waves from the cell** edge
- *s* - (ndarray(...,num_waves)) Speeds of each wave
- *amdq* - (ndarray(...,num_eqn)) Left going fluctuation
- *apdq* - (ndarray(...,num_eqn)) Right going fluctuation

Except for *problem_data*, all of the input and output values are arrays whose elements represent grid values with locations indicated by the following scheme

```
Indexing works like this:  here num_ghost=2 as an example
 0     1     2     3     4     mx+num_ghost-2     mx+num_ghost      mx+num_ghost+2
             |                          mx+num_ghost-1 |  mx+num_ghost+1
 |     |     |     |     |   ...   |     |     |     |     |
    0     1  |  2     3            mx+num_ghost-2    |mx+num_ghost
```

```
                                              mx+num_ghost-1   mx+num_ghost+1

The top indices represent the values that are located on the grid
cell boundaries such as waves, s and other Riemann problem values,
the bottom for the cell centered values.  In particular the ith grid cell
boundary has the following related information:
                  i-1          i          i+1
                   |           |           |
                   |   i-1     |     i     |
                   |           |           |
Again, grid cell boundary quantities are at the top, cell centered
values are in the cell.
```

Note: The values `q_l[i]`, `q_r[i]` are the left and right states, respectively, of the `ith` Riemann problem. This convention is different than that used in the Fortran Riemann solvers, where `q_l[i]`, `q_r[i]` are the values at the left and right edges of a cell.

All of the return values (waves, speeds, and fluctuations) are indexed by cell edge (Riemann problem being solved), with `s[i]` referring to the wave speed at interface $i-1/2$. This follows the same convention used in the Fortran solvers.

See [LeVeque_book_2002] for more details.

List of available Riemann solvers:

Acoustics

Riemann solvers for constant coefficient acoustics.

$$q_t + Aq_x = 0$$

where

$$q(x,t) = \begin{bmatrix} p(x,t) \\ u(x,t) \end{bmatrix}$$

and the coefficient matrix is

$$A = \begin{bmatrix} 0 & K \\ 1/\rho & 0 \end{bmatrix}$$

The parameters $\rho =$ density and $K =$ bulk modulus are used to calculate the impedence $= Z$ and speed of sound = c.

Authors Kyle T. Mandli (2009-02-03): Initial version

`clawpack.riemann.acoustics_1D_py.`**`acoustics_1D`**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)

Basic 1d acoustics riemann solver, with interleaved arrays

***problem_data* is expected to contain -**

- *zz* - (float) Impedence

- *cc* - (float) Speed of sound

See *Riemann Solver Package* (page 240) for more details.

Version 1.0 (2009-02-03)

Advection

Simple advection Riemann solvers

Basic advection Riemann solvers of the form (1d)

$$q_t + Aq_x = 0.$$

Authors Kyle T. Mandli (2008-2-20): Initial version

clawpack.riemann.advection_1D_py.**advection_1D**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)
Basic 1d advection riemann solver

***problem_data* should contain -**

- *u* - (float) Determines advection speed

See *Riemann Solver Package* (page 240) for more details.

Version 1.0 (2008-2-20)

Burgers Equation

Riemann solvers for Burgers equation

$$u_t + \left(\frac{1}{2}u^2\right)_x = 0$$

Authors Kyle T. Mandli (2009-2-4): Initial version

clawpack.riemann.burgers_1D_py.**burgers_1D**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)
Riemann solver for Burgers equation in 1d

***problem_data* should contain -**

- *efix* - (bool) Whether a entropy fix should be used, if not present, false is assumed

See *Riemann Solver Package* (page 240) for more details.

Version 1.0 (2009-2-4)

Euler Equations

Riemann solvers for the Euler equations

This module contains Riemann solvers for the Euler equations which have the form (in 1d):

$$q_t + f(q)_x = 0$$

where

$$q(x,t) = \begin{bmatrix} \rho \\ \rho u \\ E \end{bmatrix},$$

the flux function is

$$f(q) = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ u(E+p) \end{bmatrix}.$$

and ρ is the density, u the velocity, E is the energy and p is the pressure.

Unless otherwise noted, the ideal gas equation of state is used:

$$E = (\gamma - 1)\left(E - \frac{1}{2}\rho u^2\right)$$

Authors Kyle T. Mandli (2009-06-26): Initial version Kyle T. Mandli (2011-03-28): Interleaved version

clawpack.riemann.euler_1D_py.**euler_exact_1D**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)
Exact euler Riemann solver

Warning: This solver has not been implemented.

clawpack.riemann.euler_1D_py.**euler_hll_1D**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)
HLL euler solver

```
W_1 = Q_hat - Q_l     s_1 = min(u_l-c_l,u_l+c_l,lambda_roe_1,lambda_roe_2)
W_2 = Q_r - Q_hat     s_2 = max(u_r-c_r,u_r+c_r,lambda_roe_1,lambda_roe_2)

Q_hat = ( f(q_r) - f(q_l) - s_2 * q_r + s_1 * q_l ) / (s_1 - s_2)
```

***problem_data* should contain:**

- *gamma* - (float) Ratio of the heat capacities
- *gamma1* - (float) $1 - \gamma$

Version 1.0 (2014-03-04)

clawpack.riemann.euler_1D_py.**euler_roe_1D**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)
Roe Euler solver in 1d

***aug_global* should contain -**

- *gamma* - (float) Ratio of the heat capacities
- *gamma1* - (float) $1 - \gamma$
- *efix* - (bool) Whether to use an entropy fix or not

See *Riemann Solver Package* (page 240) for more details.

Version 1.0 (2009-6-26)

Shallow Water Equations

Riemann solvers for the shallow water equations.

The available solvers are:

- Roe - Use Roe averages to caluclate the solution to the Riemann problem
- HLL - Use a HLL solver
- **Exact - Use a newton iteration to calculate the exact solution to the** Riemann problem

$$q_t + f(q)_x = 0$$

where

$$q(x,t) = \begin{bmatrix} h \\ hu \end{bmatrix},$$

the flux function is

$$f(q) = \begin{bmatrix} hu \\ hu^2 + 1/2gh^2 \end{bmatrix}.$$

and h is the water column height, u the velocity and g is the gravitational acceleration.

Authors Kyle T. Mandli (2009-02-05): Initial version

clawpack.riemann.shallow_1D_py.**shallow_exact_1D**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)
Exact shallow water Riemann solver

Warning: This solver has not been implemented.

clawpack.riemann.shallow_1D_py.**shallow_fwave_1d**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)
Shallow water Riemann solver using fwaves

Also includes support for bathymetry but be wary if you think you might have dry states as this has not been tested.

***problem_data* should contain:**

- *grav* - (float) Gravitational constant
- *sea_level* - (float) Datum from which the dry-state is calculated.

Version 1.0 (2014-09-05)

clawpack.riemann.shallow_1D_py.**shallow_hll_1D**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)
HLL shallow water solver

```
W_1 = Q_hat - Q_l     s_1 = min(u_l-c_l,u_l+c_l,lambda_roe_1,lambda_roe_2)
W_2 = Q_r - Q_hat     s_2 = max(u_r-c_r,u_r+c_r,lambda_roe_1,lambda_roe_2)

Q_hat = ( f(q_r) - f(q_l) - s_2 * q_r + s_1 * q_l ) / (s_1 - s_2)
```

***problem_data* should contain:**

- *g* - (float) Gravitational constant

Version 1.0 (2009-02-05)

clawpack.riemann.shallow_1D_py.**shallow_roe_1D**(*q_l*, *q_r*, *aux_l*, *aux_r*, *problem_data*)
Roe shallow water solver in 1d:

```
ubar = (sqrt(u_l) + sqrt(u_r)) / (sqrt(h_l) + sqrt(h_r))
cbar = sqrt( 0.5 * g * (h_l + h_r))

W_1 = |      1      |  s_1 = ubar - cbar
      | ubar - cbar |

W_2 = |      1      |  s_1 = ubar + cbar
      | ubar + cbar |
```

```
a1 = 0.5 * ( - delta_hu + (ubar + cbar) * delta_h ) / cbar
a2 = 0.5 * (   delta_hu - (ubar - cbar) * delta_h ) / cbar
```

***problem_data* should contain:**

- *g* - (float) Gravitational constant
- *efix* - (bool) Boolean as to whether a entropy fix should be used, if not present, false is assumed

Version 1.0 (2009-02-05)

4.5 Indices and tables

- *genindex*
- *modindex*
- *search*

4.6 Citing

If you use PyClaw in work that will be published, please cite the Clawpack software:

```
@misc{clawpack,
    title={Clawpack software},
    author={Clawpack Development Team},
    url={http://www.clawpack.org},
    note={Version x.y},
    year={2014}}
```

and the paper:

```
@article{pyclaw-sisc,
        Author = {Ketcheson, David I. and Mandli, Kyle T. and Ahmadia, Aron J. and Alghamdi, Amal and
        Journal = {SIAM Journal on Scientific Computing},
        Month = nov,
        Number = {4},
        Pages = {C210--C231},
        Title = {{PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems}},
        Volume = {34},
        Year = {2012}}
```

Please fill in the version number that you used.

If you use the Classic (2nd-order) solver, you may also wish to cite:

```
@article{leveque1997,
        Author = {LeVeque, Randall J.},
        Journal = {Journal of Computational Physics},
        Pages = {327--353},
        Title = {{Wave Propagation Algorithms for Multidimensional Hyperbolic Systems}},
        Volume = {131},
        Year = {1997}}
```

If you use the SharpClaw (high order WENO) solver, you may also wish to cite:

```
@article{KetParLev13,
        Author = {Ketcheson, David I. and Parsani, Matteo and LeVeque,
        Randall J.},
        Journal = {SIAM Journal on Scientific Computing},
        Number = {1},
        Pages = {A351--A377},
        Title = {{High-order Wave Propagation Algorithms for Hyperbolic Systems}},
        Volume = {35},
        Year = {2013}}
```

CHAPTER

FIVE

VISCLAW: PLOTTING AND VISUALIZATION TOOLS

5.1 Visclaw Plotting options

5.1.1 Plotting as post-processing

Running a Fortran version of Clawpack produces output files that can then be read in to a graphics package as a post-processing step. If you understand the format of the output files, you can write custom graphics routines using your favorite visualization tools. The output formats are described in the section *Output data formats* (page 67).

Clawpack includes utilities for plotting using Python, and most of the documention assumes you will use these tools. See *Plotting options in Python* (page 248) for a description of these. The Python package matplotlib is used under the hood for 1d and 2d plots, but the tools provided with Clawpack simplify some common tasks since they handle looping over all grid patches as is generally required when plotting AMR data.

Matlab plotting tools (mostly the same as in Clawpack 4.x) are available in Visclaw. See *Plotting using Matlab* (page 268) for pointers if you wish to use these tools. For 3d plots the Matlab tools may still be the best choice.

Another alternative for 3d plots (also for 2d) is to use VisIt (https://wci.llnl.gov/codes/visit/home.html). See *Plotting with VisIt* (page 273).

Since Clawpack 4.4, a set of Python plotting tools for 1d and 2d are the recommended approach. The advantages of using the Python options are:

- Python and the graphics modules used in Clawpack are open source. Since Clawpack itself is open source we find it desirable to also have an open source plotting open for viewing the results.
- The Python tools developed so far (mostly for 1d and 2d data sets) are more powerful than the Matlab versions they replace, and can be used for example to automatically generate html versions of multiple plots each frame over all frames of a computation, to generate latex versions of the output, as well as to step through the frames one by one as with the Matlab tools. It is easier to specify a set of multiple plots to be produced for each frame.
- Matlab graphics are somewhat limited for 3d data sets, whereas several open source visualization tools such as *VisIt* are much better for dealing with large data sets, AMR meshes, etc. and have Python bindings that should allow scripting in a manner compatible with 1d and 2d. (Yet to be done.)
- Python is a powerful language that can be scripted to perform multiple runs, such as in a convergence test, and collect the results in tables or plots. We are developing tools to simplify this process.

