

FiPy Manual

Release 3.4.1+5.g14449c291

**Jonathan E. Guyer
Daniel Wheeler
James A. Warren**

Materials Science and Engineering Division
and the Center for Theoretical and Computational Materials Science
Material Measurement Laboratory

May 15, 2020

This software was developed by employees of the [National Institute of Standards and Technology \(NIST\)](#), an agency of the Federal Government and is being made available as a public service. Pursuant to [title 17 United States Code Section 105](#), works of [NIST](#) employees are not subject to copyright protection in the United States. This software may be subject to foreign copyright. Permission in the United States and in foreign countries, to the extent that [NIST](#) may hold copyright, to use, copy, modify, create derivative works, and distribute this software and its documentation without fee is hereby granted on a non-exclusive basis, provided that this notice and disclaimer of warranty appears in all copies.

THE SOFTWARE IS PROVIDED "AS IS" WITHOUT ANY WARRANTY OF ANY KIND, EITHER EXPRESSED, IMPLIED, OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY THAT THE SOFTWARE WILL CONFORM TO SPECIFICATIONS, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND FREEDOM FROM INFRINGEMENT, AND ANY WARRANTY THAT THE DOCUMENTATION WILL CONFORM TO THE SOFTWARE, OR ANY WARRANTY THAT THE SOFTWARE WILL BE ERROR FREE. IN NO EVENT SHALL NIST BE LIABLE FOR ANY DAMAGES, INCLUDING, BUT NOT LIMITED TO, DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES, ARISING OUT OF, RESULTING FROM, OR IN ANY WAY CONNECTED WITH THIS SOFTWARE, WHETHER OR NOT BASED UPON WARRANTY, CONTRACT, TORT, OR OTHERWISE, WHETHER OR NOT INJURY WAS SUSTAINED BY PERSONS OR PROPERTY OR OTHERWISE, AND WHETHER OR NOT LOSS WAS SUSTAINED FROM, OR AROSE OUT OF THE RESULTS OF, OR USE OF, THE SOFTWARE OR SERVICES PROVIDED HEREUNDER.

Certain commercial firms and trade names are identified in this document in order to specify the installation and usage procedures adequately. Such identification is not intended to imply recommendation or endorsement by the [National Institute of Standards and Technology](#), nor is it intended to imply that related products are necessarily the best available for the purpose.

Contents

I	Introduction	1
1	Overview	3
2	Installation	7
3	Git practices	15
4	Continuous Integration	17
5	Making a Release	19
6	Solvers	23
7	Viewers	27
8	Using FiPy	29
9	Frequently Asked Questions	45
10	Efficiency	53
11	Theoretical and Numerical Background	55
12	Design and Implementation	63
13	Virtual Kinetics of Materials Laboratory	69
14	Contributors	71
15	Publications	73
16	Presentations	75
17	Change Log	77
18	Glossary	99
II	Examples	101
19	Diffusion Examples	105

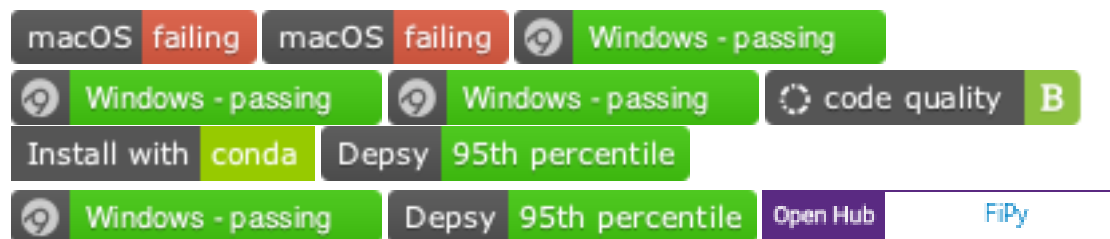
20	Convection Examples	143
21	Phase Field Examples	149
22	Level Set Examples	197
23	Cahn-Hilliard Examples	203
24	Fluid Flow Examples	209
25	Reactive Wetting Examples	215
26	Updating FiPy	221
III	fiPy Package Documentation	231
27	How to Read the Modules Documentation	233
28	fiPy.boundaryConditions package	237
29	fiPy.matrices package	243
30	fiPy.meshes package	245
31	fiPy.solvers package	293
32	fiPy.steppers package	327
33	fiPy.terms package	331
34	fiPy.tests package	377
35	fiPy.tools package	381
36	fiPy.variables package	435
37	fiPy.viewers package	519
	Bibliography	573
	Python Module Index	575
	Index	579

Part I

Introduction

Chapter 1

Overview



FiPy is an object oriented, partial differential equation (PDE) solver, written in *Python*, based on a standard finite volume (FV) approach. The framework has been developed in the Materials Science and Engineering Division (MSED) and Center for Theoretical and Computational Materials Science (CTCMS), in the Material Measurement Laboratory (MML) at the National Institute of Standards and Technology (NIST).

The solution of coupled sets of PDEs is ubiquitous to the numerical simulation of science problems. Numerous PDE solvers exist, using a variety of languages and numerical approaches. Many are proprietary, expensive and difficult to customize. As a result, scientists spend considerable resources repeatedly developing limited tools for specific problems. Our approach, combining the FV method and *Python*, provides a tool that is extensible, powerful and freely available. A significant advantage to *Python* is the existing suite of tools for array calculations, sparse matrices and data rendering.

The *FiPy* framework includes terms for transient diffusion, convection and standard sources, enabling the solution of arbitrary combinations of coupled elliptic, hyperbolic and parabolic PDEs. Currently implemented models include phase field [3] [4] [5] treatments of polycrystalline, dendritic, and electrochemical phase transformations, as well as drug eluting stents [6], reactive wetting [7], photovoltaics [8] and a level set treatment of the electrodeposition process [9].

The latest information about *FiPy* can be found at <http://www.ctcms.nist.gov/fipy/>.

See the latest updates in the *Change Log*.

1.1 Even if you don't read manuals...

...please read *Installation*, *Using FiPy* and *Frequently Asked Questions*, as well as `examples.diffusion.mesh1D`.

1.2 Download and Installation

Please refer to *Installation* for details on download and installation. *FiPy* can be redistributed and/or modified freely, provided that any derivative works bear some notice that they are derived from it, and any modified versions bear some notice that they have been modified.

1.3 Support

You can communicate with the *FiPy* developers and with other users via our [mailing list](#) and we welcome you to use the [issue tracker](#) for bugs, support requests, feature requests and patch submissions <<https://github.com/usnistgov/fipy/issues>>. We also monitor [StackOverflow](#) for questions tagged with “fipy”. We welcome collaborative efforts on this project.

1.3.1 Mailing List

In order to discuss *FiPy* with other users and with the developers, we encourage you to sign up for the mailing list by sending a [subscription email](#):

To: `fipy-request@nist.gov`

Subject: *(optional)*

Body: `subscribe`

Once you are subscribed, you can post messages to the list simply by addressing email to <<mailto:fipy@nist.gov>>. If you are new to mailing lists, you may want to read the following resource about asking effective questions: <http://www.catb.org/~esr/faqs/smart-questions.html>

To get off the list follow the instructions above, but place `unsubscribe` in the text body.

Send `help` in the text body to learn other mailing list configurations you can change.

List Archive

<https://www.mail-archive.com/fipy@nist.gov/>

Copies of messages sent to `fipy@nist.gov` are stored at [The Mail Archive](#).

(note: we have also historically sent copies to <http://dir.gmane.org/gmane.comp.python.fipy>, but the [GMANE](#) site now appears to be [defunct](#).)

1.4 Conventions and Notation

FiPy is driven by *Python* script files that you can view or modify in any text editor. *FiPy* sessions are invoked from a command-line shell, such as **tcsh** or **bash**.

Throughout, text to be typed at the keyboard will appear like `this`. Commands to be issued from an interactive shell will appear:

```
$ like this
```

where you would enter the text (“`like this`”) following the shell prompt, denoted by “\$”.

Text blocks of the form:

```
>>> a = 3 * 4
>>> a
12
>>> if a == 12:
...     print "a is twelve"
...
a is twelve
```

are intended to indicate an interactive session in the *Python* interpreter. We will refer to these as “interactive sessions” or as “doctest blocks”. The text “>>>” at the beginning of a line denotes the *primary prompt*, calling for input of a *Python* command. The text “...” denotes the *secondary prompt*, which calls for input that continues from the line above, when required by *Python* syntax. All remaining lines, which begin at the left margin, denote output from the *Python* interpreter. In all cases, the prompt is supplied by the *Python* interpreter and should not be typed by you.

Warning: *Python* is sensitive to indentation and care should be taken to enter text exactly as it appears in the examples.

When references are made to file system paths, it is assumed that the current working directory is the *FiPy* distribution directory, referred to as the “base directory”, such that:

```
examples/diffusion/steadyState/mesh1D.py
```

will correspond to, *e.g.*:

```
/some/where/FiPy-X.Y/examples/diffusion/steadyState/mesh1D.py
```

Paths will always be rendered using POSIX conventions (path elements separated by “/”). Any references of the form:

```
examples.diffusion.steadyState.mesh1D
```

are in the *Python* module notation and correspond to the equivalent POSIX path given above.

We may at times use a

Note: to indicate something that may be of interest

or a

Warning: to indicate something that could cause serious problems.

Chapter 2

Installation

The *FiPy* finite volume PDE solver relies on several third-party packages. It is *best to obtain and install those first* before attempting to install *FiPy*. This document explains how to install *FiPy*, not how to use it. See *Using FiPy* for details on how to use *FiPy*.

Note: It may be useful to set up a *Development Environment* before beginning the installation process.

2.1 Pre-Installed on Binder

A full *FiPy* installation is available for basic exploration on [Binder](#). The default notebook gives a rudimentary introduction to *FiPy* syntax and, like any [Jupyter Notebook](#) interface, tab completion will help you explore the package interactively.

2.2 Recommended Method

Install with **conda**

Attention: There are many ways to obtain the software packages necessary to run *FiPy*, but the most expedient way is with the [conda](#) package manager. In addition to the scientific *Python* stack, [conda](#) also provides virtual environment management. Keeping separate installations is useful *e.g.* for comparing *Python* 2 and *Python* 3 software stacks, or when the user does not have sufficient privileges to install software system-wide.

In addition to the default packages, many other developers provide “channels” to distribute their own builds of a variety of software. These days, the most useful channel is *conda-forge*, which provides everything necessary to install *FiPy*.

- [install Miniconda](#) on your computer
- run:

```
$ conda create --name <MYFIPYENV> --channel conda-forge python=<PYTHONVERSION>
→ fipy
```

(continues on next page)

(continued from previous page)

Note: This command creates a self-contained `conda` environment and then downloads and populates the environment with the prerequisites for *FiPy* from the `conda-forge` channel at <https://anaconda.org>.

Attention: Windows x86_64 is fully supported, but this does not work on Windows x86_32, as `conda-forge` no longer supports that platform. For Python 2.7.x, you should be able to do:

```
conda create --name <MYFIPYENV> --channel conda-forge python=2.7 numpy scipy_
→matplotlib pyparsing mayavi weave
```

and for Python 3.x, you should be able to do:

```
conda create --name <MYFIPYENV> --channel conda-forge python=3.6 numpy scipy_
→matplotlib pyparsing
```

followed, for either, by:

```
activate <MYFIPYENV>
pip install fipy
```

- enable this new environment with:

```
$ source activate <MYFIPYENV>
```

Note: `$ activate <MYFIPYENV>` on **Windows**

- move on to *Using FiPy*.

Note: On **Linux** and **Mac OS X**, you should have a pretty complete system to run and visualize *FiPy* simulations. On **Windows**, there are fewer packages available via `conda`, particularly amongst the sparse matrix *Solvers*, but the system still should be functional. Significantly, you will need to download and install *Gmsh* manually when using Python 2.7.

Attention: When installed via `conda` or `pip`, *FiPy* will not include its *examples*. These can be obtained by *cloning the repository* or downloading a *compressed archive*.

2.3 Obtaining FiPy

FiPy is freely available for download via `Git` or as a *compressed archive*. Please see *Git usage* for instructions on obtaining *FiPy* with `Git`.