5.1.2 Plotting on the fly

Describe options.

5.2 Plotting options in Python

Clawpack includes utilities for plotting using Python. Most of these are defined in the *clawpack.visclaw* module. In order to use these you will need to insure that you have the required modules installed (see *Installation of required modules* (page 52)).

See *Python Hints* (page 52) for more information on Python and pointers to many tutorials.

5.2.1 Producing html plots from the command line

In most Clawpack directories, typing:

```
$  make .plots
```

will compile the code and run it (if necessary) and then produce a set of html files that can be used to view the resulting plots. These will be in a subdirectory of the current directory as specified by PLOTDIR in the Makefile.

5.2.2 Producing a latex file with plots from the command line

A latex file with all the plots can also be produced by *printframes* (page 250), and is also typically controlled by options set in the file setplot.py.

5.2.3 Setting plot parameters with a setplot function

Typically each applications directory contains a file `setplot.py`, see for example *Plotting examples* (page 262). This file should define a function *setplot(plotdata)* that sets various attributes of an object plotdata of class *ClawPlotData* (page 251).

Documentation on how to create a setplot function and the various plotting parameters that can be set can be found in the section *Using setplot.py to specify the desired plots* (page 251).

Examples can be found at *Plotting examples* (page 262).

5.2.4 Interactive plotting with *Iplotclaw*

For interactive plotting we suggest using IPython (http://ipython.org), which is a nicer shell than the pure python shell, with things like command completion and history.

Here's an example:

```
$  ipython
In [1]: from clawpack.visclaw.Iplotclaw import Iplotclaw
In [2]: ip = Iplotclaw()
In [3]: ip.plotloop()
Executing setplot ...

Interactive plotclaw with ndim = 1 ...
Type ? at PLOTCLAW prompt for list of commands

    Start at which frame [default=0] ?
    Plotting frame 0 ...
PLOTCLAW >  n
    Plotting frame 1 ...
PLOTCLAW > q
```

```
quitting...
In [4]:
```

Type *?* at the PLOTCLAW prompt or *?command-name* for more information. Most commonly used are n for next frame, p for previous frame and j to jump to a different frame. Hitting return at the prompt repeats the previous command.

You can restart the plotloop later by doing:

```
In [4]: ip.plotloop()

Interactive plotclaw with ndim = 1 ...
Type ? at PLOTCLAW prompt for list of commands

    Start at which frame [default=1] ?
    Replot data for frame 1 [no] ?
PLOTCLAW >
```

By default it starts at the frame where you previously left off.

If you want to change plot parameters, the easiest way is to edit the file setplot.py, either in a different window or, if you use vi, by:

```
PLOTCLAW > vi setplot.py
```

and then re-execute the setplot function using:

```
PLOTCLAW > resetplot
```

If you recompute results by running the fortran code again and want to plot the new results (from this same directory), you may have to clear the frames that have already been viewed using:

```
PLOTCLAW > clearframes
```

Or you can redraw the frame you're currently looking at without clearing the rest of the cached frame data by doing:

```
PLOTCLAW > rr
```

To see what figures, axes, and items have been defined by *setplot*:

```
PLOTCLAW > show

Current plot figures, axes, and items:
----------------------------------------
  figname = Pressure, figno = 1
     axesname = AXES1, axescmd = subplot(1,1,1)
        itemname = ITEM1,  plot_type = 1d_plot

  figname = Velocity, figno = 2
     axesname = AXES1, axescmd = subplot(1,1,1)
        itemname = ITEM1,  plot_type = 1d_plot
```

Type "help" or "help command-name" at the prompt for more options.

Access to current_data

If you are viewing plots in using Iplotclaw and want to explore the data for some frame or make plots directly in your Python shell, the data that is being plotted is available to you in attributes of the Iplotclaw instance. For example:

```
>>> ip = Iplotclaw();  ip.plotloop()

Interactive plotting for Clawpack output...

Plotting data from outdir =  _output
    ...
    Plotting Frame 0 at t = 0.0
PLOTCLAW > q
quitting...

>>> pd = ip.plotdata
>>> cd = ip.current_data
```

The *cd* object contains the *current_data* (page 261) used for the most recent plot, while *pd* is the *ClawPlotData* (page 251) object that gives access to all the plotting parameters currently being used as well as to methods such as *getframe* for retrieving other frames of data from this computation.

If you want to change the directory *outdir* where the frame data is coming from, you could do, for example:

```
>>> pd.outdir = "_output2"
>>> ip.plotloop()
...
PLOTCLAW > clearframes    # to remove old frames from cache
PLOTCLAW > rr             # to redraw current frame number but with new data
```

5.2.5 printframes

Need to update

The function pyclaw.plotters.frametools.printframes can be used to produce html and latex versions of the plots:

```
>>> from clawpack.visclaw.data import ClawPlotData
>>> from clawpack.visclaw import frametools
>>> plotclaw = ClawPlotData()
>>> # set attributes as desired
>>> frametools.printframes(plotclaw)
```

A convenience method of ClawPlotData is defined to apply this function, e.g.:

```
>>> plotclaw.printframes()
```

This function is automatically called by the "make .plots" option available in most examples.

5.2.6 Specifying what and how to plot

The first step in specifying how to plot is to create a *ClawPlotData* (page 251) object to hold all the data required for plotting. This is generally done by creating a file *setplot.py* (see below).

Note that when you use Iplotclaw to do interactive plotting, e.g.:

```
>>> ip = Iplotclaw()
```

Then object *ip* will have an attribute plotdata that is a *ClawPlotData* (page 251) object. This object will have attribute setplot initialized to 'setplot.py', indicating that other attributes should be set by executing the setplot function defined in the file 'setplot.py' in this directory.

Once you have a *ClawPlotData* (page 251) object you can set various attributes to control what is plotted interactively if you want. For example,:

```
>>> plotdata.plotdir = '_plots'
>>> plotdata.setplot = 'my_setplot_file.py'
```

will cause hardcopy to go to subdirectory _plots of the current directory and will cause the plotting routines to execute:

```
>>> from my_setplot_file import setplot
>>> plotdata = setplot(plotdata)
```

before doing the plotting.

There are many other *ClawPlotData* (page 251) attributes and methods.

Most example directories contain a file setplot.py that contains a function setplot(). This function sets various attributes of the *ClawPlotData* (page 251) object to control what figures, axes, and items should be plotted for each frame of the solution.

For an outline of how a typical set of plots is specified, see *Using setplot.py to specify the desired plots* (page 251).

5.3 Using setplot.py to specify the desired plots

The desired plots are specified by creating an object of class ClawPlotData that contains specifications of what *figures* should be created, and within each figure what sets of *axes* should be drawn, and within each axes what *items* should be plotted (lines, contour plots, etc.).

5.3.1 Plotting Data Objects

More details about each class of objects can be found on these pages:

ClawPlotData

For usage see *Visclaw Plotting options* (page 247), *Using setplot.py to specify the desired plots* (page 251), and *Plotting examples* (page 262).

class ClawPlotData

Attributes

outdir : str
Path to the directory where the Clawpack output is that is to be plotted.

plotdir : str
Path to the directory where hardcopy files of plots are to be put.

overwrite : bool
Ok to overwrite old plotdir? Default is True

afterframe : str or function
Function or string to be executed after producing all plots for each frame. If a string, this string is executed using *exec*. If a function, it should be defined to have a single argument "data", [documentation to appear!] For example:

```
def afterframe(data):
    t = data.t
    print "Done plotting at time %s" % t
```

beforeframe : str or function
Function or string to be executed before starting to do plots for each frame. If a function, it should be defined to have a single argument "data", [documentation to appear!]

```
def beforeframe(data):
```

printfigs : bool
True if plots are to be generated and printed (e.g. to png files) before making html or latex files.

False if the plots have already been generated and the existing versions are to be used for making html or latex files.

print_format : str
format for hardcopy, default is 'png'

print_framenos : list of int's or 'all'
which frames to print, default is 'all'

print_fignos : list of int's or 'all'
which figures to print for each frame, default is 'all'

iplotclaw_fignos : list of int's or 'all'
which figures to print for each frame in interactive mode, default is 'all'

latex : bool
If True, create a latex file in directory plotdir that to display all the plots.

latex_fname : str
Name of latex file, default is 'plots'. The file created will be e.g. plots.tex.

latex_title : str
The title to go at the top of the latex file, default is "Clawpack Results"

latex_framesperpage : int or 'all'
How many frames to try to put on each page. Default is 'all'.

latex_framesperline : int or 'all'
How many frames to try to put on each line. Default is 'all'.

latex_figsperline : int or 'all'
How many figures to try to put on each line. Default is 'all'. Recall that several plots may be generated for each frame.

latex_pdf : bool
If True, run pdflatex on the latex file to create e.g., plots.pdf.

html : bool
If True, create a set of html files to display the plots and an index to these files called _PlotIndex.html. These will be in the directory specified by the plotdir attribute.

Methods

new_plotfigure(*name*, *figno*)
Create and return a new object of class *ClawPlotFigure* (page 253) associated with this ClawPlotData object.

getframe(*frameno*, *outdir=None*)
Return an object of class *ClawSolution* that contains the solution read in from the fort.q000N file (where N is frameno).

If outdir==None then the outdir attribute of this ClawPlotData object is used as the directory to find the data.

The data, once read in, is stored in a dictionary (the attribute framesoln_dict of this ClawPlotData object). It is read from the fort.q file only if it is not already in the dictionary.

Note that frames read from different outdir's are stored separately (with dictionary key (frameno, outdir) if outdir != None).

clearframes (*framenos*)
Remove one or more frames from the dictionary framesoln_dict. (Different outdir's not yet implemented.)

clearfigures ()
Clear all plot parameters. Useful as the first command in a *setplot* function to make sure previous parameters are cleared if the file is changed and the function is re-executed in an interactive session.

iplotclaw ()
Return True if interactive plotting with iplotclaw is being done.

getfigure (*figname*)
Return *ClawPlotFigure* (page 253) object with the specified name.

getaxes (*axesname*, *figname=None*)
Return *ClawPlotAxes* (page 254) object with the specified name. If figname==None then search over all figures and return None if it is not found or the name is not unique.

getitem (*itemname*, *axesname=None*, *figname=None*)
Return *ClawPlotItem* (page 255) object with the specified name. If axesname==None and/or figname==None then search over all figures and/or axes and return None if it is not found or the name is not unique.

showitems ()
Print a list of all the figures, axes, and items defined.

plotframe (*frameno*)
Plot all figures for frame number frameno. Convenience method that calls pyclaw.plotters.frametools.plotframe().

printframes ()
Plot and print hardcopy for all frames. Convenience method that calls pyclaw.plotters.frametools.printframes().

Note: More methods still to be documented.

ClawPlotFigure

For usage see *Using setplot.py to specify the desired plots* (page 251) and *Plotting examples* (page 262).

Objects of this class are usually created by the new_plotfigure method of a *ClawPlotData* (page 251) object.

class ClawPlotFigure

Attributes

The following attributes can be set by the user:

figno : int
Figure number, by default the next unused number will be used (starting at 1001). This is usually set as an argument to the new_plotfigure function of a *ClawPlotData* (page 251) object

kwargs : dictionary
A dictionary of keyword arguments for the figure command. For example:

```
"{'figsize':[12,5], 'facecolor':[1,0,0]}"
```

would specify that the figure should be 12 inches by 5 inches with a red background.

For more options see the matplotlib documentation (http://matplotlib.sourceforge.net/) for the figure command (http://matplotlib.sourceforge.net/api/figure_api.html#matplotlib.figure.Figure).

clf_each_frame : bool
If True, clear the figure with a clf command at the start of each frame.

show : bool
If False, suppress showing this figure.

Methods

new_plotaxes(*name=None*)
Create and return a new object of class *ClawPlotAxes* (page 254) associated with this ClawPlotFigure object. A single figure may have several axes on it.

The name specified is used as a dictionary key. If None is provided, one is generated automatically of the form AXES1, etc.

gethandle()
Returns the handle for this figure.

ClawPlotAxes

For usage see *Using setplot.py to specify the desired plots* (page 251) and *Plotting examples* (page 262).

Objects of this class are usually created by the new_plotaxes method of a *ClawPlotFigure* (page 253) object.

See also:

Using setplot.py to specify the desired plots (page 251) and *Plotting examples* (page 262)

class ClawPlotAxes

Attributes

The following attributes can be set by the user:

name : str

title : str
The title to appear at the top of the axis. Defaults to string specified by the *name* attribute.

title_with_t : bool
If True, creates a title of the form "%s at time t = %s" % (title, t)

axescmd : str

The command to be used to create this axes, for example:

- "subplot(1,1,1)" for a single axes filling the figure
- "subplot(2,1,1)" for the top half
- "axes([0.1, 0.1, 0.2, 0.8])" for a tall skinny axis.

See the matplotlib documentation for axes.

xlimits : array [xmin, xmax] or 'auto'
The x-axis limits if an array with two elements, or choose automatically

ylimits : array [ymin, ymax] or 'auto'
The y-axis limits if an array with two elements, or choose automatically

afteraxes : function or str
A string or function that is to be executed after creating all plot items on this axes. If a string, this string is executed using *exec*. If a function, it should be defined to have a single argument *current_data* (page 261).

The string version is useful for 1-liners such as:

```
afteraxes = "pylab.title('My custom title')"
```

pylab commands can be used, since pylab has been imported into the plotting module.

The function form is better if you want to do several things, or if you need access to the data stored in *current_data* (page 261). For example:

```
def afteraxes(current_data):
    # add x- and y-axes to a 1d plot already created
    from pylab import plot

    xlower = current_data.xlower
    xupper = current_data.xupper
    plot([xlower, xupper], [0.,0.], 'k')    # x-axis

    # Get y limits from variable just plotted, which is
    # available in current_data.var.
    ymin = current_data.var.min()
    ymax = current_data.var.max()
    plot([0.,0.], [ymin,ymax], 'k')  # y-axis
```

show : bool
If False, suppress showing this axes and all items on it.

Methods

new_plotitem(*name=None*, *plot_type*)
Returns an object of class *ClawPlotItem* (page 255) associated with this axes. A single axes may have several items associated with it.

The name specified is used as a dictionary key. If None is provided, one is generated automatically of the form ITEM1, etc.

gethandle()
Returns the handle for this axes.

ClawPlotItem

For usage see *Using setplot.py to specify the desired plots* (page 251) and *Plotting examples* (page 262).

Objects of this class are usually created by the new_plotitem method of a *ClawPlotAxes* (page 254) object.

The examples shown below are fragments of a typical *setplot* function which assume *plotaxes* is an instance of *ClawPlotAxes* (page 254).

See also:

Using setplot.py to specify the desired plots (page 251) and *Plotting examples* (page 262)

See also:

Plotting examples (page 262)

class `ClawPlotItem`

Attributes

The following attributes can be set by the user:

`plot_type : str`
Type of plot desired, one of the following:

- '1d_plot' : one dimensional line or set of points plotted using the matplotlib plot command.
- '1d_from_2d_data' : 1d plot generated from 2d data, for example as a slice of the data or a scatter plot of data that should be radially symmetric,
- '1d_fill_between' : 1d filled plot between two variable specified by the attributes *plot_var* and *fill_var2*.
- '2d_contour' : two dimensional contour plot,
- '2d_pcolor' : two dimensional pcolor plot,
- '2d_schlieren' : two dimensional Schlieren plot,
- '2d_patch' : two dimensional plot of only the cell and/or patch edges, no data

`outdir : str or None`
Directory to find data to be plotted. If None, then data comes from the outdir attribute of the parent ClawPlotData item.

`plot_var : int or function`
If an integer, then this specifies which component of q to plot (with the Python convention that plot_var=0 corresponds to the first component).

If a function, then this function is applied to q on each patch to compute the variable var that is plotted. The signature is

- var = plot_var(current_data)

The *current_data* (page 261) object holds data from the current frame that can be used to compute the variable to be plotted.

`afteritem : str or function or None`
A string or function that is to be executed after plotting this item. If a string, this string is executed using *exec*. If a function, it should be defined to have a single argument *current_data* (page 261).

For example:

```
def afteritem(current_data):
```

`afterpatch : str or function or None`
A string or function that is to be executed after plotting this item on each patch. (There may be more than 1

patch in an AMR calculation.) If a string, this string is executed using *exec*. If a function, it should be defined to have a single argument "data", [documentation to appear!]

For example:

```
def afterpatch(current_data):
    cd = current_data
    print "On patch number %s, xlower = %s, ylower = %s" \
          % (cd.patchno, cd.xlower, cd.ylower)
```

would print out the patch number and lower left corner for each patch in a 2d computation after the patch is plotted.

MappedGrid : bool
If True, the mapping specified by the *mapc2p* attribute of the underlying *ClawPlotData* object should be applied to the patch before plotting.

show : bool
If False, plotting of this object is suppressed.

The other attributes required depend on the plot_type, as summarized below:

Special attributes for all 1d plots, plot_type = '1d...'
plotstyle : str
Anything that is valid as a fmt group in the matplotlib plot command (http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot). For example:

- '-' for a solid line, '- -' for a dashed line,
- 'o' for circles, 'x' for x's, '-o' for circles and a line,
- 'bo' for blue circles (though if the *color* attribute is also set that will overrule the color in the format string).

color : str
Any matplotlib color, for example red can be specified as 'r' or 'red' or '[1,0,0]' or '#ff0000'.

Special attributes for plot_type = '1d_plot' No extra attributes.

Special attributes for plot_type = '1d_fill_between' This gives a filled polygon between two curves using the matplotlib fill_between command (http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.fill_between).

plot_var : int or function
as described above defines one curve,

plot_var2 : int or function
defines the second curve for the fill_between command. Default is the zero function.

fill_where : str or None

defines the *where* attribute of the fill_between command.

Example:

```
plotitem = plotaxes.new_plotitem(plot_type='1d_fill_between')
plotitem.plot_var = 0     # means use q[:,0]
```

would produce a filled curve between y=q[:,0] and y=0.

Example:

```
plotitem = plotaxes.new_plotitem(plot_type='1d_fill_between')
plotitem.plot_var = 0    # means use q[:,0]
plotitem.plot_var2 = 1
```

would produce a filled curve between y=q[:,0] and y=q[:,1].

Special attributes for plot_type = '1d_from_2d_data'

map_2d_to_1d : function

Example: In a 2d computation where the solution q[:,:,0] should be radially symmetric about (x,y)=(0,0), the following will result in a scatter plot of the cell values q[i,j,0] vs. the radius r(i,j):

```
def q0_vs_radius(current_data):
    # convert 2d (x,y,q) into (r,q) for scatter plot
    from numpy import sqrt
    x = current_data.x
    y = current_data.y
    r = sqrt(x**2 + y**2)
    q0 = current_data.var   # the variable specified by plot_var
    # q0 = current_data.q[:,:,0]   # would also work
    return r,q0

plotitem = plotaxes.new_plotitem(plot_type='1d_from_2d_data')
plotitem.plot_var = 0     # use q[:,:,0]
plotitem.plotstyle = 'o'  # symbol not line is best for scatter plot
plotitem.map_2d_to_1d = q0_vs_radius   # the function defined above
```

See *current_data* (page 261) for a description of the *current_data* argument.

Special attributes for all 2d plots, plot_type = '2d...'

celledges_show : bool

If True, draw the cell edges on the plot. The attribute 'amr_celledges_show' should be used for AMR computations to specify that cell edges should be shown on some levels and not others. See *AMR Attributes* (page 259).

patchedges_show : bool

If True, draw the edges of patches, mostly useful in AMR computations.

Special attributes for plot_type = '2d_contour'

contour_levels : numpy array or None

If a numpy array, the contour levels. If None, then the next three attributes are used to set the levels.

contour_nlevels : int

Number of contour levels

contour_min : float

Minimum contour level

contour_max : float

Maximum contour level

contour_colors : color specification

Colors of contour lines. Can be a single color such as 'b' or '#0000ff', or a colormap.

amr_contour_colors : list of color specifications

As with other attributes (see *AMR Attributes* (page 259) below), instead of contour_colors you can specify *amr_contour_colors* to be a list of colors (or colormaps) to use on each AMR level, e.g.:

```
amr_contour_colors = ['k','b','r']
```

to use black lines on Level 1, blue on Level 2, and red for all subsequent levels. This is useful since with the matplotlib contour plotter you will see both fine and coarse cell edges on top of one another in refined regions (Matplotlib lacks the required hidden line removal to blank out the lines from coarser patches easily. See also the next attributes.)

contour_show : boolean
Show the contour lines only if this attribute is true. This is most commonly used in the form of the next attribute,

amr_contour_show : list or tuple of booleans
Determines whether to show the contour lines on each AMR level. Useful if you only want to view the lines on the finest patches.

contour_kwargs : dictionary
Other keyword arguments for the contour command.

Special attributes for plot_type = '2d_pcolor'
pcolor_cmap : matplotlib colormap

pcolor_cmin : float

pcolor_cmax : float
In general you should specify *pcolor_cmin* and *pcolor_cmax* to specify the range of q values over which the colormap applies. If they are not specified they will be chosen automatically and may vary from frame to frame. Also, if AMR is used, they may vary from patch to patch, yielding very confusing plots.

pcolor_colorbar : bool
If True, a colorbar is added to the plot.

AMR Attributes

Many attributes listed above also have a second related attribute with the same name pre-pended with *amr_*. If this attribute is set, it should be a list whose elements are of the type specified by the original name, and the elements of the list will be used for different AMR refinement levels.

For example, the following commands:

```
plotitem = plotaxes.new_plotitem(plot_type='2d_contour')
plotitem.contour_color = 'r'
```

will result in all contour lines being red on all levels of AMR. On the other hand:

```
plotitem = plotaxes.new_plotitem(plot_type='2d_contour')
plotitem.amr_contour_color = ['k', 'b']
```

will result in contour lines on patches at level 1 being black and on patches of level 2 or higher being blue.

Note that if the list is shorter than the number of levels, the last element is used repeatedly.

If both attributes *contour_color* and *amr_contour_color* are set, only *amr_contour_color* is used.

A common use is to show cell edges only on coarse levels, not on finer levels, e.g.:

```
plotitem.amr_celledges_show = [1,1,0]
```

will result in celledges being shown only on levels 1 and 2, not on finer levels.

Methods

getframe(*frameno*)
Returns an object of class *pyclaw.solution.Solution* (page 221) containing the solution being plotted by this object for frame number frameno.

gethandle()
Returns the handle for this item.

For examples, see *Plotting examples* (page 262).

5.3.2 Overview

The approach outlined below may seem more complicated than necessary, and it would be if all you ever want to do is plot one set of data at each output time. However, when adaptive mesh refinement is used each frame of data may contain several patches and so creating the desired plot requires looping over all patches. This is done by the plotting utilities described in *Visclaw Plotting options* (page 247), but for this to work it is necessary to specify what plot(s) are desired.

Most example directories contain a file setplot.py that contains a function setplot(plotdata). This function sets various attributes of plotdata in order to specify how plotting is to be done.

The object plotdata is of class ClawPlotData. The way to set up the plot structure is to follow this outline:

- Specify some attributes of setplot that determine what sort of plots will be produced and where they will be stored, e.g.:

  ```
  plotdata.plotdir = '_plots'
  ```

  will cause hardcopy to go to subdirectory _plots of the current directory. (Attributes like plotdir that are only used for hardcopy are often set in the script plotclaw.py rather than in setplot. See *Specifying what and how to plot* (page 250).)

  There are many other *ClawPlotData* (page 251) attributes and methods.