Warning: Keep in mind that if you choose to download the *compressed archive* you will then need to preserve your changes when upgrades to *FiPy* become available (upgrades via `Git` will handle this issue automatically).

2.4 Installing FiPy

Details of the *Required Packages* and links are given below, but for the courageous and the impatient, *FiPy* can be up and running quickly by simply installing the following prerequisite packages on your system:

- *Python*
- *NumPy*
- At least one of the *Solvers*
- At least one of the *Viewers* (*FiPy*'s tests will run without a viewer, but you'll want one for any practical work)

Other *Optional Packages* add greatly to *FiPy*'s capabilities, but are not necessary for an initial installation or to simply run the test suite.

It is not necessary to formally install *FiPy*, but if you wish to do so and you are confident that all of the requisite packages have been installed properly, you can install it by typing:

```
$ pip install fipy
```

or by unpacking the archive and typing:

```
$ python setup.py install
```

at the command line in the base *FiPy* directory. You can also install *FiPy* in “development mode” by typing:

```
$ python setup.py develop
```

which allows the source code to be altered in place and executed without issuing further installation commands.

Alternatively, you may choose not to formally install *FiPy* and to simply work within the base directory instead. In this case or if you are making a non-standard install (without admin privileges), read about setting up your *Development Environment* before beginning the installation process.

2.5 Required Packages

2.5.1 Python

<http://www.python.org/>

FiPy is written in the *Python* language and requires a *Python* installation to run. *Python* comes pre-installed on many operating systems, which you can check by opening a terminal and typing `python`, e.g.:

```
$ python
Python 2.7.15 | ...
...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

If necessary, you can [download](http://www.python.org/download) and install it for your platform <<http://www.python.org/download>>.

Note: *FiPy* requires at least version 2.7.x of *Python*.

Python along with many of *FiPy*'s required and optional packages is available with one of the following distributions.

2.5.2 NumPy

<http://numpy.scipy.org>

Obtain and install the *NumPy* package. *FiPy* requires at least version 1.0 of *NumPy*.

2.6 Optional Packages

2.6.1 Gmsh

<http://www.geuz.org/gmsh/>

Gmsh is an application that allows the creation of irregular meshes. When running in parallel, *FiPy* requires a version of *Gmsh* ≥ 2.5 and < 4.0 .

2.6.2 SciPy

<http://www.scipy.org/>

SciPy provides a large collection of functions and tools that can be useful for running and analyzing *FiPy* simulations. Significantly improved performance has been achieved with the judicious use of C language inlining (see the *Command-line Flags and Environment Variables* section for more details), via the *weave* module.

2.7 Level Set Packages

To use the level set ([1]) components of *FiPy* one of the following is required.

2.7.1 Scikit-fmm

<http://packages.python.org/scikit-fmm/>

Scikit-fmm is a python extension module which implements the fast marching method.

2.7.2 LSMLIB

<http://ktchu.serendipityresearch.org/software/lsmllib/index.html>

The Level Set Method Library (*LSMLIB*) provides support for the serial and parallel simulation of implicit surface and curve dynamics in two- and three-dimensions.

Install *LSMLIB* as per the instructions on the website. Additionally *PyLSMLIB* is required. To install, follow the instructions on the website, <https://github.com/ktchu/LSMLIB/tree/master/pylsmllib#pylsmllib>.

2.8 Development Environment

It is often preferable to not formally install packages in the system directories. The reasons for this include:

- developing or altering the package source code,
- trying out a new package along with its dependencies without violating a working system,
- dealing with conflicting packages and dependencies,
- or not having admin privileges.

To avoid tampering with the system *Python* installation, you can employ one of the utilities that manage packages and their dependencies independently of the system package manager and the system directories. These utilities include *conda*, *Nix*, *Stow*, *Virtualenv* and *Buildout*, amongst others. *Conda* and *Nix* are only ones of these we have the resources to support.

Our preferred development environment is set up with:

```
$ conda create --name <MYFIPYENV> --channel conda-forge python=<PYTHONVERSION> fipy
$ source activate <MYFIPYENV>
$ pip install scikit-fmm
$ conda remove --channel conda-forge --force fipy
$ git clone https://github.com/usnistgov/fipy.git
$ cd fipy
$ python setup.py develop
```

2.9 Git usage

All stages of *FiPy* development are archived in a Git repository at [GitHub](https://github.com/usnistgov/fipy). You can browse through the code at <https://github.com/usnistgov/fipy> and, using a *Git client*, you can download various tagged revisions of *FiPy* depending on your needs.

Attention: Be sure to follow *Installation* to obtain all the prerequisites for *FiPy*.

2.9.1 Git client

A *git* client application is needed in order to fetch files from our repository. This is provided on many operating systems (try executing `which git`) but needs to be installed on many others. The sources to build Git, as well as links to various pre-built binaries for different platforms, can be obtained from <http://git-scm.com/>.

2.9.2 Git branches

In general, most users will not want to download the very latest state of *FiPy*, as these files are subject to active development and may not behave as desired. Most users will not be interested in particular version numbers either, but instead with the degree of code stability. Different branches are used to indicate different stages of *FiPy* development. For the most part, we follow a *successful Git branching model*. You will need to decide on your own risk tolerance when deciding which stage of development to track.

A fresh copy of the *FiPy* source code can be obtained with:

```
$ git clone https://github.com/usnistgov/fipy.git
```

An existing Git checkout of FiPy can be shifted to a different *<branch>* of development by issuing the command:

```
$ git checkout <branch>
```

in the base directory of the working copy. The main branches for FiPy are:

master designates the (ready to) release state of FiPy. This code is stable and should pass all of the tests (or should be documented that it does not).

Past releases of FiPy are tagged as

x.y.z Any released version of FiPy will be designated with a fixed tag: The current version of FiPy is 3.4.1+5.g14449c291. (Legacy version-x_y_z tags are retained for historical purposes, but won't be added to.)

Tagged releases can be found with:

```
$ git tag --list
```

Any other branches will not generally be of interest to most users.

Note: For some time now, we have done all significant development work on branches, only merged back to *master* when the tests pass successfully. Although we cannot guarantee that *master* will never be broken, you can always check our *Continuous Integration* status to find the most recent revision that it is running acceptably.

Historically, we merged to *develop* before merging to *master*. We no longer do this, although for time being, *develop* is kept synchronized with *master*. In a future release, we will remove the *develop* branch altogether.

For those who are interested in learning more about Git, a wide variety of online sources are available, starting with the [official Git website](#). The [Pro Git book](#) [2] is particularly instructive.

2.10 Nix

2.10.1 Nix Installation

FiPy now has a [Nix](#) expression for installing *FiPy* using [Nix](#). [Nix](#) is a powerful package manager for Linux and other Unix systems that makes package management reliable and reproducible. The recipe works on both Linux and Mac OS X.

Getting Started with Nix

There are a number of tutorials on getting started with [Nix](#). The page that I used when getting started is on the Blog of the HPC team of GRICAD,

<https://gricad.github.io/calcul/nix/tuto/2017/07/04/nix-tutorial.html>

I also made my own notes,

<https://github.com/wd15/nixes/blob/master/NIX-NOTES.md>

which are a succinct steps that I use when setting up a new system with Nix.

Installing

Once you have a working Nix installation use:

```
$ nix-shell --pure
```

in the base *FiPy* directory to install *FiPy* with Python 3 by default. Modify the *shell.nix* file to use another version of Python. *nix-shell* drops the user into a shell with a working version of *FiPy*. To test your installation use:

```
$ nix-shell --pure --command "python setup.py test"
```

Note: *Trilinos* is currently not available as part of the Nix *FiPy* installation.

Additional Packages

To install additional packages available from *Nixpkgs* include them in the *nativeBuildInputs* list in *shell.nix*.

Using Pip

Packages unavailable from Nix can be installed using *Pip*. In this case, the installation has been set up so that the Nix shell knows about a *.local* directory in the base *FiPy* directory used by *Pip* for installation. So, for example, to install the *toolz* package from within the Nix shell use:

```
$ pip install --user toolz
```

The *.local* directory will persist after the Nix shell has been closed.

Chapter 3

Git practices

Refer to *Git usage* for the current branching conventions.

3.1 Branches

Whether fixing a bug or adding a feature, all work on FiPy should be conducted on a branch and submitted as a [pull request](#). If there is already a reported [GitHub issue](#), name the branch accordingly:

```
$ BRANCH=issue12345-Summary_of_what_branch_addresses  
$ git checkout -b $BRANCH master
```

Edit and add to branch:

```
$ emacs ...  
$ git commit -m "refactoring_stage_A"  
$ emacs ...  
$ git commit -m "refactoring_stage_B"
```

3.1.1 Merging changes from master to the branch

Make sure master is up to date:

```
$ git fetch origin
```

Merge updated state of master to the branch:

```
$ git diff origin/master  
$ git merge origin/master
```

Resolve any conflicts and test:

```
$ python setup.py test
```

3.1.2 Submit branch for code review

If necessary, fork the [fipy](#) repository.

Add a “remote” link to your fork:

```
$ git remote add <MYFORK> <MYFORKURL>
```

Push the code to your fork on [GitHub](#):

```
$ git push <MYFORK> $BRANCH
```

Now [create a pull request](#) from your `$BRANCH` against the master branch of `usnistgov/fipy`. The [pull request](#) should initiate automated testing. Check the [Continuous Integration](#) status. Fix (or, if absolutely necessary, document) any failures.

Note: If your branch is still in an experimental state, but you would like to check its impact on the tests, you may prepend “WIP:” to your [pull request](#) title. This will prevent your branch from being merged before it’s complete, but will allow the automated tests to run.

Please be respectful of the [Continuous Integration](#) resources and do the bulk of your testing on your local machine or against your own [Continuous Integration](#) accounts (if you have a lot of testing to do, before you create a [pull request](#), push your branch to your own [fork](#) and enable the [Continuous Integration](#) services there).

You can avoid testing individual commits by adding “[skip ci]” to the commit message title.

When your [pull request](#) is ready and successfully passes the tests, you can [request a pull request review](#) or send a message to the mailing list about it if you like, but the FiPy developers should automatically see the pull request and respond to it without further action on your part.

3.1.3 Refactoring complete: merge branch to master

Attention: Administrators Only!

Use the [GitHub](#) interface to [merge the pull request](#).

Note: Particularly for branches with a long development history, consider doing a [Squash and merge](#).

Chapter 4

Continuous Integration

We use three different cloud services for continuous integration (CI). Each service offers unique capabilities, but this also serves to distribute the load.



4.1 Linux

Linux builds are performed on [CircleCI](#). This CI is configured in `{FiPySource}/.circleci/config.yml`.

4.2 Mac OS X

Mac OS X builds are performed on [TravisCI](#). This CI is configured in `{FiPySource}/.travis.yml`.

4.3 Windows

Windows builds are performed on [AppVeyor](#). This CI is configured in `{FiPySource}/.appveyor.yml`.

Chapter 5

Making a Release

Attention: Administrators Only!

5.1 Source

Make sure `master` is ready for release:

```
$ git checkout master
```

Check the [issue](#) list and update the *Change Log*:

```
$ git commit CHANGELOG.txt -m "REL: update new features for release"
```

Note: You can use:

```
$ python setup.py changelog --after=<x.y>
```

or:

```
$ python setup.py changelog --milestone=<x.z>
```

to obtain a ReST-formatted list of every [GitHub pull request](#) and [issue](#) closed since the last release.

Particularly for major and feature releases, be sure to curate the output so that it's clear what's a big deal about this release. Sometimes a [pull request](#) will be redundant to an [issue](#), e.g., “Issue123 blah blah”. If the [pull request](#) fixes a bug, preference is given to the corresponding [issue](#) under **Fixes**. Alternatively, if the [pull request](#) adds a new feature, preference is given to the item under **Pulls** and corresponding [issue](#) should be removed from **Fixes**. If appropriate, be sure to move the “Thanks to @mention” to the appropriate [issue](#) to recognize outside contributors.

Attention: Requires [PyGithub](#) and [Pandas](#).

Attention: If *Continuous Integration* doesn't show all green boxes for this release, make sure to add appropriate notes in `README.txt` or `INSTALLATION.txt`!

5.2 Release from master