- Specify one or more Figures to be created for each frame, e.g.:

  ```
  plotfigure = plotdata.new_plotfigure(name='Solution', figno=1)
  ```

  plotfigure is now an object of class ClawPlotFigure and various attributes can be set, e.g.:

  ```
  plotfigure.kwargs = {'figsize':[8,12], 'facecolor':'#ff9999'}
  ```

  to specify any keyword arguments that should be used when creating this figure in matplotlib. The above would create a figure that is 8 inches by 12 inches with a pink background.

  For more options, see the matplotlib documentation (http://matplotlib.sourceforge.net/) for the figure command (http://matplotlib.sourceforge.net/api/figure_api.html#matplotlib.figure.Figure).

  There are many other *plotfigure* attributes and methods.

- Specify one or more Axes to be created within each figure, e.g.:

  ```
  plotaxes = plotfigure.new_plotaxes()
  ```

  Note that new_plotaxes is a method of class ClawPlotFigure and creates a set of axes specific to the particular object plotfigure.

  plotaxes is now an object of class ClawPlotAxes and various attributes can be set, e.g.:

```
plotfigure.ylimits = [-1, 1]
```

There are many other *ClawPlotAxes* (page 254) attributes and methods.

- Specify one or more Items to be created within these axes, e.g.:

```
plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
```

Note that new_plotitem is a method of class ClawPlotAxes and creates a plot object (e.g. line, contour plot, etc) specific to the particular object plotaxes.

plotitem is now an object of class ClawPlotItem and various attributes can be set, e.g.:

```
plotitem.plotstyle = '-'
plotitem.color = 'r'
```

for a solid line that is red.

There are many other *ClawPlotItem* (page 255) attributes and methods.

5.4 current_data

The Python plotting routines often allow the user to specify a call back function as plotting parameters, for example *ClawPlotAxes* (page 254) has an attribute *afteraxes* that can be set to a function to be executed after performing all plots on the specified axes. This is useful for setting parameters for these axes that are not covered by the provided attributes, for annotating the plots on these axes, for adding a plot of the true solution, etc.

Call back functions include:

- *beforeframe* and *afterframe* attributes of *ClawPlotData* (page 251)
- *afteraxes* attribute of *ClawPlotAxes* (page 254)
- *afteritem*, *afterpatch*, *plot_var*, *map2d_to_1d* attributes of *ClawPlotItem* (page 255)

All of these functions are designed to take a single argument *current_data*, an object with attributes that may be useful to the user in evaluating the function.

Warning: *The mapc2p function* (page 68) is one exception that does not take argument *current_data*.

5.4.1 Attributes of *current_data*:

Some of these may be unavailable because they don't make sense in the current context, e.g. in a *beforeframe* function.

plotdata :
parent *ClawPlotData* (page 251) object. From this object virtually any relevant data can be accessed. Other attributes are defined for convenience. If you find you frequently need something else, you could modify `pyclaw.plotters.frametools` to add this to *current_data*.

frameno :
The current frame number

patch :
Object of `pyclaw.solution.patch` with data for the last patch plotted.

patchno :
Grid number of this patch, of interest only in AMR calculations.

q :
q array for current frame, so for example the in a scalar 2d problem the value in the (i,j) cell would be *current_data.q[0,i,j]* (remember that Python always indexes starting at 0).

In an AMR calculation q will be from the last patch plotted.

aux :
aux array for current frame, provided these have been output by the Fortran code. At the moment this requires modifying the library routine *outN.f* to set outaux = .true. so that fort.a files are produced along with fort.q files. [This should be an input parameter!]

If fort.a files are not found then current_data.aux will be None.

In an AMR calculation aux will be from the last patch plotted.

var :
array of the variable actually plotted most recently, e.g. if *plotitem.plot_var == 0* then in 2d *current_data.var[i,j] == current_data.q[0,i,j]*.

level :
For AMR computations, where *current_data.patch* is for the last patch plotted, *current_data.level* is the AMR level of this patch. Particularly useful in *afterpatch* functions.

t :
the current time t.

x :
x array of cell centers corresponding to *current_data.q*.

y :
y array of cell centers corresponding to *current_data.q* (only in 2d).

xlower :
left edge of current patch.

ylower :
left edge of current patch in y (only in 2d).

5.5 Plotting examples

See *Using setplot.py to specify the desired plots* (page 251) and the examples in the *Galleries of all Clawpack applications* (page 25). The code that produced each set of plots can be found in the Clawpack directory indicated.

If you wish to use the Matlab tools, see the *Matlab Gallery* (page 273).

5.6 Plotting hints and FAQ

See also:

Visclaw Plotting options (page 247), *Using setplot.py to specify the desired plots* (page 251), *Plotting examples* (page 262)

Contents

More to come!

5.6.1 What's the difference between *make .plots* and *make plots* ?

The default Makefile configuration in Version 4.5.1 was changed to allow two different targets `.plots` and `plots`. The former creates a hidden file `.plots` whose modification time is used to check dependencies. The plotting commands will be performed again only if the output of `setplot` file has been changed, and will also re-run the code if it appears that the output is out of date relative to the `setrun` file, for example. If you want to create the plots without checking dependencies (and perhaps accidentally re-running the code), use the `make plots` option. `plots` is a phony target in the default Makefile.

5.6.2 How to plot a something other than a component of q?

Objects of class *ClawPlotItem* (page 255) have an attribute `plot_var`. If this is set to an integer than the corresponding component of q is plotted (remembering that Python indexing starts at 0, so `plot_var = 0` will specify plotting the first component of q, for example).

If plot_var is a function then this function is applied to applied to *current_data* (page 261) and should return the array of values to be plotted. For an example, see *plotexample-acou-1d-6*.

Sometimes you want to plot something other than the solution on the patch, for example to add another feature to a plot of the solution. This can be done via an `afteraxes` command, for example, which is called after all items have been plotted on the current axes. See *ClawPlotAxes* (page 254) for details and an example.

5.6.3 How to add another curve to a plot, e.g. the true solution?

The `afteraxes` attribute of a *ClawPlotAxes'* object can be specified as either a string or a function. The string is executed (using `exec(...)`) or the function is called after performing all plots on these axes (as specified by *ClawPlotItem'* objects). This can be used to add a curve to a plot.

For example, if the true solution to an advection equation is known to be $q(x,t) = \sin(x+t)$, this could be added to a plot as a red curve via:

```
def add_true_solution(current_data):
    from pylab import sin, plot
    x = current_data.x
    t = current_data.t
    qtrue = sin(x+t)
```

```
    plot(x, qtrue, 'r')

plotaxes.afteraxes = add_true_solution
```

5.6.4 How to change the title in a plot?

The `title` attribute of a *ClawPlotAxes*' object determines the title that appears a the top of the plot.

The `title_with_t` attributed determines if the time is included in this title. If True (the default value), then "at time t = ..." is appended to the title. The time is printed using format `14.8f` if `(t>=0.001) & (t<1000.)`, or format `14.8e` more generally.

It is also possible to change the title using and afteraxes function. For example, to create a larger title with fontsize 20 and only 4 digits in t:

```
def add_title(current_data):
    from pylab import title
    t = current_data.t
    title("Solution at time t = %10.4e" % t, fontsize=20)

plotaxes.afteraxes = add_title
```

5.6.5 How to specify `outdir`, the directory to find `fort.*` files for plotting?

This is normally determined by the `outdir` attribute of the *ClawPlotData* (page 251) object directing the plotting. But see the next FAQ for the option of using different directories for some plot items (e.g. to compare results of two computations).

If you are making a set of hardcopy plots using:

```
$ make .plots
```

or

```
$ make plots
```

then `outdir` is specified in the Makefile by setting the `CLAW_OUTDIR` variable.

If you are making plots interactively using **Iplotclaw_**, then you can directly specify the `outdir` as a parameter, e.g.:

```
In[1]: ip=Iplotclaw(outdir="_output");   ip.plotloop()
```

If you don't specify this parameter, **'Iplotclaw'_** will look for a file `.output` in the current directory. If you created the `fort.*` files by the command:

```
$ make .output
```

then the output directory is set in the Makefile and the file `.output` contains the path to the output directory.

Note: If you use

```
$ make output
```

which does not check dependencies, this also does not create a target file `.output`.

If the file `.output` does not exist, `outdir = '.'` is used by default, the current directory.

Note that if you stop a calculation mid-stream using `<ctrl>-C`, the file `.output` may not exist or be correct, since this file is written after the execution finishes.

5.6.6 How to specify a different `outdir` for some plot item?

If you want one plot item on an axis to use the default `plotdata.outdir` while another to take data from a different directory (in order to compare two computations, for example), you can set the `outdir` attribute of a *ClawPlotItem* (page 255) directly. If you do not set it, by default it inherits from the *ClawPlotFigure* (page 253) object this item belongs to.

For example, you might have the following in your `setplot` function:

```
plotfigure = plotdata.new_plotfigure(name='compare', figno=1)
plotaxes = plotfigure.new_plotaxes()

plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
plotitem.plot_var = 0
plotitem.plotstyle = '-o'
plotitem.color = 'b'

plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
import os
plotitem.outdir = os.path.join(os.getcwd(), '_output2')
plotitem.plot_var = 0
plotitem.plotstyle = '-+'
plotitem.color = 'r'
```

This would plot results from `plotdata.outdir` as blue circles and results from `./_output2` as red plus signs. It's best to give the full path name, e.g. as done here using `os.path.join(os.getcwd(), '_output2')`.

5.6.7 How to set plot parameters that are not provided as attributes of *ClawPlotItem*?

Some commonly used plotting parameters can be specified as an attribute of a *ClawPlotItem*', for example:

```
plotitem = plotaxes.new_plotitem(plot_type='1d_plot')
plotitem.plot_var = 0
plotitem.plotstyle = '-'
plotitem.color = 'b'
```

specifies plotting a blue line. These attributes are used in the call to the matplotlib `plot` function. The `plot` function has many other keyword parameters that are not all duplicated as attributes of *ClawPlotItem*'. To change these, the `kwargs` attribute can be used.

For example, to plot as above, but with a wider blue line, append the following:

```
plotitem.kwargs = {'linewidth': 2}
```

If you try to specify the same keyword argument two different ways, e.g.:

```
plotitem.color = 'b'
plotitem.kwargs = {'linewidth': 2, 'color': 'r'}
```

the value in `kwargs` takes precedence. It is the `kwargs` dictionary that is actually used in the call, and the `color` attribute is checked only if it has not been defined by the user in the `kwargs` attribute.

5.6.8 How to change the size or background color of a figure?

By default, a figure is created of the default matplotlib size, with a tan background. Any desired keyword arguments to the matplotlib figure command can be passed using the `kwargs` attributed of *ClawPlotFigure*'. For example, to

create a figure that is 10 inches by 5 inches with a pink background:

```
plotfigure = plotdata.new_plotfigure(name='pinkfig', figno=1)
plotfigure.kwargs = {'figsize': [10,5],  'facecolor': [1, .7, .7]}
```

5.6.9 Specifying colormaps for pcolor or contourf plots

The matplotlib module *matplotlib.cm* provides many colormaps that can be acquired as follows, for example:

```
from matplotlib import cm
cmap = cm.get_cmap('Greens')
```

matplotlib.colors provides some tools for working with colormaps, and some additional colormaps and tools can be found in *clawpack.visclaw.colormaps*.

In particular, the *make_colormaps* function simplifies the creation of new colormaps interpolating between specified colors. For example, a colormap fading from blue to yellow to red can be created with the command:

```
from clawpack.visclaw import colormaps
yellow_red_blue = colormaps.make_colormap({0.:'#ffff00', 0.5:[1,0,0], 1.:'b'})
```

The argument of *make_colormaps* is a dictionary that maps values to colors, with linear interpolation between the specified values. Each color can be specified in various ways, e.g. in the example above blue is specified as the matlab style 'b', yellow with an html hex string, and red with an RGB tuple *[1,0,0]*.

The colormap above is also predefined as *clawpack.visclaw.colormaps.yellow_red_blue* and is used in many Clawpack examples.

The function *clawpack.visclaw.colormaps.showcolors(cmap)* can be used to display a colormap. '

5.6.10 How to debug setplot.py?

Suppose you are working in an interactive Python shell such as ipython and encounter the following when trying to plot with **'Iplotclaw'_**:

```
In [3]: ip=Iplotclaw(); ip.plotloop()
*** Error in call_setplot: Problem executing function setplot
*** Problem executing setplot in Iplotclaw
    setplot =  setplot.py
*** Either this file does not exist or
    there is a problem executing the function setplot in this file.
*** PLOT PARAMETERS MAY NOT BE SET! ***

Interactive plotting for Clawpack output...

Plotting data from outdir =  _output
Type ? at PLOTCLAW prompt for list of commands

    Start at which frame [default=0] ?
```

This tells you that there was some problem importing `setplot.py`, but is not very informative and it is hard to debug from within the `Iplotclaw.plotloop` method. You may also run into this if you modify `setplot.py` (inadvertantly introducing a bug) and then use the `resetplot` option:

```
PLOTCLAW > resetplot
Executing setplot from  setplot.py
*** Error in call_setplot: Problem executing function setplot
```

```
*** Problem re-executing setplot
PLOTCLAW >
```

If you can't spot the bug by examing `setplot.py`, it is easiest to debug by exiting the plotloop and doing:

```
PLOTCLAW > q
quitting...

In [4]: import setplot
In [5]: pd = ip.plotdata
In [6]: pd = setplot.setplot(pd)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)

      8
      9     # Figure for q[0]
---> 10     plotfigure = plotdata.new_plotfgure(name='q[0]', figno=1)
     11
     12     # Set up for axes in this figure:

AttributeError: 'ClawPlotData' object has no attribute 'new_plotfgure'
```

In this case, the error is that `new_plotfigure` is mis-spelled.

In ipython you can also easily turn on the Python debugger pdb:

```
In [9]: pdb
Automatic pdb calling has been turned ON

In [10]: pd = setplot.setplot(pd)
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
      8
      9     # Figure for q[0]
---> 10     plotfigure = plotdata.new_plotfgure(name='q[0]', figno=1)
     11
     12     # Set up for axes in this figure:

AttributeError: 'ClawPlotData' object has no attribute 'new_plotfgure'

ipdb>
```

For more complicated debugging you could now explore the current state using any pdb commands, described in the documentation (http://docs.python.org/library/pdb.html). See also the ipython documentation (http://ipython.scipy.org/doc/manual/html/index.html).

5.7 GeoClaw plotting tools

Needs updating

The module *$CLAW/visclaw/geoplot.py* contains some useful tools for plotting GeoClaw output.

To be continued. See comments in the module.

5.7.1 Plotting water depth or surface elevation

For information on using masked arrays, see:

- Masked array operations (http://docs.scipy.org/doc/numpy/reference/routines.ma.html)

5.7.2 Tips on latitude-longitude coordinate axes

With the default style, matplotlib axis labels are often hard to read when plotting in latitude and longitude. It may help to disable offset labeling and to rotate the longitude labels:

```
ticklabel_format(format='plain',useOffset=False)
xticks(rotation=20)
```

To set the aspect ratio so that latitude and longitude are scaled appropriately, set *mean_latitude* to some average latitude value in the region of interest and then:

```
gca().set_aspect(1./cos(mean_latitude*pi/180.))
```

5.8 Plotting using Matlab

Before version 4.3, Clawpack used Matlab (http://www.mathworks.com) (Mathworks, Inc.) for plotting and visualizing results of simulations. For this purpose, an extensive set of plotting tools were developed. These are still available in *$CLAW/visclaw/src/matlab*. The user interface for these routines is essentially unchanged from the previous versions, although several new features have been added.

These graphics tools extend standard Matlab plotting routines by allowing for easy plotting of both 2d and 3d adaptively refined mesh data produced from AMRClaw and solutions on 2d manifolds, produced from either single grid Clawpack, or AMRClaw. In each of these cases, the user can easily switch on or off the plotting of grid lines (on a per-level basis), contour lines, isosurfaces, and AMR patch borders, cubes and other graphical items. In 3d, the user can create a custom set of slices, and then loop through the slices in the x,y or z directions. All visualization assumes finite volume data, and individual plot "patches" use cell-averaged values to color mesh cells directly. No graphical interpolation is done when mapping to the colormap.

5.8.1 The Matlab search path

To use the Matlab plotting tools with Clawpack, the user will need to first be sure that the necessary Matlab routines are on the Matlab search path. This can be done by explicitly setting the MATLABPATH environment variable. In bash, this is done via

```
$ export MATLABPATH ${CLAW}/visclaw/src/matlab
```

Alternatively, one can permanently add this directory to the Matlab search path using the Matlab "pathtool" command:

```
>> pathtool
```

5.8.2 Creating output files

To visualize Clawpack output using the Matlab plotting routines, first produce output files from an example using:

```
$ make .output
```

This will build the appropriate Clawpack executable, create necessary input files for the executable, and finally run the executable to create output files. These output files are stored by default in the directory '_output'.

5.8.3 The plotclaw command

Once output files, e.g. 'fort.q0000', 'fort.q0001', and so on, and corresponding fort.t0000' or 'fort.t0001' files have been created, the user can plot them in Matlab by entering one of the following commands (depending on whether the output is one, two or three dimensional) at the Matlab prompt:

```
>> plotclaw1
```

or:

```
>> plotclaw2
```

or:

```
>> plotclaw3
```

Initially, the user is prompted to run a file **'setplot'_**. For example:

```
>> plotclaw2

plotclaw2  plots 2d results from clawpack or amrclaw
Execute setplot2 (default = yes)?
```

The setplot file sets various parameters in the base workspace needed to create the plot (see below for more details on this file). Entering [enter] at this prompt will run the file.

Successively hitting [enter] steps through and plots data from each of the fort files in order. In one and two dimensions, this plotting prompt is:

```
Frame 2 at time t = 0.2
Hit <return> for next plot, or type k, r, rr, j, i, q, or ?
```

In three dimensions, one can optionally step through user defined slices of data by entering 'x', 'y' or 'z' at the command prompt:

```
Frame 1 at time t = 0.0625
Hit <return> for next plot, or type k, r, rr, j, i, x, y, z, q, or ?
```

A description of the plot prompt options is given by entering '?':

```
Frame 2 at time t = 0.2
Hit <return> for next plot, or type k, r, rr, j, i, q, or ?  ?
  k  -- keyboard input.  Type any commands and then "return"
  r  -- redraw current frame, without re-reading data
  rr -- re-read current file,and redraw frame
  j  -- jump to a particular frame
  i  -- info about parameters and solution
  x  -- loop over slices in x direction (3d only)
  y  -- loop over slices in y direction (3d only)
  z  -- loop over slices in z direction (3d only)
  q  -- quit
```

After the graphics routines have created the plot, but before the user is returned to the plot prompt, a file afterframe (page 270) is called. This file contains user commands to set various plot properties. See below for more details on what the user might wish to include in this file.

5.8.4 The setplot file

The properties of the Matlab plot are controlled through two main user-defined files located, typically, in the current working directory. The first of these files, the 'setplot' file (e.g. setplot1.m, setplot2.m or setplot3.m) control basic plot properties that must be known before the plot is created. Such properties include

- component of q to plot, (.e.g. rho=1, rhou=2, rhov=3 and so on).
- a user defined quantity (e.g. pressure or velocity) to plot,
- the maximum number of frames to plot
- the location of the output files
- the line type or symbol type for 1d plots or scatter plots. Different symbols or line types can be specified for each AMR level.
- the plot type, e.g. a pseudo-color plot, a Schlieren type plot or a scatter plot.
- grid mappings for mapped grids or manifold calculations,
- user defined slices through the data (3d data)
- isosurface properties (3d plots)

A typical setplot file might contain the following parameter settings:

```
% ------------------------------------------------
% file: setplot2.m (not all parameters are shown)
% ------------------------------------------------
OutputDir = '_output';
PlotType = 1;                % Create a pseudo-color plot
mq = 1;                      % which component of q to plot
UserVariable = 0;            % set to 1 to specify a user-defined variable
UserVariableFile = ' ';      % name of m-file mapping data to q
MappedGrid = 0;              % set to 1 if mapc2p.m exists for nonuniform grid
MaxFrames = 1000;            % max number of frames to loop over
MaxLevels = 6;               % max number of AMR levels
...
```

One of the main uses of the 'keyboard' option described in the plotclaw (page 269) section is to allow the user to temporarily change the value of plotting parameters set in the setplot file.

To ensure that the required set of variables is defined, the user is encouraged to create and modify a local copy of setplot1.m, setplot2.m or setplot3.m found in '${CLAW}visclaw/src/matlab'.

To get more help on what types of settings can be specified in the setplot file, enter the following command:

```
>> help setplot
```

Each of the examples in Clawpack include a 'setplot' file which you can browse to get an idea as to what can be put in the file.

5.8.5 The afterframe file

The 'afterframe.m' script is the second file which control aspects of the plot and is called after the plot has been created. The following are commonly set in the afterframe file:

- set axis limits and scaling
- add a 1d reference solution (1d plots and scatter plots)
- print out the current frame to a png, jpg or other graphics format file.

- add, show or hide contour lines on slices (2d/3d)
- show or hide AMR patch and cube borders (2d/3d)
- modify the colormap (2d/3d)
- set the color axis (2d/3d)
- show or hide grid lines on different AMR levels (2d/3d)
- add lighting to isosurfaces (3d)
- hide or show isosurfaces (3d)
- show or hide slices (3d)

A typical 'afterframe' file might contain the following commands:

```
% ------------------------------------------------
% file: afterframe.m
% ------------------------------------------------
axis([-1 1 -1 1]);      % Set the axis limits
daspect([1 1 1]);       % Set the aspect ratio

colormap(jet);

showpatchborders;       % Show outlines of AMR patch borders
showgridlines(1:2);     % Show gridlines on level 1 and 2 grids

cv = linspace(-1,1,21); % Values for contour levels
cv(cv == 0) = [];
drawcontourlines(cv);   % add contour lines to a plot

caxis([-1 1]);          % Set the color axis

shg;                    % Bring figure window to the front

fstr = framename(Frame,'frame0000','png','_plots');
print('-dpng',fstr);       % Create .png file of figure.

clear afterframe;
```

The final 'clear' statement is added so that any modifications that the user makes to the afterframe file while stepping through plot frames will take effect immediately.

When plotting results from AMR runs, the user can also create an 'aftergrid.m' file. This file will be called after each individual grid of data is plotted.

The user is encouraged to browse the 'afterframe.m' file available with each Clawpack example to get a better idea as to what one might include in this file.

5.8.6 Getting help

To get help on any of the topics available in the Matlab graphics tools, you can always issue the command:

```
>> help clawgraphics
```

at the Matlab prompt. This will bring up a list of topics which you can get further help on.

5.8.7 Trouble shooting

Below are a few potential problems one can run into with the Matlab plotting routines.

Output files not found

The following error message indicates that the output files have not been found:

```
Hit <return> for next plot, or type k, r, rr, j, i, x, y, z, q, or ?

Frame 2 (./fort.t0002) does not exist ***


Frame 2(ascii) does not exist ***
```

Be sure to check that that the variable 'OutputDir', set in the setplot file, points to the proper location of the output files that you wish to plot. Second, double check that you actually have fort.[t/q]XXXX files in that directory.

MaxFrames not set

The error message below most likely means that a 'setplot' script containing a definition for MaxFrames was not run:

```
>> plotclaw2

plotclaw2  plots 2d results from clawpack or amrclaw
Execute setplot2 (default = yes)? no

MaxFrames parameter not set... you may need to execute setplot2
```

To correct this problem, the user should make sure that they have local copy of a setplot file in their working directory, that it defines the required set of variables and that it is run at least once before the plotclaw command.