```
$ git checkout master
```

Resolve any conflicts and tag the release as appropriate (see *Git practices* above):

```
$ git tag --annotate x.y master
```

Push the tag to [GitHub](#):

```
$ git push --tags origin master
```

Upon successful completion of the *Continuous Integration* systems, fetch the tagged build products and place in `dist/`:

FiPy-x.y.tar.gz From [CircleCI build-binaries](#) Artifacts

`~/project/documentation/_build/latex/fipy.pdf` From [CircleCI build-36-docs](#) Artifacts

~/project/html.tar.gz From [CircleCI build-36-docs](#) Artifacts

FiPy-x.y.win32.zip From [AppVeyor](#) Artifacts

From the *FiPySource* directory, unpack `dist/html.tar.gz` into `file:documentation/_build` with:

```
$ tar -xzf dist/html.tar.gz -C documentation/_build
```

5.3 Upload

Attach `dist/FiPy-x.y.tar.gz`, `dist/FiPy-x.y.win32.zip`, and `documentation/_build/latex/fipy-x.y.pdf` to a [GitHub release](#) associated with tag `x.y`.

Upload the build products to PyPI with [twine](#):

```
$ twine upload dist/FiPy-${FIPY_VERSION}.tar.gz
```

Upload the web site to CTCMS

```
$ export FIPY_WWWHOST=bunter:/u/WWW/wd15/fipy
$ export FIPY_WWWACTIVATE=updatewww
$ python setup.py upload_products --html
```

Warning: Some versions of `rsync` on Mac OS X have caused problems when they try to upload erroneous `\rsrc` directories. Version 2.6.2 does not have this problem.

5.4 Update conda-forge feedstock

Once you push the tag to [GitHub](#), the [fipy-feedstock](#) should automatically receive a pull request. Review and amend this pull request as necessary and ask the [feedstock maintainers](#) to merge it.

This automated process only runs once an hour, so if you don't wish to wait (or it doesn't trigger for some reason), you can manually generate a pull request to update the [fipy-feedstock](#) with:

- revised version number
- revised sha256 (use `openssl dgst -sha256 /path/to/fipy-x.y.tar.gz`)
- reset build number to 0

5.5 Announce

Make an announcement to fipy@nist.gov

Chapter 6

Solvers

FiPy requires either *Pysparse*, *SciPy* or *Trilinos* to be installed in order to solve linear systems. From our experiences, *FiPy* runs most efficiently in serial when *Pysparse* is the linear solver. *Trilinos* is the most complete of the three solvers due to its numerous preconditioning and solver capabilities and it also allows *FiPy* to *run in parallel*. Although less efficient than *Pysparse* and less capable than *Trilinos*, *SciPy* is a very popular package, widely available and easy to install. For this reason, *SciPy* may be the best linear solver choice when first installing and testing *FiPy* (and it is the only viable solver under *Python 3.x*).

FiPy chooses the solver suite based on system availability or based on the user supplied *Command-line Flags and Environment Variables*. For example, passing `--no-pysparse`:

```
$ python -c "from fipy import *; print DefaultSolver" --no-pysparse
<class 'fipy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver'>
```

uses a *Trilinos* solver. Setting `FIPY_SOLVERS` to `scipy`:

```
$ FIPY_SOLVERS=scipy
$ python -c "from fipy import *; print DefaultSolver"
<class 'fipy.solvers.scipy.linearLUSolver.LinearLUSolver'>
```

uses a *SciPy* solver. Suite-specific solver classes can also be imported and instantiated overriding any other directives. For example:

```
$ python -c "from fipy.solvers.scipy import DefaultSolver; \
> print DefaultSolver" --no-pysparse
<class 'fipy.solvers.scipy.linearLUSolver.LinearLUSolver'>
```

uses a *SciPy* solver regardless of the command line argument. In the absence of *Command-line Flags and Environment Variables*, *FiPy*'s order of precedence when choosing the solver suite for generic solvers is *Pysparse* followed by *Trilinos*, *PyAMG* and *SciPy*.

6.1 PETSc

<https://www.mcs.anl.gov/petsc>

PETSc (the Portable, Extensible Toolkit for Scientific Computation) is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. It employs the *MPI* standard for all message-passing communication (see *Solving in Parallel* for more details).

Attention: *PETSc* requires the *petsc4py* and *mpi4py* interfaces.

Note: *FiPy* does not implement any preconditioner objects for *PETSc*. Simply pass one of the *PCType* strings in the *precon=* argument when declaring the solver.

6.2 Pysparse

<http://pysparse.sourceforge.net>

Pysparse is a fast serial sparse matrix library for *Python*. It provides several sparse matrix storage formats and conversion methods. It also implements a number of iterative solvers, preconditioners, and interfaces to efficient factorization packages. The only requirement to install and use *Pysparse* is *NumPy*.

Warning: *FiPy* requires version 1.0 or higher of *Pysparse*.

6.3 SciPy

<http://www.scipy.org/>

The `scipy.sparse` module provides a basic set of serial Krylov solvers, but no preconditioners.

6.4 PyAMG

<http://code.google.com/p/pyamg/>

The *PyAMG* package provides adaptive multigrid preconditioners that can be used in conjunction with the *SciPy* solvers.

6.5 pyamgx

<https://pyamgx.readthedocs.io/>

The *pyamgx* package is a *Python* interface to the NVIDIA *AMGX* library. *pyamgx* can be used to construct complex solvers and preconditioners to solve sparse sparse linear systems on the GPU.

6.6 Trilinos

<http://trilinos.sandia.gov>

Trilinos provides a more complete set of solvers and preconditioners than either *Pysparse* or *SciPy*. *Trilinos* preconditioning allows for iterative solutions to some difficult problems that *Pysparse* and *SciPy* cannot solve, and it enables parallel execution of *FiPy* (see *Solving in Parallel* for more details).

Attention: Be sure to build or install the *PyTrilinos* interface to *Trilinos*.

Attention: *FiPy* runs more efficiently when *Pysparse* is installed alongside *Trilinos*.

Attention: *Trilinos* is a large software suite with its own set of prerequisites, and can be difficult to set up. It is not necessary for most problems, and is **not** recommended for a basic install of *FiPy*.

Attention: *Trilinos* must be compiled with *MPI* support for *Solving in Parallel*.

Tip: *Trilinos* parallel efficiency is greatly improved by also installing *Pysparse*. If *Pysparse* is not installed, be sure to use the `--no-pysparse` flag.

Note: *Trilinos* solvers frequently give intermediate output that *FiPy* cannot suppress. The most commonly encountered messages are

Gen_Prolongator warning : **Max eigen <= 0.0** which is not significant to *FiPy*.

Aztec status AZ_loss: **loss of precision** which indicates that there was some difficulty in solving the problem to the requested tolerance due to precision limitations, but usually does not prevent the solver from finding an adequate solution.

Aztec status AZ_ill_cond: **GMRES hessenberg ill-conditioned** which indicates that GMRES is having trouble with the problem, and may indicate that trying a different solver or preconditioner may give more accurate results if GMRES fails.

Aztec status AZ_breakdown: **numerical breakdown** which usually indicates serious problems solving the equation which forced the solver to stop before reaching an adequate solution. Different solvers, different preconditioners, or a less restrictive tolerance may help.

Viewers

A viewer is required to see the results of *FiPy* calculations. *Matplotlib* is by far the most widely used *Python* based viewer and the best choice to get *FiPy* up and running quickly. *Matplotlib* is also capable of publication quality plots. *Matplotlib* has only rudimentary 3D capability, which *FiPy* does not attempt to use. *Mayavi* is required for 3D viewing.

7.1 Matplotlib

<http://matplotlib.sourceforge.net>

Matplotlib is a *Python* package that displays publication quality results. It displays both 1D X-Y type plots and 2D contour plots for both structured and unstructured data, but does not display 3D data. It works on all common platforms.

7.2 Mayavi

<http://code.enthought.com/projects/mayavi/>

The *Mayavi* Data Visualizer is a free, easy to use scientific data visualizer. It displays 1D, 2D and 3D data. It is the only *FiPy* viewer available for 3D data. *Matplotlib* is probably a better choice for 1D or 2D viewing.

Mayavi requires *VTK*, which can be difficult to build from source.

Note: MayaVi 1 is no longer supported.

Chapter 8

Using FiPy

This document explains how to use *FiPy* in a practical sense. To see the problems that *FiPy* is capable of solving, you can run any of the scripts in the *examples*.

Note: We strongly recommend you proceed through the *examples*, but at the very least work through *examples.diffusion.mesh1D* to understand the notation and basic concepts of *FiPy*.

We exclusively use either the UNIX command line or *IPython* to interact with *FiPy*. The commands in the *examples* are written with the assumption that they will be executed from the command line. For instance, from within the main *FiPy* directory, you can type:

```
$ python examples/diffusion/mesh1D.py
```

A viewer should appear and you should be prompted through a series of examples.

Note: From within *IPython*, you would type:

```
>>> run examples/diffusion/mesh1D.py
```

In order to customize the examples, or to develop your own scripts, some knowledge of Python syntax is required. We recommend you familiarize yourself with the excellent [Python tutorial \[11\]](#) or with [Dive Into Python \[12\]](#). Deeper insight into Python can be obtained from the [\[13\]](#).

As you gain experience, you may want to browse through the *Command-line Flags and Environment Variables* that affect *FiPy*.

8.1 Testing FiPy

For a general installation, *FiPy* can be tested by running:

```
$ python -c "import fipy; fipy.test()"
```

This command runs all the test cases in *FiPy's modules*, but doesn't include any of the tests in *FiPy's examples*. To run the test cases in both *modules* and *examples*, use:

```
$ python setup.py test
```

Note: You may need to first run:

```
$ python setup.py egg_info
```

for this to work properly.

in an unpacked *FiPy* archive. The test suite can be run with a number of different configurations depending on which solver suite is available and other factors. See *Command-line Flags and Environment Variables* for more details.

FiPy will skip tests that depend on *Optional Packages* that have not been installed. For example, if *Mayavi* and *Gmsh* are not installed, *FiPy* will warn something like:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Skipped 131 doctest examples because `gmsh` cannot be found on the $PATH
Skipped 42 doctest examples because the `tvtk` package cannot be imported
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Although the test suite may show warnings, there should be no other errors. Any errors should be investigated or reported on the [issue tracker](#). Users can see if there are any known problems for the latest *FiPy* distribution by checking *FiPy's automated test display*.

Below are a number of common *Command-line Flags* for testing various *FiPy* configurations.

8.1.1 Parallel Tests

If *FiPy* is configured for *Solving in Parallel*, you can run the tests on multiple processor cores with:

```
$ mpirun -np {# of processors} python setup.py test --trilinos
```

or:

```
$ mpirun -np {# of processors} python -c "import fipy; fipy.test('--trilinos')"
```

8.2 Command-line Flags and Environment Variables

FiPy chooses a default run time configuration based on the available packages on the system. The *Command-line Flags* and *Environment Variables* sections below describe how to override *FiPy*'s default behavior.

8.2.1 Command-line Flags

You can add any of the following case-insensitive flags after the name of a script you call from the command line, e.g.:

```
$ python myFiPyScript --someflag
```

--inline

Causes many mathematical operations to be performed in C, rather than Python, for improved performance. Requires the *weave* package.

--cache

Causes lazily evaluated *FiPy Variable* objects to retain their value.

--no-cache

Causes lazily evaluated *FiPy Variable* objects to always recalculate their value.

The following flags take precedence over the *FIPY_SOLVERS* environment variable:

--pysparse

Forces the use of the *Pysparse* solvers.

--trilinos

Forces the use of the *Trilinos* solvers, but uses *Pysparse* to construct the matrices.

--no-pysparse

Forces the use of the *Trilinos* solvers without any use of *Pysparse*.

--scipy

Forces the use of the *SciPy* solvers.

--pyamg

Forces the use of the *PyAMG* preconditioners in conjunction with the *SciPy* solvers.

--pyamgx

Forces the use of the *pyamgx* solvers.

--lsmlib

Forces the use of the *LSMLIB* level set solver.

--skfmm

Forces the use of the *Scikit-fmm* level set solver.

8.2.2 Environment Variables

You can set any of the following environment variables in the manner appropriate for your shell. If you are not running in a shell (e.g., you are invoking *FiPy* scripts from within *IPython* or *IDLE*), you can set these variables via the `os.environ` dictionary, but you must do so before importing anything from the *fipy* package.

FIPY_DISPLAY_MATRIX

If present, causes the graphical display of the solution matrix of each equation at each call of `solve()` or `sweep()`. Setting the value to “terms” causes the display of the matrix for each *Term* that composes the equation. Requires the *Matplotlib* package. Setting the value to “print” causes the matrix to be printed to the console.

FIPY_INLINE

If present, causes many mathematical operations to be performed in C, rather than Python. Requires the *weave* package.

FIPY_INLINE_COMMENT

If present, causes the addition of a comment showing the Python context that produced a particular piece of *weave* C code. Useful for debugging.

FIPY_SOLVERS

Forces the use of the specified suite of linear solvers. Valid (case-insensitive) choices are “pysparse”, “trilinos”, “no-pysparse”, “scipy” and “pyamg”.

FIPY_VERBOSE_SOLVER

If present, causes the linear solvers to print a variety of diagnostic information.

FIPY_VIEWER

Forces the use of the specified viewer. Valid values are any *<viewer>* from the *fipy.viewers*. *<viewer>*Viewer modules. The special value of *dummy* will allow the script to run without displaying anything.

FIPY_INCLUDE_NUMERIX_ALL

If present, causes the inclusion of all functions and variables of the *numerix* module in the *fipy* namespace.

FIPY_CACHE

If present, causes lazily evaluated *FiPy Variable* objects to retain their value.

8.3 Solving in Parallel

FiPy can use *PETSc* or *Trilinos* to solve equations in parallel. Most mesh classes in *fipy.meshes* can solve in parallel. This includes all “...Grid...” and “...Gmsh...” class meshes. Currently, the only remaining serial-only meshes are *Tri2D* and *SkewedGrid2D*.

Attention: *FiPy* requires *mpi4py* to work in parallel.

Tip: You are strongly advised to force the use of only one *OpenMP* thread with *Trilinos*:

```
$ export OMP_NUM_THREADS=1
```

See *OpenMP Threads vs. MPI Ranks* for more information.

Note: *Trilinos 12.12* has support for *Python 3*, but *PyTrilinos* on *conda-forge* presently only provides 12.10, which is limited to *Python 2.x*. *PETSc* is available for both *Python 3* and *Python 2.7*.

It should not generally be necessary to change anything in your script. Simply invoke:

```
$ mpirun -np {# of processors} python myScript.py --petsc
```

or:

```
$ mpirun -np {# of processors} python myScript.py --trilinos
```

instead of:

```
$ python myScript.py
```

The following plot shows the scaling behavior for multiple processors. We compare solution time vs number of *Slurm* tasks (available cores) for a *Method of Manufactured Solutions Allen-Cahn* problem.

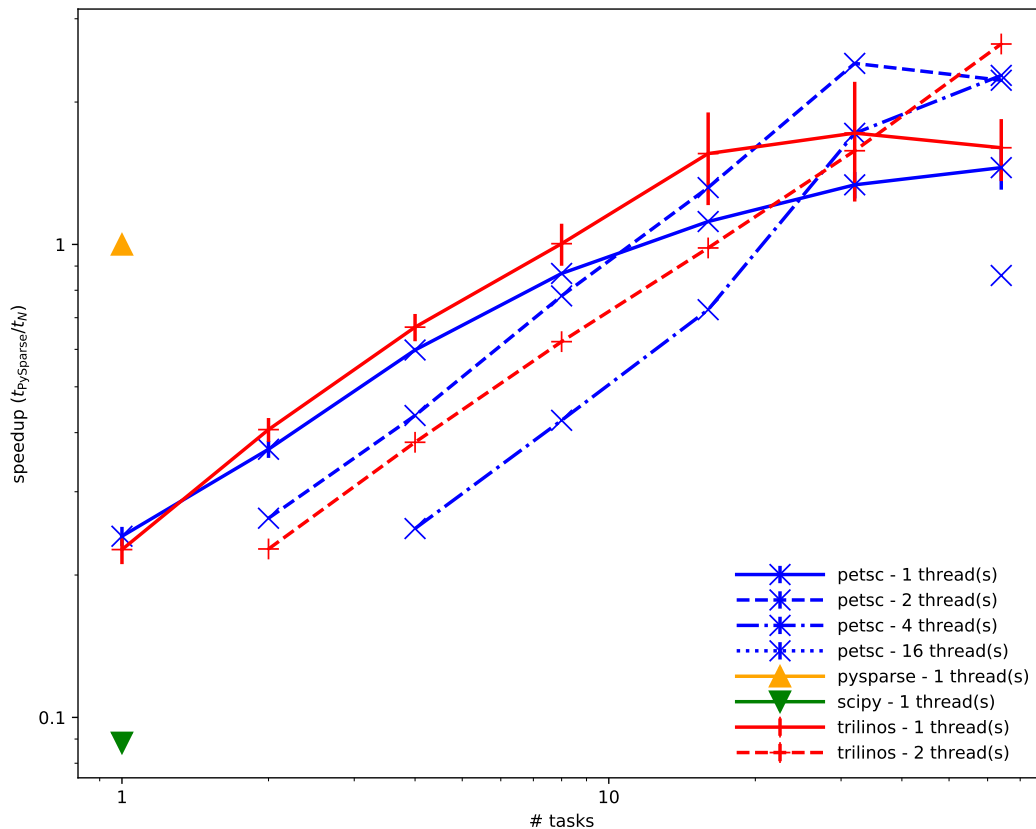


Fig. 1: Scaling behavior of different solver packages

“Speedup” relative to *Pysparse* (bigger numbers are better) versus number of tasks (processes) on a log-log plot. The number of threads per *MPI* rank is indicated by the line style (see legend). *OpenMP* threads \times *MPI* ranks = *Slurm* tasks.

A few things can be observed in this plot:

- Both *PETSc* and *Trilinos* exhibit power law scaling, but the power is only about 0.7. At least one source of this poor scaling is that our “...Grid...” meshes parallelize by dividing the mesh into slabs, which leads to more communication overhead than more compact partitions. The “...Gmsh...” meshes partition more efficiently, but carry more overhead in other ways. We’ll be making efforts to improve the partitioning of the “...Grid...” meshes in a future release.
- PETSc* and *Trilinos* have fairly comparable performance, but lag *Pysparse* by a considerable margin. The *SciPy* solvers are even worse. Some of this discrepancy may be because the different packages are not all doing the same thing. Different solver packages have different default solvers and preconditioners. Moreover, the meaning of the solution tolerance depends on the normalization the solver uses and it is not always obvious

which of several possibilities a particular package employs. We will endeavor to normalize the normalizations in a future release.

- *PETSc* with one thread is faster than with two threads until the number of tasks reaches about 10 and is faster than with four threads until the number of tasks reaches more than 20. *Trilinos* with one thread is faster than with two threads until the number of tasks is more than 30. We don't fully understand the reasons for this, but there may be a *modest* benefit, *when using a large number of cpus*, to allow two to four *OpenMP* threads per *MPI* rank. See *OpenMP Threads vs. MPI Ranks* for caveats and more information.

These results are likely both problem and architecture dependent. You should develop an understanding of the scaling behavior of your own codes before doing “production” runs.

The easiest way to confirm that *FiPy* is properly configured to solve in parallel is to run one of the examples, e.g.,:

```
$ mpirun -np 2 examples/diffusion/mesh1D.py
```

You should see two viewers open with half the simulation running in one of them and half in the other. If this does not look right (e.g., you get two viewers, both showing the entire simulation), or if you just want to be sure, you can run a diagnostic script:

```
$ mpirun -np 3 python examples/parallel.py
```

which should print out:

```
mpi4py           PyTrilinos           petsc4py           FiPy
processor 0 of 3 :: processor 0 of 3 :: processor 0 of 3 :: 5 cells on processor 0 of 3
processor 1 of 3 :: processor 1 of 3 :: processor 1 of 3 :: 7 cells on processor 1 of 3
processor 2 of 3 :: processor 2 of 3 :: processor 2 of 3 :: 6 cells on processor 2 of 3
```

If there is a problem with your parallel environment, it should be clear that there is either a problem importing one of the required packages or that there is some problem with the *MPI* environment. For example:

```
mpi4py           PyTrilinos           petsc4py           FiPy
processor 0 of 3 :: processor 0 of 1 :: processor 0 of 3 :: 10 cells on processor 0
of 1
[my.machine.com:69815] WARNING: There were 4 Windows created but not freed.
processor 1 of 3 :: processor 0 of 1 :: processor 1 of 3 :: 10 cells on processor 0
of 1
[my.machine.com:69814] WARNING: There were 4 Windows created but not freed.
processor 2 of 3 :: processor 0 of 1 :: processor 2 of 3 :: 10 cells on processor 0
of 1
[my.machine.com:69813] WARNING: There were 4 Windows created but not freed.
```

indicates *mpi4py* is properly communicating with *MPI* and is running in parallel, but that *Trilinos* is not, and is running three separate serial environments. As a result, *FiPy* is limited to three separate serial operations, too. In this instance, the problem is that although *Trilinos* was compiled with *MPI* enabled, it was compiled against a different *MPI* library than is currently available (and which *mpi4py* was compiled against). The solution, in this instance, is to solve with *PETSc* or to rebuild *Trilinos* against the active *MPI* libraries.

When solving in parallel, *FiPy* essentially breaks the problem up into separate sub-domains and solves them (somewhat) independently. *FiPy* generally “does the right thing”, but if you find that you need to do something with the entire solution, you can use `var.globalValue`.

Note: One option for debugging in parallel is:

```
$ mpirun -np {# of processors} xterm -hold -e "python -m ipdb myScript.py"
```

8.3.1 OpenMP Threads vs. MPI Ranks

By default, *PETSc* and *Trilinos* spawn as many *OpenMP* threads as there are cores available. This may very well be an intentional optimization, where they are designed to have one *MPI* rank per node of a cluster, so each of the child threads would help with computation but would not compete for I/O resources during ghost cell exchanges and file I/O. However, Python's *Global Interpreter Lock* (GIL) binds all of the child threads to the same core as their parent! So instead of improving performance, each core suffers a heavy overhead from managing those idling threads.

The solution to this is to force these solvers to use only one *OpenMP* thread:

```
$ export OMP_NUM_THREADS=1
```

Because this environment variable affects all processes launched in the current session, you may prefer to restrict its use to *FiPy* runs:

```
$ OMP_NUM_THREADS=1 mpirun -np {# of processors} python myScript.py --trilinos
```

The difference can be extreme. We have observed the *FiPy* test suite to run in just over two minutes when `OMP_NUM_THREADS=1`, compared to over an hour and 23 minutes when *OpenMP* threads are unrestricted. We don't know why, but other platforms do not suffer the same degree of degradation.

Conceivably, allowing these parallel solvers unfettered access to *OpenMP* threads with no *MPI* communication at all could perform as well or better than purely *MPI* parallelization. The plot below demonstrates this is not the case. We compare solution time vs number of *OpenMP* threads for fixed number of slots for a *Method of Manufactured Solutions Allen-Cahn problem*. *OpenMP* threading always slows down *FiPy* performance.

“Speedup” relative to one thread (bigger numbers are better) versus number of threads for 32 *Slurm* tasks on a log-log plot. *OpenMP* threads \times *MPI* ranks = *Slurm* tasks.

See <https://www.mail-archive.com/fipy@nist.gov/msg03393.html> for further analysis.

It may be possible to configure these packages to use only one *OpenMP* thread, but this is not the configuration of the version available from *conda-forge* and building *Trilinos*, at least, is *NotFun™*.

8.4 Meshing with Gmsh

FiPy works with arbitrary polygonal meshes generated by *Gmsh*. *FiPy* provides two wrappers classes (*Gmsh2D* and *Gmsh3D*) enabling *Gmsh* to be used directly from python. The classes can be instantiated with a set of *Gmsh* style commands (see *examples.diffusion.circle*). The classes can also be instantiated with the path to either a *Gmsh* geometry file (*.geo*) or a *Gmsh* mesh file (*.msh*) (see *examples.diffusion.anisotropy*).

As well as meshing arbitrary geometries, *Gmsh* partitions meshes for parallel simulations. Mesh partitioning automatically occurs whenever a parallel communicator is passed to the mesh on instantiation. This is the default setting for all meshes that work in parallel including *Gmsh2D* and *Gmsh3D*.

Note: *FiPy* solution accuracy can be compromised with highly non-orthogonal or non-conjunctional meshes.

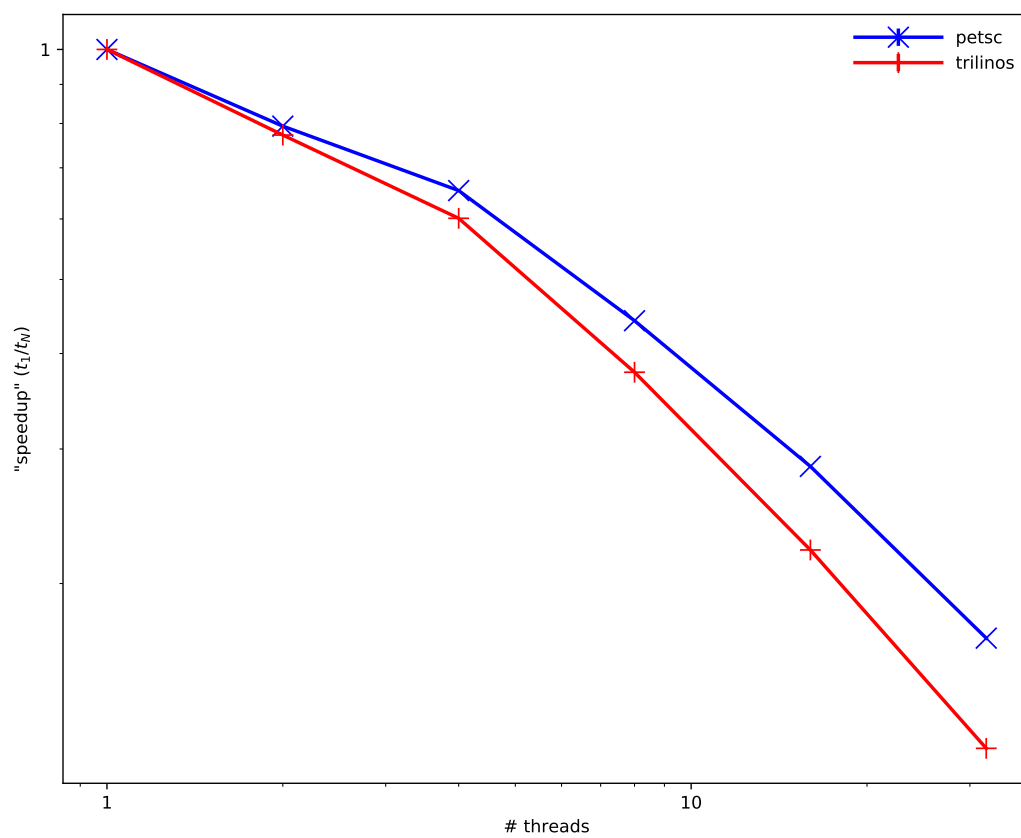


Fig. 2: Effect of having more *OpenMP* threads for each *MPI* rank

8.5 Coupled and Vector Equations

Equations can now be coupled together so that the contributions from all the equations appear in a single system matrix. This results in tighter coupling for equations with spatial and temporal derivatives in more than one variable. In *FiPy* equations are coupled together using the `&` operator:

```
>>> eqn0 = ...
>>> eqn1 = ...
>>> coupledEqn = eqn0 & eqn1
```

The `coupledEqn` will use a combined system matrix that includes four quadrants for each of the different variable and equation combinations. In previous versions of *FiPy* there has been no need to specify which variable a given term acts on when generating equations. The variable is simply specified when calling `solve` or `sweep` and this functionality has been maintained in the case of single equations. However, for coupled equations the variable that a given term operates on now needs to be specified when the equation is generated. The syntax for generating coupled equations has the form:

```
>>> eqn0 = Term00(coeff=..., var=var0) + Term01(coeff=..., var=var1) == source0
>>> eqn1 = Term10(coeff=..., var=var0) + Term11(coeff=..., var=var1) == source1
>>> coupledEqn = eqn0 & eqn1
```

and there is now no need to pass any variables when solving:

```
>>> coupledEqn.solve(dt=..., solver=...)
```

In this case the matrix system will have the form

$$\left(\begin{array}{c|c} \text{Term00} & \text{Term01} \\ \hline \text{Term10} & \text{Term11} \end{array} \right) \left(\begin{array}{c} \text{var0} \\ \text{var1} \end{array} \right) = \left(\begin{array}{c} \text{source0} \\ \text{source1} \end{array} \right)$$

FiPy tries to make sensible decisions regarding each term's location in the matrix and the ordering of the variable column array. For example, if `Term01` is a transient term then `Term01` would appear in the upper left diagonal and the ordering of the variable column array would be reversed.

The use of coupled equation is described in detail in [examples.diffusion.coupled](#). Other examples that demonstrate the use of coupled equations are [examples.phase.binaryCoupled](#), [examples.phase.polyxtalCoupled](#) and [examples.cahnHilliard.mesh2DCoupled](#). As well as coupling equations, true vector equations can now be written in *FiPy* (see [examples.diffusion.coupled](#) for more details).

8.6 Boundary Conditions

8.6.1 Applying fixed value (Dirichlet) boundary conditions

To apply a fixed value boundary condition use the `constrain()` method. For example, to fix `var` to have a value of 2 along the upper surface of a domain, use

```
>>> var.constrain(2., where=mesh.facesTop)
```

Note: The old equivalent `FixedValue` boundary condition is now deprecated.

8.6.2 Applying fixed gradient boundary conditions (Neumann)

To apply a fixed Gradient boundary condition use the `faceGrad.constrain()` method. For example, to fix `var` to have a gradient of $(0,2)$ along the upper surface of a 2D domain, use

```
>>> var.faceGrad.constrain(((0,),(2,)), where=mesh.facesTop)
```

If the gradient normal to the boundary (e.g., $\hat{n} \cdot \nabla \phi$) is to be set to a scalar value of 2, use

```
>>> var.faceGrad.constrain(2 * mesh.faceNormals, where=mesh.exteriorFaces)
```

8.6.3 Applying fixed flux boundary conditions

Generally these can be implemented with a judicious use of `faceGrad.constrain()`. Failing that, an exterior flux term can be added to the equation. Firstly, set the terms' coefficients to be zero on the exterior faces,

```
>>> diffCoeff.constrain(0., mesh.exteriorFaces)
>>> convCoeff.constrain(0., mesh.exteriorFaces)
```

then create an equation with an extra term to account for the exterior flux,

```
>>> eqn = (TransientTerm() + ConvectionTerm(convCoeff)
...       == DiffusionCoeff(diffCoeff)
...       + (mesh.exteriorFaces * exteriorFlux).divergence)
```

where `exteriorFlux` is a rank 1 `FaceVariable`.

Note: The old equivalent `FixedFlux` boundary condition is now deprecated.

8.6.4 Applying outlet or inlet boundary conditions

Convection terms default to a no flux boundary condition unless the exterior faces are associated with a constraint, in which case either an inlet or an outlet boundary condition is applied depending on the flow direction.

8.6.5 Applying spatially varying boundary conditions

The use of spatial varying boundary conditions is best demonstrated with an example. Given a 2D equation in the domain $0 < x < 1$ and $0 < y < 1$ with boundary conditions,

$$\phi = \begin{cases} xy & \text{on } x > 1/2 \text{ and } y > 1/2 \\ \vec{n} \cdot \vec{F} = 0 & \text{elsewhere} \end{cases}$$

where \vec{F} represents the flux. The boundary conditions in *FiPy* can be written with the following code,

```
>>> X, Y = mesh.faceCenters
>>> mask = ((X < 0.5) | (Y < 0.5))
>>> var.faceGrad.constrain(0, where=mesh.exteriorFaces & mask)
>>> var.constrain(X * Y, where=mesh.exteriorFaces & ~mask)
```

then

```
>>> eqn.solve(...)
```

Further demonstrations of spatially varying boundary condition can be found in `examples.diffusion.mesh20x20` and `examples.diffusion.circle`

8.6.6 Applying Robin boundary conditions

The Robin condition applied on the portion of the boundary S_R

$$\hat{n} \cdot (\vec{a}\phi + b\nabla\phi) = g \quad \text{on } S_R$$

can often be substituted for the flux in an equation

$$\begin{aligned} \frac{\partial\phi}{\partial t} &= \nabla \cdot (\vec{a}\phi) + \nabla \cdot (b\nabla\phi) \\ \int_V \frac{\partial\phi}{\partial t} dV &= \int_S \hat{n} \cdot (\vec{a}\phi + b\nabla\phi) dS \\ \int_V \frac{\partial\phi}{\partial t} dV &= \int_{S \notin S_R} \hat{n} \cdot (\vec{a}\phi + b\nabla\phi) dS + \int_{S \in S_R} g dS \end{aligned}$$

At faces identified by `mask`,

```
>>> a = FaceVariable(mesh=mesh, value=..., rank=1)
>>> a.setValue(0., where=mask)
>>> b = FaceVariable(mesh=mesh, value=..., rank=0)
>>> b.setValue(0., where=mask)
>>> g = FaceVariable(mesh=mesh, value=..., rank=0)
>>> eqn = (TransientTerm() == PowerLawConvectionTerm(coeff=a)
...       + DiffusionTerm(coeff=b)
...       + (g * mask * mesh.faceNormals).divergence)
```

When the Robin condition does not exactly map onto the boundary flux, we can attempt to apply it term by term. The Robin condition relates the gradient at a boundary face to the value on that face, however *FiPy* naturally calculates variable values at cell centers and gradients at intervening faces. Using a first order upwind approximation, the boundary value of the variable at face f can be written in terms of the value at the neighboring cell P and the normal gradient at the boundary:

$$\begin{aligned} \phi_f &\approx \phi_P - (\vec{d}_{fP} \cdot \nabla\phi)_f \\ &\approx \phi_P - (\hat{n} \cdot \nabla\phi)_f (\vec{d}_{fP} \cdot \hat{n})_f \end{aligned} \tag{8.1}$$

where \vec{d}_{fP} is the distance vector from the face center to the adjoining cell center. The approximation $(\vec{d}_{fP} \cdot \nabla\phi)_f \approx (\hat{n} \cdot \nabla\phi)_f (\vec{d}_{fP} \cdot \hat{n})_f$ is most valid when the mesh is orthogonal.

Substituting this expression into the Robin condition:

$$\begin{aligned} \hat{n} \cdot (\vec{a}\phi + b\nabla\phi)_f &= g \\ \hat{n} \cdot \left[\vec{a}\phi_P - \vec{a}(\hat{n} \cdot \nabla\phi)_f (\vec{d}_{fP} \cdot \hat{n})_f + b\nabla\phi \right]_f &\approx g \\ (\hat{n} \cdot \nabla\phi)_f &\approx \frac{g_f - (\hat{n} \cdot \vec{a})_f \phi_P}{-\left(\vec{d}_{fP} \cdot \vec{a}\right)_f + b_f} \end{aligned} \tag{8.2}$$

we obtain an expression for the gradient at the boundary face in terms of its neighboring cell. We can, in turn, substitute this back into (8.1)

$$\begin{aligned}\phi_f &\approx \phi_P - \frac{g_f - (\hat{n} \cdot \vec{a})_f \phi_P}{-\left(\vec{d}_{fP} \cdot \vec{a}\right)_f + b_f} \left(\vec{d}_{fP} \cdot \hat{n}\right)_f \\ &\approx \frac{-g_f \left(\hat{n} \cdot \vec{d}_{fP}\right)_f + b_f \phi_P}{-\left(\vec{d}_{fP} \cdot \vec{a}\right)_f + b_f}\end{aligned}\tag{8.3}$$

to obtain the value on the boundary face in terms of the neighboring cell.

Substituting (8.2) into the discretization of the *DiffusionTerm*:

$$\begin{aligned}\int_V \nabla \cdot (\Gamma \nabla \phi) dV &= \int_S \Gamma \hat{n} \cdot \nabla \phi S \\ &\approx \sum_f \Gamma_f (\hat{n} \cdot \nabla \phi)_f A_f \\ &= \sum_{f \notin S_R} \Gamma_f (\hat{n} \cdot \nabla \phi)_f A_f + \sum_{f \in S_R} \Gamma_f (\hat{n} \cdot \nabla \phi)_f A_f \\ &\approx \sum_{f \notin S_R} \Gamma_f (\hat{n} \cdot \nabla \phi)_f A_f + \sum_{f \in S_R} \Gamma_f \frac{g_f - (\hat{n} \cdot \vec{a})_f \phi_P}{-\left(\vec{d}_{fP} \cdot \vec{a}\right)_f + b_f} A_f\end{aligned}$$

An equation of the form