Switching examples

The graphics are controlled to a large extent using variables that are set in the Matlab base workspace. This can lead to unpredictable results when switching between Clawpack examples.

To illustrate what can go wrong, suppose one sets:

```
MappedGrid = 1;          % assumes that mapc2p file exists
```

in the setplot file for one example, and then switches to a second example which is not a simulation on a mapped grid. If the variable 'MappedGrid' is not explicitly set to zero in the setplot file for the second example, the Matlab routines will look for a grid mapping file 'mapc2p.m' which may not be found for the second example.

To avoid such potential clashes of variables, the user is strongly encouraged to enter the command:

```
>> clear all;
```

before switching examples. This will clear the base workspace of all plotting parameters and avoid potential conflicts in base variable settings.

The user is also encouraged to issue a command:

```
>> close all
```

in situations where the one example explicitly sets plotting features such as a colormap, or axes scaling that are not overridden by subsequent plot commands.

5.8.8 Matlab Gallery

The interested user is encouraged to browse the Matlab Gallery (http://math.boisestate.edu/ calhoun/visclaw/matlab_gallery/test_graphics/index.html) for examples of the types of plots that can be created with the Clawpack Matlab plotting routines.

5.9 Plotting with VisIt

2d and 3d plots can be rendered using the visualization package VisIt (https://wci.llnl.gov/codes/visit/home.html). VisIt has a Claw reader that can be used to import data from Clawpack, see Application Toolkit Formats (http://www.visitusers.org/index.php?title=Detailed_list_of_file_formats_VisIt_supports#Application_Toolkit_Formats) for other formats that VisIt supports.

The ASCII output files generated by Clawpack can be read in directly to VisIt if one additional file is added to the directory of output files: a file named *plot.claw* is required with contents:

```
DIR=.
TIME_FILES_SCANF=fort.t%04d
GRID_FILES_SCANF=fort.q%04d
```

When using the VisIt GUI, simply open this file to load the data. See the VisIt documentation (https://wci.llnl.gov/codes/visit/doc.html) for information on how to use the GUI.

To do:

- Create Python tools using the Python interface to VisIt so that plots can be specified in *setplot.py*.
- Add routines to Clawpack to output data in Silo format, the binary format recommended for VisIt, and/or other binary formats.

CHAPTER

SIX

DEVELOPERS' RESOURCES

6.1 Clawpack Community

We welcome new users!

If you run into problems or can't find the answer to your question in these docs, please send a note to the claw-users google group (https://groups.google.com/forum/#!forum/claw-users).

We are also starting to experiment with the following:

- clawpack on Gitter (https://gitter.im/clawpack/public) (Sign in with your github account if you want to post a comment or ask a question.)
- clawpack on twitter (https://twitter.com/clawpack).

We welcome new contributors and developers!

The pages below give some tips for those who want to get involved.

- *Contributing examples and applications* (page 43) for ideas on how to make applications or examples you've developed available to other users.
- *Developers' Guide* (page 276) for instructions on how to clone the git repositories, create your own forks, issue pull requests, etc.
- Join or browse the claw-dev google group (https://groups.google.com/forum/#!forum/claw-dev) to track discussion on the code.

6.1.1 Workshops and Sprint Sessions

Upcoming

Please join us at one of the upcoming sessions, or help plan one elsewhere...

- Tentative plans for sprinting following the SIAM CSE (http://www.siam.org/meetings/cse15/submissions.php) in Salt Lake City, March 19, 2015 (?).

  If you're at the conference, check out the Minisymposterium Sunday, March 15, 2015 on Clawpack Development, Extensions and Applications (http://meetings.siam.org/sess/dsp_programsess.cfm?SESSIONCODE=20422)

- Claw-Dev workshop in September (?), 2015 (Date and location TBD).

Previous

See the links below for summaries of some projects that have been tackled (and/or are still work in progress).

- $[HPC]^3$ workshops at KAUST in 2011 (https://sites.google.com/site/hpc3atkaust/), 2012 (https://github.com/clawpack/pyclaw/wiki/HPC3-2012), 2014 (https://github.com/clawpack/pyclaw/wiki/HPC3-2014)
- Claw-Dev workshop at UW in 2013 (http://www.clawpack.org/clawdev2013/)

6.2 Developers' Guide

Contents

6.2.1 Guidelines for contributing

When preparing contributions, please follow the guidelines in *contribution*. Also:

- If the planned changes are substantial or will be backward-incompatible, it's best to discuss them on the claw-dev Google group (http://groups.google.com/group/claw-dev) before starting.
- Make sure all tests pass and all the built-in examples run correctly.
- Be verbose and detailed in your commit messages and your pull request.
- It may be wise to have one of the maintainers look at your changes before they are complete (especially if the changes will necessitate modifications of tests and/or examples).
- If your changes are not backward-compatible, your pull request should include instructions for users to update their own application codes.

Reporting and fixing bugs

If you find a bug, post an issue with as much explanation as possible on the appropriate issue tracker (for instance, the PyClaw issue tracker is at https://github.com/clawpack/pyclaw/issues. If you're looking for something useful to do, try tackling one of the issues listed there.

Developer communication

Developer communication takes place on the google group at http://groups.google.com/group/claw-dev/, and (increasingly) within the issue trackers on Github.

6.2.2 Installation instructions for developers

Cloning the most recent code from Github

You can create a read-only development version of Clawpack via:

```
git clone git://github.com/clawpack/clawpack.git
cd clawpack
python setup.py git-dev
```

This downloads the following clawpack modules as subrepositories checked out at specific commits (as opposed to the tip of a branch).

- https://github.com/clawpack/pyclaw (Python code, some of which is needed also for Fortran version)
- https://github.com/clawpack/clawutil (Utility functions, Makefile.common used in multiple repositories)
- https://github.com/clawpack/classic (Classic single-grid code)
- https://github.com/clawpack/amrclaw (AMR version of Fortran code)
- https://github.com/clawpack/riemann (Riemann solvers)
- https://github.com/clawpack/visclaw (Python graphics and visualization tools)
- https://github.com/clawpack/geoclaw (GeoClaw)

This should give a snapshot of the repositories that work well together. (Note that there are many inter-dependencies between code in the repositories and checking out a different commit in one repository may break things in a different repository.)

If you want to also install the PyClaw Python components, you can then do:

```
python setup.py install
```

If you plan to work on the Python parts of Clawpack as a developer, you may instead wish to do:

```
pip install -e .
```

The advantage of this is that when you edit Python code in your clawpack directly, it will immediately take effect, without the need to install again. However, the (potential) danger of this approach is that the path to your clawpack directory will be stored in the file site-packages/easy-install.pth and prepended to your PYTHONPATH whenever you run Python. This path will take precedence over any manually added paths, unless you delete the .pth file.

If you want to use the Fortran versions in *classic*, *amrclaw*, *geoclaw*, etc., you need to set environment variables and proceed as described at *Set environment variables* (page 9).

Updating to the latest development version

The repositories will each be checked out to a specific commit and will probably be in a detached-head state. You will need to checkout *master* in each repository to see the current head of the master branch.

You should never commit to *master*, only to a feature branch, so the *master* branch should always reflect what's in the main *clawpack* repository. You can update it to reflect any changes via:

```
git checkout master
git fetch origin
git merge origin/master
```

or simply:

```
git pull origin master:master
```

Remember that you need to do this in each repository before running anything to make sure everything is up to date with *master*.

Adding your fork as a remote

If you plan to make changes and issue pull requests to one or more repositories, you will need to do the following steps for each such repository:

1. Go to http://github.com/clawpack and fork the repository to your own Github account. (Click on the repository name and then the *Fork* button at the top of the screen.)
2. Add a *remote* pointing to your repository. For example, if you have forked the *amrclaw* repository to account *username*, you would do:

   ```
   cd amrclaw
   git remote add username git@github.com:username/amrclaw.git
   ```

   You should push only to this remote, not to *origin*, e.g.:

   ```
   git push username
   ```

You might also want to clone some or all of the following repositories:

- https://github.com/clawpack/doc (documentation)
- https://github.com/clawpack/apps (To collect applications)
- https://github.com/clawpack/regression (Regression tests)
- https://github.com/clawpack/clawpack-4.x (Previous versions, 4.6)

These are not brought over by cloning the top *clawpack* super-repository. You can get one of these in read-only mode by doing, e.g.:

```
git clone git://github.com/clawpack/doc.git
```

Then go through the above steps to add your own fork as a remote if you plan to modify code and issue pull requests.

6.2.3 Modifying code

Before making changes, make sure *master* is up to date:

```
git checkout master
git pull
```

Then create a new branch based on *master* for any new commits:

```
git checkout -b new_feature master
```

Now make changes, add and commit them, and then push to your own fork:

```
# make some changes
# git add the modified files
git commit -m "describe the changes"

git push username new_feature
```

If you want these changes pulled into *master*, you can issue a pull request from the github page for your fork of this repository (make sure to select the correct branch of your repository).

Note: If you accidentally commit to *master* rather than creating a feature branch first, you can easily recover:

```
git checkout -b new_feature
```

will create a new branch based on the current state and history (including your commits to *master*) and you can just continue adding additional commits.

The only problem is your *master* branch no longer agrees with the history on Github and you want to throw away the commits you made to *master*. The easiest way to do this is just to make sure you're on a different branch, e.g.,

```
git checkout new_feature
```

and then:

```
git branch -D master
git checkout -b master origin/master
```

This deletes your local branch named *master* and recreates a branch with the same name based on *origin/master*, which is what you want.

Issuing a pull request

Before issuing a pull request, you should make sure you have not broken anything:

1. Make sure you are up to date with *master*:

   ```
   git checkout master
   git pull
   ```

   If this does not say "Already up-to-date" then you might want to rebase your modified code onto the updated master. With your feature branch checked out, you can see what newer commits have been added to *master* via:

   ```
   git checkout new_feature
   git log HEAD..master
   ```

   If your new feature can be added on to the updated master, you can rebase:

   ```
   git rebase master
   ```

   which gives a cleaner history than merging the branches.

2. Run the appropriate regression tests. If you have modified code in pyclaw or riemann, then you should run the pyclaw tests. First, if you have modified any Fortran code, you need to recompile:

   ```
   cd clawpack/
   pip install -e .
   ```

   Then run the tests:

   ```
   cd pyclaw
   nosetests
   ```

If any tests fail, you should fix them before issuing a pull request.

To issue a pull request (PR), go to the Github page for your fork of the repository in question, select the branch from which you want the pull request to originate, and then click the *Pull Request* button.

Testing a pull request

To test out someone else's pull request, follow these instructions: For example, if you want to try out a pull request coming from a branch named *bug-fix* from user *rjleveque* to the *master* branch of the *amrclaw* repository, you would do:

```
cd $CLAW/amrclaw   # (and make sure you don't have uncommitted changes)
git checkout master
git pull  # to make sure you are up to date

git checkout -b rjleveque-bug-fix master
git pull https://github.com/rjleveque/amrclaw.git bug-fix
```

This puts you on a new branch of your own repository named *rjleveque-bug-fix* that has the proposed changes pulled into it.

Once you are done testing, you can get rid of this branch via:

```
git checkout master
git branch -D rjleveque-bug-fix
```

Top-level pull requests

The top level *clawpack* repository keeps track of what versions of the subrepositories work well together.

If you make pull requests in two different repositories that are linked, say to both *pyclaw* and *riemann*, then you should also push these changes to the top-level *clawpack* repository and issue a PR for this change:

```
cd $CLAW   # top-level clawpack repository
git checkout master
git pull
git checkout -b pyclaw-riemann-changes
git add pyclaw riemann
git commit -m "Cross-update pyclaw and riemann."
git push username pyclaw-riemann-changes
```

Git workflow

See *git-resources* for useful links.