```
>>> eqn = TransientTerm() == DiffusionTerm(coeff=Gamma0)
```

can be constrained to have a Robin condition at faces identified by *mask* by making the following modifications

```
>>> Gamma = FaceVariable(mesh=mesh, value=Gamma0)
>>> Gamma.setValue(0., where=mask)
>>> dPf = FaceVariable(mesh=mesh,
...                     value=mesh._faceToCellDistanceRatio * mesh.cellDistanceVectors)
>>> n = mesh.faceNormals
>>> a = FaceVariable(mesh=mesh, value=..., rank=1)
>>> b = FaceVariable(mesh=mesh, value=..., rank=0)
>>> g = FaceVariable(mesh=mesh, value=..., rank=0)
>>> RobinCoeff = (mask * Gamma0 * n / (-dPf.dot(a) + b)
>>> eqn = (TransientTerm() == DiffusionTerm(coeff=Gamma) + (RobinCoeff * g).divergence
...       - ImplicitSourceTerm(coeff=(RobinCoeff * n.dot(a)).divergence)
```

Similarly, for a *ConvectionTerm*, we can substitute (8.3):

$$\begin{aligned}\int_V \nabla \cdot (\vec{u} \phi) dV &= \int_S \hat{n} \cdot \vec{u} \phi dS \\ &\approx \sum_f (\hat{n} \cdot \vec{u})_f \phi_f A_f \\ &= \sum_{f \notin S_R} (\hat{n} \cdot \vec{u})_f \phi_f A_f + \sum_{f \in S_R} (\hat{n} \cdot \vec{u})_f \frac{-g_f \left(\hat{n} \cdot \vec{d}_{fP}\right)_f + b_f \phi_P}{-\left(\vec{d}_{fP} \cdot \vec{a}\right)_f + b_f} A_f\end{aligned}$$

Note: An expression like the heat flux convection boundary condition $-k \nabla T \cdot \hat{n} = h(T - T_\infty)$ can be put in the form of the Robin condition used above by letting $\vec{a} \equiv h \hat{n}$, $b \equiv k$, and $g \equiv h T_\infty$.

8.6.7 Applying internal “boundary” conditions

Applying internal boundary conditions can be achieved through the use of implicit and explicit sources.

Internal fixed value

An equation of the form

```
>>> eqn = TransientTerm() == DiffusionTerm()
```

can be constrained to have a fixed internal value at a position given by mask with the following alterations

```
>>> eqn = (TransientTerm() == DiffusionTerm()
...         - ImplicitSourceTerm(mask * largeValue)
...         + mask * largeValue * value)
```

The parameter `largeValue` must be chosen to be large enough to completely dominate the matrix diagonal and the RHS vector in cells that are masked. The mask variable would typically be a `CellVariable` Boolean constructed using the cell center values.

Internal fixed gradient

An equation of the form

```
>>> eqn = TransientTerm() == DiffusionTerm(coeff=Gamma0)
```

can be constrained to have a fixed internal gradient magnitude at a position given by mask with the following alterations

```
>>> Gamma = FaceVariable(mesh=mesh, value=Gamma0)
>>> Gamma[mask.value] = 0.
>>> eqn = (TransientTerm() == DiffusionTerm(coeff=Gamma)
...         + DiffusionTerm(coeff=largeValue * mask)
...         - ImplicitSourceTerm(mask * largeValue * gradient
...                               * mesh.faceNormals).divergence)
```

The parameter `largeValue` must be chosen to be large enough to completely dominate the matrix diagonal and the RHS vector in cells that are masked. The mask variable would typically be a `FaceVariable` Boolean constructed using the face center values.

Internal Robin condition

Nothing different needs to be done when *applying Robin boundary conditions* at internal faces.

Note: While we believe the derivations for *applying Robin boundary conditions* are “correct”, they often do not seem to produce the intuitive result. At this point, we think this has to do with the pathology of “internal” boundary conditions, but remain open to other explanations. *FiPy* was designed with diffuse interface treatments (phase field and level set) in mind and, as such, internal “boundaries” do not come up in our own work and have not received much attention.

Warning: The constraints mechanism is not designed to constrain internal values for variables that are being solved by equations. In particular, one must be careful to distinguish between constraining internal cell values during the solve step and simply applying arbitrary constraints to a `CellVariable`. Applying a constraint,

```
>>> var.constrain(value, where=mask)
```

simply fixes the returned value of `var` at `mask` to be `value`. It does not have any effect on the implicit value of `var` at the `mask` location during the linear solve so it is not a substitute for the source term machinations described above. Future releases of *FiPy* may implicitly deal with this discrepancy, but the current release does not.

A simple example can be used to demonstrate this:

```
>>> m = Grid1D(nx=2, dx=1.)
>>> var = CellVariable(mesh=m)
```

We wish to solve $\nabla^2 \phi = 0$ subject to $\phi|_{\text{right}} = 1$ and $\phi|_{x < 1} = 0.25$. We apply a constraint to the faces for the right side boundary condition (which works).

```
>>> var.constrain(1., where=m.facesRight)
```

We create the equation with the source term constraint described above

```
>>> mask = m.x < 1.
>>> largeValue = 1e+10
>>> value = 0.25
>>> eqn = DiffusionTerm() - ImplicitSourceTerm(largeValue * mask) + largeValue *
↪mask * value
```

and the expected value is obtained.

```
>>> eqn.solve(var)
>>> print var
[ 0.25  0.75]
```

However, if a constraint is used without the source term constraint an unexpected solution is obtained

```
>>> var.constrain(0.25, where=mask)
>>> eqn = DiffusionTerm()
>>> eqn.solve(var)
>>> print var
[ 0.25  1. ]
```

although the left cell has the expected value as it is constrained.

FiPy has simply solved $\nabla^2 \phi = 0$ with $\phi|_{\text{right}} = 1$ and (by default) $\hat{n} \cdot \nabla \phi|_{\text{left}} = 0$, giving $\phi = 1$ everywhere, and then subsequently replaced the cells $x < 1$ with $\phi = 0.25$.

8.7 Running under Python 2

Thanks to the `future` package and to the contributions of `pya` and `woodscn`, *FiPy* runs under both *Python 3* and *Python 2.7*, without conversion or modification.

Because *Python* itself will drop support for *Python 2.7* on January 1, 2020 and many of the prerequisites for *FiPy* have pledged to drop support for *Python 2.7* no later than 2020, we have prioritized adding support for better *Python 3* solvers, starting with *petsc4py*.

Because the faster *PySparse* and *Trilinos* solvers are not available under *Python 3*, we will maintain *Python 2.x* support as long as practical. Be aware that the `conda-forge` packages that *FiPy* depends upon are not well-maintained on *Python*

2.x and our support for that configuration is rapidly becoming impractical, despite the present performance benefits. Hopefully, we will learn how to optimize our use of *PETSc* and/or *Trilinos* 12.12 will become available on *conda-forge*.

8.8 Manual

You can view the manual online at <http://www.ctcms.nist.gov/fipy/> or you can download the latest manual from <http://www.ctcms.nist.gov/fipy/download/>. Alternatively, it may be possible to build a fresh copy by issuing the following command in the base directory:

```
$ python setup.py build_docs --pdf --html
```

Note: This mechanism is intended primarily for the developers. At a minimum, you will need at least version 1.7.0 of *Sphinx*, plus all of its prerequisites. We install via *conda*:

```
$ conda install --channel conda-forge sphinx
```

Bibliographic citations require the *sphinxcontrib-bibtex* package:

```
$ pip install sphinxcontrib-bibtex
```

Some documentation uses *numpydoc* styling:

```
$ pip install numpydoc
```

Some embedded figures require *matplotlib*, *pandas*, and *imagemagick*:

```
$ conda install --channel conda-forge matplotlib pandas imagemagick
```

The PDF file requires *SIunits.sty* available, e.g., from *texlive-science*.

Spelling is checked automatically in the course of *Continuous Integration*. If you wish to check manually, you will need *pyspelling*, *hunspell*, and the *libreoffice* dictionaries:

```
$ conda install --channel conda-forge hunspell
$ pip install pyspelling
$ wget -O en_US.aff https://cgit.freedesktop.org/libreoffice/dictionaries/plain/en/
→en_US.aff?id=a4473e06b56bfe35187e302754f6baaa8d75e54f
$ wget -O en_US.dic https://cgit.freedesktop.org/libreoffice/dictionaries/plain/en/en_
→US.dic?id=a4473e06b56bfe35187e302754f6baaa8d75e54f
```


Frequently Asked Questions

9.1 How do I represent an equation in FiPy?

As explained in *Theoretical and Numerical Background*, the canonical governing equation that can be solved by *FiPy* for the dependent *CellVariable* ϕ is

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} + \underbrace{\nabla \cdot (\vec{u}\phi)}_{\text{convection}} = \underbrace{[\nabla \cdot (\Gamma_i \nabla)]^n}_{\text{diffusion}} \phi + \underbrace{S_\phi}_{\text{source}}$$

and the individual terms are discussed in *Discretization*.

A physical problem can involve many different coupled governing equations, one for each variable. Numerous specific examples are presented in Part *Examples*.

9.1.1 Is there a way to model an anisotropic diffusion process or more generally to represent the diffusion coefficient as a tensor so that the diffusion term takes the form $\partial_i \Gamma_{ij} \partial_j \phi$?

Terms of the form $\partial_i \Gamma_{ij} \partial_j \phi$ can be posed in *FiPy* by using a list, tuple rank 1 or rank 2 *FaceVariable* to represent a vector or tensor diffusion coefficient. For example, if we wished to represent a diffusion term with an anisotropy ratio of 5 aligned along the x-coordinate axis, we could write the term as,

```
>>> DiffusionTerm([[[5, 0], [0, 1]]])
```

which represents $5\partial_x^2 + \partial_y^2$. Notice that the tensor, written in the form of a list, is contained within a list. This is because the first index of the list refers to the order of the term not the first index of the tensor (see *Higher order diffusion*). This notation, although succinct can sometimes be confusing so a number of cases are interpreted below.

```
>>> DiffusionTerm([5, 1])
```

This represents the same term as the case examined above. The vector notation is just a short-hand representation for the diagonal of the tensor. Off-diagonals are assumed to be zero.

```
>>> DiffusionTerm([5, 1])
```

This simply represents a fourth order isotropic diffusion term of the form $5(\partial_x^2 + \partial_y^2)^2$.

```
>>> DiffusionTerm([[1, 0], [0, 1]])
```

Nominally, this should represent a fourth order diffusion term of the form $\partial_x^2 \partial_y^2$, but *FiPy* does not currently support anisotropy for higher order diffusion terms so this may well throw an error or give anomalous results.