6.2.4 Catching errors with Pyflakes and Pylint

Pyflakes and Pylint are Python packages designed to help you catch errors or poor coding practices. To run pylint on the whole PyClaw package, do:

```
cd $PYCLAW
pylint -d C pyclaw
```

The *-d* option suppresses a lot of style warnings, since PyClaw doesn't generally conform to PEP8. To run pylint on just one module, use something like:

```
pylint -d C pyclaw.state
```

Since pylint output can be long, it's helpful to write it to an html file and open that in a web browser:

```
pylint -d C pyclaw.state -f html > pylint.html
```

Pyflakes is similar to pylint but aims only to catch errors. If you use Vim, there is a nice extension package pyflakes.vim (https://github.com/kevinw/pyflakes-vim) that will catch errors as you code and underline them in red.

6.2.5 Checking test coverage

You can use nose to see how much of the code is covered by the current suite of tests and track progress if you add more tests

```
nosetests --with-coverage --cover-package=pyclaw --cover-html
```

This creates a set of html files in *./cover*, showing exactly which lines of code have been tested.

6.3 git resources

6.3.1 Tutorials and summaries

- github help (http://help.github.com) has an excellent series of how-to guides.
- learn.github (http://learn.github.com/) has an excellent series of tutorials
- The pro git book (http://progit.org/) is a good in-depth book on git.
- A git cheat sheet (http://github.com/guides/git-cheat-sheet) is a page giving summaries of common commands.
- The git user manual (http://www.kernel.org/pub/software/scm/git/docs/user-manual.html)
- The git tutorial (http://www.kernel.org/pub/software/scm/git/docs/gittutorial.html)
- The git community book (http://book.git-scm.com/)
- git ready (http://www.gitready.com/) — a nice series of tutorials
- git casts (http://www.gitcasts.com/) — video snippets giving git how-tos.
- git magic (http://www-cs-students.stanford.edu/ blynn/gitmagic/index.html) — extended introduction with intermediate detail
- The git parable (http://tom.preston-werner.com/2009/05/19/the-git-parable.html) is an easy read explaining the concepts behind git.
- git foundation (http://matthew-brett.github.com/pydagogue/foundation.html) expands on the git parable (http://tom.preston-werner.com/2009/05/19/the-git-parable.html).
- Fernando Perez' git page — Fernando's git page (http://www.fperez.org/py4science/git.html) — many links and tips
- A good but technical page on git concepts (http://www.eecs.harvard.edu/ cduan/technical/git/)
- git svn crash course (http://git-scm.com/course/svn.html): git for those of us used to subversion (http://subversion.tigris.org/)
- Use gitk to understand git (http://lostechies.com/joshuaflanagan/2010/09/03/use-gitk-to-understand-git/) has a good description of merging.

- Many more documentation links (https://git.wiki.kernel.org/index.php/GitDocumentation)

6.3.2 Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management (http://kerneltrap.org/Linux/Git_Management)
- Linus Torvalds on linux git workflow (http://www.mail-archive.com/dri-devel@lists.sourceforge.net/msg39091.html) . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

6.3.3 Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add (http://www.kernel.org/pub/software/scm/git/docs/git-add.html)
- git branch (http://www.kernel.org/pub/software/scm/git/docs/git-branch.html)
- git checkout (http://www.kernel.org/pub/software/scm/git/docs/git-checkout.html)
- git clone (http://www.kernel.org/pub/software/scm/git/docs/git-clone.html)
- git commit (http://www.kernel.org/pub/software/scm/git/docs/git-commit.html)
- git config (http://www.kernel.org/pub/software/scm/git/docs/git-config.html)
- git diff (http://www.kernel.org/pub/software/scm/git/docs/git-diff.html)
- git log (http://www.kernel.org/pub/software/scm/git/docs/git-log.html)
- git pull (http://www.kernel.org/pub/software/scm/git/docs/git-pull.html)
- git push (http://www.kernel.org/pub/software/scm/git/docs/git-push.html)
- git remote (http://www.kernel.org/pub/software/scm/git/docs/git-remote.html)
- git status (http://www.kernel.org/pub/software/scm/git/docs/git-status.html)

6.4 Guide for updating this documentation

The restructured text files are in the clawpac/doc (https://github.com/clawpack/doc) repository in *$CLAW/doc/doc*.

Before proceeding, first make sure other repositories are checked out to master, since some pages now have literalinclude's that bring in code (e.g. setaux_defaults.rst, etc).

To create html files:

```
cd $CLAW/doc/doc
make html
```

To view the files, point your browser to *$CLAW/doc/doc/_build/html/index.html*

If you like what you see, you can push back to your fork and then issue a pull request to have these changes incorporated into the documentation.

6.4.1 Updating the webpages

A few developers can push html files to the repository clawpack/clawpack.github.com (https://github.com/clawpack/clawpack.github.com) which causes them to show up on the web at http://clawpack.github.io.

To do so, first create the html files as described above in *$CLAW/doc/doc/_build/html*. Commit any changed source files and push to clawpack/doc (https://github.com/clawpack/doc).

Then do:

```
cd $CLAW/clawpack.github.com
git checkout master
git pull origin  # make sure you are up to date before doing next steps!

cd $CLAW/doc/doc
source rsync_clawpack.github.sh
```

This copies the contents of *$CLAW/doc/doc/_build/html/* to *$CLAW/clawpack.github.com/*

Then move to the latter repository and add and commit any new or changed files. All files are needed, so

```
git add .
```

should work. For the commit message you might want to add the commit hash of the most recent commit in $CLAW/doc/doc:

```
cd $CLAW/clawpack.github.com
git add .
git commit -m "changes from doc/doc commit <hash>"
```

And finally push to the web:

```
git push origin
```

which assumes that *origin* is *git@github.com:clawpack/clawpack.github.com.git*.

It may take a few minutes for the updated webpages to appear at http://clawpack.github.io/.

Note that http://clawpack.org and http://www.clawpack.org should also resolve properly to http://clawpack.github.io/. and that *www.clawpack.org* should appear in the browser address bar. The file *extra_files/CNAME* combined with settings on the domain server *godaddy.com* determine this behavior.

6.4.2 Extra files for webpages not built by Sphinx

Any files placed in *$CLAW/doc/doc/extra_files* will be copied verbatim (recursively for subdirectories) to the directory *$CLAW/doc/doc/_build/html* when Sphinx is used to build the documentation. These will be copied to *$CLAW/clawpack.github.com/* when the *rsync_clawpack.github.sh* script is run and hence will appear on the webpages.

For example, the file *$CLAW/doc/doc/extra_files/clawdev2013/index.html* should appear at http://www.clawpack.org/clawdev2013/index.html.

The files in *$CLAW/doc/doc/extra_files/links* provide redirects so that links like http://www.clawpack.org/links/an11 resolve properly to webpages on the University of Washington server. Links of this nature have been provided in published paper and some contain large amounts of data that have not been copied to Github.

6.5 Regression testing

Contents

Clawpack includes a number of tests that can be used to check for a working installation or to see whether new changes to the code have broken anything.

6.5.1 Running the tests

If you use multiple git branches, before running the tests you should check that you have checked out appropriate branches of all relevant repositories; see *Keeping track of git versions* (page 287).

PyClaw

Regression tests can be performed via:

```
cd $CLAW/pyclaw/examples
nosetests
```

For more details, see *Running and writing tests in PyClaw* (page 285).

Fortran codes

A few quick tests can be perfomed of the *classic*, *amrclaw*, or *geoclaw* codes by running *make tests* in the corresponding *tests* subdirectory, e.g.:

```
cd $CLAW/classic/tests
make tests
```

This uses *nosetests* to run a few Python scripts that in turn run the Fortran codes and then compare a small set of values derived from the output of the run with values that are stored in these directories. If one of these tests fails then there is a problem to be investigated, but these tests do not provide good coverage of the code or check that everything is working properly.

A somewhat more complete set of tests can be run by executing all of the codes in the *examples* subdirectories and comparing the resulting plots with those archived in the *Galleries of all Clawpack applications* (page 25). An attempt at automating this can be found in the *$CLAW/amrclaw/examples* directory, which uses the *imagediff* tool described below. This is still under development.

Travis continuous integration

Most Clawpack git repositories now contain a file *.travis.yml* at the top level so that every time a pull request is issued on Github, a basic set of tests is run. This uses the Travis continuous integration (https://travis-ci.org/) platform. Shortly after a PR is issued, Travis will run the commands in the *.travis.yml* and report the results on the PR page. Look for a green check mark (good) or a red X (bad) next to a commit hash and click on it to see the Travis output. [Sample output] (https://travis-ci.org/clawpack/clawpack/builds/15269312)

6.5.2 Diff tools for checking test output

chardiff tool for line-by-line comparison of output files

If *_output_old* and *_output_new* are two sets of output files from old and new versions of a code, then it is often useful to do a line by line comparison of all of the files in each directory and display any differences. Standard tools such as *xxdiff* in linux or *opendiff* on a Mac are not very good for this since they try to match up blocks of lines to give the best match and may not compare the files line by line.

The Python script *$CLAW/clawutil/src/python/clawutil/chardiff.py* can be used for this purpose:

```
$ python $CLAW/clawutil/src/python/clawutil/chardiff.py _output_old _output_new
```

will create a new directory with html files showing all differences. It can also be used to compare two individual files. See the docstring for more details.

imagediff tool for pixel comparison of plots

If *_plots_old* and *_plots_new* contain two sets of plots that we hope are identical, the Python script *$CLAW/clawutil/src/python/clawutil/imagediff.py* can be used to compare the corresponding images in each directory and produce html files that show each pair of images side by side. If the images are not identical it also shows an image indicating which pixels are different in the two:

```
$ python $CLAW/clawutil/src/python/clawutil/imagediff.py _plots_old _plots_new
```

will create a new directory with html files showing all differences. It can also be used to compare two individual files. See the docstring for more details.

6.5.3 Running and writing tests in PyClaw

Running the tests

The PyClaw test suite is built around nosetests (http://nose.readthedocs.org/en/latest/) for automatic test discovery, with supplementary functionality from the `pyclaw.util` (page 237) module. To run the complete test suite with helpful output, issue the following command at the top-level of the pyclaw source directory:

```
nosetests -vs
```

To run the parallel versions of the tests (if petsc4py is installed), run:

```
mpirun -n 4 nosetests -vs
```

Replace 4 with the number of processes you'd like test on. Try prime numbers if you're really trying to break things!

The *-vs* switch tells nose to be verbose and to show you stdout, which can be useful when debugging tests. To run the tests with less output, omit the *-vs*.

Running serial tests simultaneously

When running the tests, if your machine has multiple cores you can take advantage of them by doing:

```
nosetests -vs --processes=2
```

(replace "2" with the number of processes you want to spawn). However, using large numbers of processes occasionally causes spurious failure of some tests due to issues with the operating system. If you see this behavior, it's best to run the tests in serial or with a small number of processes.

Running a specific test

The PyClaw tests are associated with particular applications in the examples/ sub- directory of the primary repository directory. If you want to run tests for a specific application, simply specify the directory containing the application you are interested in:

```
nosetests -vs examples/acoustics_3d_variable
```

You can also specify a single file to run the tests it contains.

Doctests

Several of the main PyClaw modules also have doctests (tests in their docstrings). You can run them by executing the corresponding module:

```
cd $PYCLAW/src/pyclaw
python grid.py
python state.py
```

If the tests pass, you will see no output. You can get more output by using the *-v* option:

```
python state.py -v
```

Writing New Tests

If you contribute new functionality to PyClaw, it is expected that you will also write at least one or two new tests that exercise your contribution, so that further changes to other parts of PyClaw or your code don't break your feature.

This section describes some functions in *pyclaw.util* that facilitate testing. You do not have to use any of the functionality offered by *pyclaw.util*, but it may simplify your test-writing and allow you to check more cases than you would easily specify by hand.

The most important function in `pyclaw.util` (page 237) is `pyclaw.util.gen_variants()` (page 239), which allows you to perform combinatorial testing without manually specifying every feature you'd like to perform. Currently, `gen_variants()` (page 239) can multiplicatively exercise kernel_languages (Fortran or Python) and pure PyClaw or PetClaw implementations. This allows you to write one function that tests four variants.

Another function provided by `util` (page 237) is `pyclaw.util.test_app()` (page 239). The `test_app()` (page 239) function will run an application as if started from the command line with the specified keyword arguments passed in. This is useful for testing specific code that does not necessarily work with `petclaw`, for example, and is not expected to.

You will notice that both `gen_variants()` (page 239) and `test_app()` (page 239) require a *verifier* method as an argument. These functions both effectively run tests and verify output with the following function calls:

```
output = application(**kwargs)
check_values = verifier(output)
```

The *verifier* method needs to return *None* if there is no problem with the output, or a sequence of three values describing what was expected, what it received, and more details about the error. A very simple *verifier* method that you can use is `pyclaw.util.check_diff()` (page 238), which can use either an absolute tolerance or a relative tolerance to compare an expected value against the test output from the application.

See *examples/acoustics_1d_homogeneous/test_acoustics.py* for a comprehensive example of how to use `gen_variants()` (page 239) and `check_diff()` (page 238). See *examples/shallow_sphere/test_shallow_sphere.py* for an example that uses `test_app()` (page 239) and also loads a known solution from disk using numpy.

6.6 Keeping track of git versions

The command:

```
python $CLAW/clawutil/src/python/clawutil/claw_git_status.py
```

will report on the status of all Clawpack repositories found, e.g. what branch is checked out, the hash of the most recent commit, and any tracked files with uncommitted changes. This information will be saved to a file *claw_git_status.txt* and any diffs found for uncommitted changes will be saved to a file *claw_git_diffs.txt*.

An optional command line argument allows you to save these files in a different directory, e.g.

```
python $CLAW/clawutil/src/python/clawutil/claw_git_status.py _output
```

This is often useful to do when running a code if you want to later determine exactly what version of the code was used, particularly when doing regression testing.

The function *$CLAW/clawutil/src/python/clawutil/runclaw.py* now has an argument *print_git_status* (with default value *False*). Calling *runclaw* with *print_git_status == True* will write these files to the output directory specified by the *outdir* argument.

Setting the environment variable *GIT_STATUS* to True will insure that *make .output* creates output directories containing the *claw_git_status* files.

See also *Installation instructions for developers* (page 277)

CHAPTER

SEVEN

BIBLIOGRAPHY

7.1 Examples from the book FVMHP

The book Finite Volume Methods for Hyperbolic Problems (http://faculty.washington.edu/rjl/book.html) contains many examples that link to Clawpack codes used to create the figures in the book. These codes are available for Clawpack 4.3 via the book webpage (http://faculty.washington.edu/rjl/book.html)

Some of have been converted to Clawpack 5.x form, with a *setrun.py* file for setting run time data and a *setplot.py* file for specifying plots with Python. See:

- *Specifying classic run-time parameters in setrun.py* (page 56)
- *Using setplot.py to specify the desired plots* (page 251)

7.1.1 Available examples

The examples converted so far can be found in the directory *apps/fvmbook* if you clone the *apps* repository. (See *Clawpack Applications repository* (page 41).)

You can also browse the examples in the *Gallery of fvmbook applications* (page 35).

7.2 Bibliography

7.2.1 Papers describing the Clawpack software and algorithms

7.2.2 Papers describing applications

Note: Add more...

7.2.3 Other references

[BaleLevMitRoss02] D. S. Bale, R. J. LeVeque, S. Mitran, and J. A. Rossmanith. A wave-propagation method for conservation laws with spatially varying flux functions, (http://faculty.washington.edu/rjl/pubs/vcflux/index.html) *SIAM J. Sci.* Comput 24 (2002), 955-978.

```
@article{BaleLevMitRoss02,
    Author = {D. Bale and R. J. LeVeque and S. Mitran and J. A. Rossmanith},
    Title = {A wave-propagation method for conservation laws and balance laws
            with spatially varying flux functions},
    Journal = {SIAM J. Sci. Comput.},
    Pages = {955--978},
    Volume = {24},
    Year = {2002}}
```

[BergerColella89] M. J. Berger and P Colella. 1989. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.* 82, 64–84.

[BergerGeorgeLeVequeMandli11] M. J. Berger, D. L. George, R. J. LeVeque and K. M. Mandli, The GeoClaw software for depth-averaged flows with adaptive refinement, Advances in Water Resources 34 (2011), pp. 1195-1206.

```
@article{BergerGeorgeLeVequeMandli11,
 Author = {M. J. Berger and D. L. George and R. J. LeVeque and K. T.  Mandli},
 Journal = {Adv. Water Res.},
 Pages = {1195-1206},
 Title = {The {GeoClaw} software for depth-averaged flows with adaptive refinement},
 Volume = {34},
 Year = {2011},
 Url = {\url{www.clawpack.org/links/papers/awr11}}}
```

[BergerLeVeque98] M. J. Berger and R. J. LeVeque. 1998. Adaptive Mesh Refinement using Wave-Propagation Algorithms for Hyperbolic Systems. (http://www.amath.washington.edu/ rjl/pubs/amrclaw/index.html) *SIAM J. Numer. Anal.* 35, 2298–2316.

```
@article{BergerLeVeque98,
    Author = {M. J. Berger and R. J. LeVeque},
    Journal = {SIAM J. Numer. Anal.},
    Pages = {2298--2316},
    Title = {Adaptive Mesh Refinement using Wave-Propagation Algorithms for Hyperbolic Systems},
    Volume = {35},
    Year = {1998}}
```

[BergerOliger84] M. J. Berger and J. Oliger. 1984. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comput. Phys.* 53, 484–512.

[BergerRigoutsis91] M. J. Berger and I. Rigoutsos. 1991. An Algorithm for Point Clustering and Grid Generation. *IEEE Trans. Sys. Man & Cyber.* 21, 1278–1286.

[LangsethLeVeque00] J. O. Langseth and R. J. LeVeque. 2000. A wave-propagation method for three-dimensional hyperbolic conservation laws. (http://www.amath.washington.edu/ rjl/pubs/wp3d/index.html) *J. Comput. Phys.* 165, 126–166.

```
@article{LangsethLeVeque00,
    Author = {J. O. Langseth and R. J. LeVeque},
    Title = {A wave-propagation method for three-dimensional hyperbolic conservation laws},
    Journal = {J. Comput. Phys.},
    Pages = {126--166},
    Volume = {165},
    Year = {2000}}
```

[LeVeque96] R. J. LeVeque, 1996. High-resolution conservative algorithms for advection in incompressible flow, (http://faculty.washington.edu/rjl/pubs/hiresadv/index.html)

```
@article{LeVeque1996,
  author="R. J. LeVeque",
  title="High-resolution conservative algorithms for advection in
  incompressible flow",
  journal="SIAM J. Numer. Anal.",
  volume="33",
  year="1996",
  pages="627-665"
}
```

[LeVeque97] R. J. LeVeque, 1997. Wave propagation algorithms for multi-dimensional hyperbolic systems. (http://www.amath.washington.edu/ rjl/pubs/wpalg/index.html) *J. Comput. Phys.* 131, 327–353.:

```
@article{rjl:wpalg,
    Author = {R. J. LeVeque},
    Title = {Wave propagation algorithms for multi-dimensional hyperbolic systems},
    Journal = {J. Comput. Phys.},
    Pages = {327--353},
    Volume = {131},
    Year = {1997}}
```

[LeVeque-FVMHP] R. J. LeVeque. Finite Volume Methods for Hyperbolic Problems. (http://www.amath.washington.edu/ claw/book.html) Cambridge University Press, Cambridge, UK, 2002.

```
@book{LeVeque-FVMHP,
    Author = {R. J. LeVeque},
    Title = {Finite Volume Methods for Hyperbolic Problems},
    Publisher = {Cambridge University Press},
    Year = {2002},
    Url = {http://www.clawpack.org/book.html}}
```

See *Gallery of fvmbook applications* (page 35) for some sample results from this book.

[LeVequeGeorgeBerger] R. J. LeVeque, D. L. George, and M. J. Berger, 2011, Tsunami modelling with adaptively refined finite volume methods, *Acta Numerica,* pp. 211-289.

```
@article{mjb-dg-rjl:actanum2011,
    Author = {R.J. LeVeque  and D. L. George and M. J. Berger},
    Title = {Adaptive Mesh Refinement Techniques for Tsunamis and Other
            Geophysical Flows Over Topography},
    Journal = {Acta Numerica},
    Pages = {211-289},
    Year = {2011}}
```

[KetParLev13] D. I. Ketcheson, Matteo Parsani, and R J LeVeque, 2013, High-order Wave Propagation Algorithms for Hyperbolic Systems, *SIAM Journal on Scientific Computing*, 35(1):A351-A377 (2013)

@article{KetParLev13, Author = {Ketcheson, David I. and Parsani, Matteo and LeVeque, Randall J.}, Journal = {SIAM Journal on Scientific Computing}, Number = {1}, Pages = {A351–A377}, Title = {{High-order Wave Propagation Algorithms for Hyperbolic Systems}}, Volume = {35}, Year = {2013}}

[KetchesonMandliEtAl] David I. Ketcheson, Kyle T. Mandli, Aron J. Ahmadia, Amal Alghamdi, Manuel Quezada de Luna, Matteo Parsani, Matthew G. Knepley, and Matthew Emmett, 2012, PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems, *SIAM Journal on Scientific Computing*, 34(4):C210-C231

```
@article{pyclaw-sisc,
    Author = {Ketcheson, David I. and Mandli, Kyle T. and Ahmadia, Aron J. and
        Alghamdi, Amal and {Quezada de Luna}, Manuel and Parsani, Matteo and
        Knepley, Matthew G. and Emmett, Matthew},
    Title = {{PyClaw: Accessible, Extensible, Scalable Tools for Wave Propagation Problems}},
    Journal = {SIAM Journal on Scientific Computing},
    Month = nov,
    Number = {4},
    Pages = {C210--C231},
    Volume = {34},
    Year = {2012}}
```

[CalHelLeV08] D. A. Calhoun, C. Helzel, and R. J. LeVeque. Logically Rectangular Grids and Finite Volume Methods for PDEs in Circular and Spherical Domains, (http://www.amath.washington.edu/ rjl/pubs/circles/index.html) *SIAM Review* 50 (2008), 723-752.

[LeVeque09] R. J. LeVeque. Python Tools for Reproducible Research on Hyperbolic Problems (http://www.amath.washington.edu/ rjl/pubs/cise09/index.html) *Computing in Science and Engineering (CiSE)* 11(2009), pp. 19-27.

[LeVYon03] R. J. LeVeque and Darryl H. Yong. Solitary Waves in Layered Nonlinear Media, *SIAM J. Appl. Math* 63 (2003) pp. 1539-1560.

[Mandli13a] Mandli, K. T. *A Numerical Method for the Two Layer Shallow Water Equations with Dry States. Ocean Modelling* 72, 80–91 (2013).

```
@article{Mandli:2013it,
         author = {Mandli, Kyle T},
         title = {{A Numerical Method for the Two Layer Shallow Water Equations with Dry States}},
         journal = {Ocean Modelling},
         year = {2013},
         volume = {72},
         pages = {80--91},
         month = aug
         }
```

[Mandli13b] Mandli, K. T. & Dawson, C. N. *Adaptive Mesh Refinement for Storm Surge. Ocean Modelling* 75, 36–50 (2014).

```
@article{Mandli:ws,
         author = {Mandli, Kyle T and Dawson, Clint N},
         title = {{Adaptive Mesh Refinement for Storm Surge}},
         journal = {Ocean Modelling},
         year = {2014},
         volume = {75},
         pages = {36--50}}
```

[Okada85] Y. Okada. Surface deformation due to shear and tensile faults in a half-space, Bull. Seism. Soc. Am.* 75 (1985), pp. 1135-1154.

c

p