```
>>> x, y = mesh.cellCenters
>>> DiffusionTerm(CellVariable(mesh=mesh,
...                             value=[[x**2, x * y], [-x * y, -y**2]]))
```

This represents an anisotropic diffusion coefficient that varies spatially so that the term has the form $\partial_x(x^2 \partial_x + xy \partial_y) + \partial_y(-xy \partial_x - y^2 \partial_y) \equiv x \partial_x - y \partial_y + x^2 \partial_x^2 - y^2 \partial_y^2$.

Generally, anisotropy is not conveniently aligned along the coordinate axes; in these cases, it is necessary to apply a rotation matrix in order to calculate the correct tensor values, see [examples.diffusion.anisotropy](#) for details.

9.1.2 How do I represent a ... term that *doesn't* involve the dependent variable?

It is important to realize that, even though an expression may superficially resemble one of those shown in [Discretization](#), if the dependent variable *for that PDE* does not appear in the appropriate place, then that term should be treated as a source.

How do I represent a diffusive source?

If the governing equation for ϕ is

$$\frac{\partial \phi}{\partial t} = \nabla \cdot (D_1 \nabla \phi) + \nabla \cdot (D_2 \nabla \xi)$$

then the first term is a *TransientTerm* and the second term is a *DiffusionTerm*, but the third term is simply an explicit source, which is written in Python as

```
>>> (D2 * xi.faceGrad).divergence
```

Higher order diffusive sources can be obtained by simply nesting the references to *faceGrad* and *divergence*.

Note: We use *faceGrad*, rather than *grad*, in order to obtain a second-order spatial discretization of the diffusion term in ξ , consistent with the matrix that is formed by *DiffusionTerm* for ϕ .

How do I represent a convective source?

The convection of an independent field ξ as in

$$\frac{\partial \phi}{\partial t} = \nabla \cdot (\vec{u} \xi)$$

can be rendered as

```
>>> (u * xi.arithmeticFaceValue).divergence
```

when \vec{u} is a rank-1 *FaceVariable* (preferred) or as

```
>>> (u * xi).divergence
```

if \vec{u} is a rank-1 *CellVariable*.

How do I represent a transient source?

The time-rate-of change of an independent variable ξ , such as in

$$\frac{\partial(\rho_1\phi)}{\partial t} = \frac{\partial(\rho_2\xi)}{\partial t}$$

does not have an abstract form in *FiPy* and should be discretized directly, in the manner of Equation (11.3), as

```
>>> TransientTerm(coeff=rho1) == rho2 * (xi - xi.old) / timeStep
```

This technique is used in *examples.phase.anisotropy*.

9.1.3 What if my term involves the dependent variable, but not where FiPy puts it?

Frequently, viewing the term from a different perspective will allow it to be cast in one of the canonical forms. For example, the third term in

$$\frac{\partial\phi}{\partial t} = \nabla \cdot (D_1\nabla\phi) + \nabla \cdot (D_2\phi\nabla\xi)$$

might be considered as the diffusion of the independent variable ξ with a mobility $D_2\phi$ that is a function of the dependent variable ϕ . For *FiPy*'s purposes, however, this term represents the convection of ϕ , with a velocity $D_2\nabla\xi$, due to the counter-diffusion of ξ , so

```
>>> eq = TransientTerm() == (DiffusionTerm(coeff=D1)
...                          + <Specific>ConvectionTerm(coeff=D2 * xi.faceGrad))
```

Note: With the advent of *Coupled and Vector Equations* in *FiPy* 3.x, it is now possible to represent both terms with *DiffusionTerm*.

9.1.4 What if the coefficient of a term depends on the variable that I'm solving for?

A non-linear coefficient, such as the diffusion coefficient in $\nabla \cdot [\Gamma_1(\phi)\nabla\phi] = \nabla \cdot [\Gamma_0\phi(1 - \phi)\nabla\phi]$ is not a problem for *FiPy*. Simply write it as it appears:

```
>>> diffTerm = DiffusionTerm(coeff=Gamma0 * phi * (1 - phi))
```

Note: Due to the nonlinearity of the coefficient, it will probably be necessary to “sweep” the solution to convergence as discussed in *Iterations, timesteps, and sweeps? Oh, my!*.

9.2 How can I see what I'm doing?

9.2.1 How do I export data?

The way to save your calculations depends on how you plan to make use of the data. If you want to save it for “restart” (so that you can continue or redirect a calculation from some intermediate stage), then you’ll want to “pickle” the *Python* data with the *dump* module. This is illustrated in *examples.phase.anisotropy*, *examples.phase.impingement.mesh40x1*, *examples.phase.impingement.mesh20x20*, and *examples.levelSet.electroChem.howToWriteAScript*.

On the other hand, pickled *FiPy* data is of little use to anything besides *Python* and *FiPy*. If you want to import your calculations into another piece of software, whether to make publication-quality graphs or movies, or to perform some analysis, or as input to another stage of a multiscale model, then you can save your data as an ASCII text file of tab-separated-values with a *TSVViewer*. This is illustrated in *examples.diffusion.circle*.

9.2.2 How do I save a plot image?

Some of the viewers have a button or other mechanism in the user interface for saving an image file. Also, you can supply an optional keyword filename when you tell the viewer to *plot()*, e.g.

```
>>> viewer.plot(filename="myimage.ext")
```

which will save a file named *myimage.ext* in your current working directory. The type of image is determined by the file extension “.ext”. Different viewers have different capabilities:

Matplotlib accepts “.eps”, “.jpg” (Joint Photographic Experts Group), and “.png” (Portable Network Graphics).

Attention: Actually, *Matplotlib* supports different extensions, depending on the chosen *backend*, but our *MatplotlibViewer* classes don’t properly support this yet.

What if I only want the saved file, with no display on screen?

To our knowledge, this is only supported by *Matplotlib*, as is explained in the *Matplotlib FAQ on image backends*. Basically, you need to tell *Matplotlib* to use an “image backend,” such as “Agg” or “Cairo.” Backends are discussed at <http://matplotlib.sourceforge.net/backends.html>.

9.2.3 How do I make a movie?

FiPy has no facilities for making movies. You will need to save individual frames (see the previous question) and then stitch them together into a movie, using one of a variety of different free, shareware, or commercial software packages. The guidance in the *Matplotlib FAQ on movies* should be adaptable to other *Viewers*.

9.2.4 Why doesn't the `Viewer` look the way I want?

FiPy's viewers are utilitarian. They're designed to let you see *something* with a minimum of effort. Because different plotting packages have different capabilities and some are easier to install on some platforms than on others, we have tried to support a range of *Python* plotters with a minimal common set of features. Many of these packages are capable of much more, however. Often, you can invoke the *Viewer* you want, and then issue supplemental commands for the underlying plotting package. The better option is to make a "subclass" of the *FiPy Viewer* that comes closest to producing the image you want. You can then override just the behavior you want to change, while letting *FiPy* do most of the heavy lifting. See `examples.phase.anisotropy` and `examples.phase.polyxtal` for examples of creating a custom *Matplotlib Viewer* class; see `examples.cahnHilliard.sphere` for an example of creating a custom *Mayavi Viewer* class.

9.3 Iterations, timesteps, and sweeps? Oh, my!

Any non-linear solution of partial differential equations is an approximation. These approximations benefit from repetitive solution to achieve the best possible answer. In *FiPy* (and in many similar PDE solvers), there are three layers of repetition.

iterations This is the lowest layer of repetition, which you'll generally need to spend the least time thinking about.

FiPy solves PDEs by discretizing them into a set of linear equations in matrix form, as explained in *Discretization* and *Linear Equations*. It is not always practical, or even possible, to exactly solve these matrix equations on a computer. *FiPy* thus employs "iterative solvers", which make successive approximations until the linear equations have been satisfactorily solved. *FiPy* chooses a default number of iterations and solution tolerance, which you will not generally need to change. If you do wish to change these defaults, you'll need to create a new *Solver* object with the desired number of iterations and solution tolerance, *e.g.*

```
>>> mySolver = LinearPCGSolver(iterations=1234, tolerance=5e-6)
:
:
>>> eq.solve(..., solver=mySolver, ...)
```

Note: The older *Solver* `steps=` keyword is now deprecated in favor of `iterations=` to make the role clearer.

Solver iterations are changed from their defaults in `examples.flow.stokesCavity` and `examples.updating.update0_1tol_0`.

sweeps This middle layer of repetition is important when a PDE is non-linear (*e.g.*, a diffusivity that depends on concentration) or when multiple PDEs are coupled (*e.g.*, if solute diffusivity depends on temperature and thermal conductivity depends on concentration). Even if the *Solver* solves the *linear* approximation of the PDE to absolute perfection by performing an infinite number of iterations, the solution may still not be a very good representation of the actual *non-linear* PDE. If we resolve the same equation *at the same point in elapsed time*, but use the result of the previous solution instead of the previous timestep, then we can get a refined solution to the *non-linear* PDE in a process known as "sweeping."

Note: Despite references to the "previous timestep," sweeping is not limited to time-evolving problems. Non-linear sets of quasi-static or steady-state PDEs can require sweeping, too.

We need to distinguish between the value of the variable at the last timestep and the value of the variable at the last sweep (the last cycle where we tried to solve the *current* timestep). This is done by first modifying the way the variable is created:

```
>>> myVar = CellVariable(..., hasOld=True)
```

and then by explicitly moving the current value of the variable into the “old” value only when we want to:

```
>>> myVar.updateOld()
```

Finally, we will need to repeatedly solve the equation until it gives a stable result. To clearly distinguish that a single cycle will not truly “solve” the equation, we invoke a different method “*sweep()*”:

```
>>> for sweep in range(sweeps):  
...     eq.sweep(var=myVar, ...)
```

Even better than sweeping a fixed number of cycles is to do it until the non-linear PDE has been solved satisfactorily:

```
>>> while residual > desiredResidual:  
...     residual = eq.sweep(var=myVar, ...)
```

Sweeps are used to achieve better solutions in *examples.diffusion.mesh1D*, *examples.phase.simple*, *examples.phase.binaryCoupled*, and *examples.flow.stokesCavity*.

timesteps This outermost layer of repetition is of most practical interest to the user. Understanding the time evolution of a problem is frequently the goal of studying a particular set of PDEs. Moreover, even when only an equilibrium or steady-state solution is desired, it may not be possible to simply solve that directly, due to non-linear coupling between equations or to boundary conditions or initial conditions. Some types of PDEs have fundamental limits to how large a timestep they can take before they become either unstable or inaccurate.

Note: Stability and accuracy are distinctly different. An unstable solution is often said to “blow up”, with radically different values from point to point, often diverging to infinity. An inaccurate solution may look perfectly reasonable, but will disagree significantly from an analytical solution or from a numerical solution obtained by taking either smaller or larger timesteps.

For all of these reasons, you will frequently need to advance a problem in time and to choose an appropriate interval between solutions. This can be simple:

```
>>> timeStep = 1.234e-5  
>>> for step in range(steps):  
...     eq.solve(var=myVar, dt=timeStep, ...)
```

or more elaborate:

```
>>> timeStep = 1.234e-5  
>>> elapsedTime = 0  
>>> while elapsedTime < totalElapsedTime:  
...     eq.solve(var=myVar, dt=timeStep, ...)  
...     elapsedTime += timeStep  
...     timeStep = SomeFunctionOfVariablesAndTime(myVar1, myVar2, elapsedTime)
```

A majority of the examples in this manual illustrate time evolving behavior. Notably, boundary conditions are made a function of elapsed time in *examples.diffusion.mesh1D*. The timestep is chosen based on the expected interfacial velocity in *examples.phase.simple*. The timestep is gradually increased as the kinetics slow down in *examples.cahnHilliard.mesh2DCoupled*.

Finally, we can (and often do) combine all three layers of repetition:

```

>>> myVar = CellVariable(..., hasOld=1)
:
:
>>> mySolver = LinearPCGSolver(iterations=1234, tolerance=5e-6)
:
:
>>> while elapsedTime < totalElapsedTime:
...     myVar.updateOld()
...     while residual > desiredResidual:
...         residual = eq.sweep(var=myVar, dt=timeStep, ...)
...     elapsedTime += timeStep

```

9.4 Why the distinction between `CellVariable` and `FaceVariable` coefficients?

FiPy solves field variables on the cell centers. Transient and source terms describe the change in the value of a field at the cell center, and so they take a `CellVariable` coefficient. Diffusion and convection terms involve fluxes *between* cell centers, and are calculated on the face between two cells, and so they take a `FaceVariable` coefficient.

Note: If you supply a `CellVariable` var when a `FaceVariable` is expected, *FiPy* will automatically substitute `var.arithmeticFaceValue`. This can have undesirable consequences, however. For one thing, the arithmetic face average of a non-linear function is not the same as the same non-linear function of the average argument, *e.g.*, for $f(x) = x^2$,

$$f\left(\frac{1+2}{2}\right) = \frac{9}{4} \neq \frac{f(1) + f(2)}{2} = \frac{5}{2}$$

This distinction is not generally important for smoothly varying functions, but can dramatically affect the solution when sharp changes are present. Also, for many problems, such as a conserved concentration field that cannot be allowed to drop below zero, a harmonic average is more appropriate than an arithmetic average.

If you experience problems (unstable or wrong results, or excessively small timesteps), you may need to explicitly supply the desired `FaceVariable` rather than letting *FiPy* assume one.

9.5 How do I represent boundary conditions?

See the *Boundary Conditions* section for more details.

9.6 What does this error message mean?

ValueError: frames are not aligned This error most likely means that you have provided a `CellVariable` when *FiPy* was expecting a `FaceVariable` (or vice versa).

MA.MA.MAError: Cannot automatically convert masked array to Numeric because data is masked

This not-so-helpful error message could mean a number of things, but the most likely explanation is that the solution has become unstable and is diverging to $\pm\infty$. This can be caused by taking too large a timestep or by using explicit terms instead of implicit ones.

repairing catalog by removing key This message (not really an error, but may cause test failures) can result when using the `weave` package via the `--inline` flag. It is due to a bug in *SciPy* that has been patched in their source repository: <http://www.scipy.org/maillinglists/mailman?fn=scipy-dev/2005-June/003010.html>.

numerix Numeric 23.6 This is neither an error nor a warning. It's just a sloppy message left in *SciPy*: <http://thread.gmane.org/gmane.comp.python.scientific.user/4349>.

9.7 How do I change FiPy's default behavior?

FiPy tries to make reasonable choices, based on what packages it finds installed, but there may be times that you wish to override these behaviors. See the *Command-line Flags and Environment Variables* section for more details.

9.8 How can I tell if I'm running in parallel?

See *Solving in Parallel*.

9.9 Why don't my scripts work anymore?

FiPy has experienced three major API changes. The steps necessary to upgrade older scripts are discussed in *Updating FiPy*.

9.10 What if my question isn't answered here?

Please post your question to the mailing list <<http://www.ctcms.nist.gov/fipy/mail.html>> or file an issue at <<https://github.com/usnistgov/fipy/issues/new>>.

Chapter 10

Efficiency

This section will present results and discussion of efficiency evaluations with *FiPy*. Programming in *Python* allows greater efficiency when designing and implementing new code, but it has some intrinsic inefficiencies during execution as compared with the C or FORTRAN programming languages. These inefficiencies can be minimized by translating sections of code that are used frequently into C.

FiPy has been tested against an in-house phase field code, written at NIST, to model grain growth and subsequent impingement. This problem can be executed by running:

```
$ examples/phase/impingement/mesh20x20.py \  
> --numberOfElements=10000 --numberOfSteps=1000
```

from the base *FiPy* directory. The in-house code was written by Ryo Kobayashi and is used to generate the results presented in [10].

The raw CPU execution times for 10 time steps are presented in the following table. The run times are in seconds and the memory usage is in kilobytes. The Kobayashi code is given the heading of FORTRAN while *FiPy* is run with and without inlining. The memory usage is for *FiPy* simulations with the *--inline*. The *--no-cache* flag is on in all cases for the following table.

Ele- ments	FiPy (s)	FiPy <i>--inline</i> (s)	FORTTRAN (s)	FiPy (KiB)	memory	FORTTRAN (KiB)	memory
100	0.77	0.30	0.0009	39316		772	
400	0.87	0.37	0.0031	39664		828	
1600	1.4	0.65	0.017	40656		1044	
6400	3.7	2.0	0.19	46124		1880	
25600	19	10	1.3	60840		5188	
102400	79	43	4.6	145820		18436	

The plain *Python* version of *FiPy*, which uses `Numeric` for all array operations, is around 17 times slower than the FORTRAN code. Using the *--inline* flag, this penalty is reduced to about 9 times slower.

It is hoped that in future releases of *FiPy* the process of C inlining for `Variable` objects will be automated. This may result in some efficiency gains, greater than we are seeing for this particular problem since all the `Variable` objects will be inlined. Recent analysis has shown that a `Variable` with multiple operations could be up to 6 times faster at calculating its value when inlined.

As presented in the above table, memory usage was also recorded for each *FiPy* simulation. From the table, once base memory usage is subtracted, each cell requires approximately 1.4 kilobytes of memory. The measurement of

the maximum memory spike is hard with dynamic memory allocation, so these figures should only be used as a very rough guide. The FORTRAN memory usage is exact since memory is not allocated dynamically.

10.1 Efficiency comparison between `--no-cache` and `--cache` flags

This table shows results for efficiency tests when using the caching flags. Examples with more variables involved in complex expressions show the largest improvement in memory usage. The `--no-cache` option mainly prevents intermediate variables created due to binary operations from caching their values. This results in large memory gains while not effecting run times substantially. The table below is with `--inline` switched on and with 102400 elements for each case. The `--no-cache` flag is the default option.

Example	time per step <code>--no-cache</code> (s)	time per step <code>--cache</code> (s)	memory per cell <code>--no-cache</code> (KiB)	memory per cell <code>--cache</code> (KiB)
<code>examples.phase. impingement.mesh20x20</code>	4.3	4.1	1.4	2.3
<code>examples.phase. anisotropy</code>	3.5	3.2	1.1	1.9
<code>examples.cahnHilliard. mesh2D</code>	3.0	2.5	1.1	1.4
<code>examples.levelSet. electroChem. simpleTrenchSystem</code>	62	62	2.0	2.8

10.2 Efficiency discussion of Pysparse and Trilinos

Trilinos provides multigrid capabilities which are beneficial for some problems, but has significant overhead compared to Pysparse. The matrix-building step takes significantly longer in Trilinos, and the solvers also have more overhead costs in memory and performance than the equivalent Pysparse solvers. However, the multigrid preconditioning capabilities of Trilinos can, in some cases, provide enough of a speedup in the solution step to make up for the overhead costs. This depends greatly on the specifics of the problem, but is most likely in the cases when the problem is large and when Pysparse cannot solve the problem with an iterative solver and must use an LU solver, while Trilinos can still have success with an iterative method.

Theoretical and Numerical Background

This chapter describes the numerical methods used to solve equations in the *FiPy* programming environment. *FiPy* uses the finite volume method (FVM) to solve coupled sets of partial differential equations (PDEs). For a good introduction to the FVM see Nick Croft's PhD thesis [15], Patankar [20] or Versteeg and Malalasekera [21].

Essentially, the FVM consists of dividing the solution domain into discrete finite volumes over which the state variables are approximated with linear or higher order interpolations. The derivatives in each term of the equation are satisfied with simple approximate interpolations in a process known as discretization. The (FVM) is a popular discretization technique employed to solve coupled PDEs used in many application areas (*e.g.*, Fluid Dynamics).

The FVM can be thought of as a subset of the Finite Element Method (FEM), just as the Finite Difference Method (FDM) is a subset of the FVM. A system of equations fully equivalent to the FVM can be obtained with the FEM using as weighting functions the characteristic functions of FV cells, *i.e.*, functions equal to unity [22]. Analogously, the discretization of equations with the FVM reduces to the FDM on Cartesian grids.

11.1 General Conservation Equation

The equations that model the evolution of physical, chemical and biological systems often have a remarkably universal form. Indeed, PDEs have proven necessary to model complex physical systems and processes that involve variations in both space and time. In general, given a variable of interest ϕ such as species concentration, pH, or temperature, there exists an evolution equation of the form

$$\frac{\partial \phi}{\partial t} = H(\phi, \lambda_i) \quad (11.1)$$

where H is a function of ϕ , other state variables λ_i , and higher order derivatives of all of these variables. Examples of such systems are wide ranging, but include problems that exhibit a combination of diffusing and reacting species, as well as such diverse problems as determination of the electric potential in heart tissue, of fluid flow, stress evolution, and even the Schrödinger equation.

A general conservation equation, solved using *FiPy*, can include any combination of the following terms,

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} + \underbrace{\nabla \cdot (\vec{u}\phi)}_{\text{convection}} = \underbrace{[\nabla \cdot (\Gamma_i \nabla)]^n}_{\text{diffusion}} \phi + \underbrace{S_\phi}_{\text{source}} \quad (11.2)$$

where ρ , \vec{u} and Γ_i represent coefficients in the transient, convection and diffusion terms, respectively. These coefficients can be arbitrary functions of any parameters or variables in the system. The variable ϕ represents the unknown

quantity in the equation. The diffusion term can represent any higher order diffusion-like term, where the order is given by the exponent n . For example, the diffusion term can represent conventional Fickian diffusion [*i.e.*, $\nabla \cdot (\Gamma \nabla \phi)$] when the exponent $n = 1$ or a Cahn-Hilliard term [*i.e.*, $\nabla \cdot (\Gamma_1 \nabla [\nabla \cdot \Gamma_2 \nabla \phi])$] [16] [17] [18]] when $n = 2$, or a phase field crystal term [*i.e.*, $\nabla \cdot (\Gamma_1 \nabla [\nabla \cdot \Gamma_2 \nabla \{\nabla \cdot \Gamma_3 \nabla \phi\}])$] [19]] when $n = 3$, although spectral methods are probably a better approach. Higher order terms ($n > 3$) are also possible, but the matrix condition number becomes quite poor.

11.2 Finite Volume Method

To use the FVM, the solution domain must first be divided into non-overlapping polyhedral elements or cells. A solution domain divided in such a way is generally known as a mesh (as we will see, a *Mesh* is also a *FiPy* object). A mesh consists of vertices, faces and cells (see Figure *Mesh*). In the FVM the variables of interest are averaged over control volumes (CVs). The CVs are either defined by the cells or are centered on the vertices.

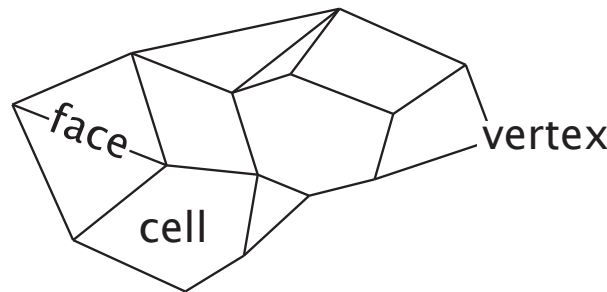


Fig. 1: Mesh

A mesh consists of cells, faces and vertices. For the purposes of *FiPy*, the divider between two cells is known as a face for all dimensions.

11.2.1 Cell Centered FVM (CC-FVM)

In the CC-FVM the CVs are formed by the mesh cells with the cell center “storing” the average variable value in the CV, (see Figure *CV structure for an unstructured mesh*). The face fluxes are approximated using the variable values in the two adjacent cells surrounding the face. This low order approximation has the advantage of being efficient and requiring matrices of low band width (the band width is equal to the number of cell neighbors plus one) and thus low storage requirement. However, the mesh topology is restricted due to orthogonality and conjunctionality requirements. The value at a face is assumed to be the average value over the face. On an unstructured mesh the face center may not lie on the line joining the CV centers, which will lead to an error in the face interpolation. *FiPy* currently only uses the CC-FVM.

11.2.2 Vertex Centered FVM (VC-FVM)

In the VC-FVM, the CV is centered around the vertices and the cells are divided into sub-control volumes that make up the main CVs (see Figure *CV structure for an unstructured mesh*). The vertices “store” the average variable values over the CVs. The CV faces are constructed within the cells rather than using the cell faces as in the CC-FVM. The face fluxes use all the vertex values from the cell where the face is located to calculate interpolations. For this reason, the VC-FVM is less efficient and requires more storage (a larger matrix band width) than the CC-FVM. However, the mesh topology does not have the same restrictions as the CC-FVM. *FiPy* does not have a VC-FVM capability.

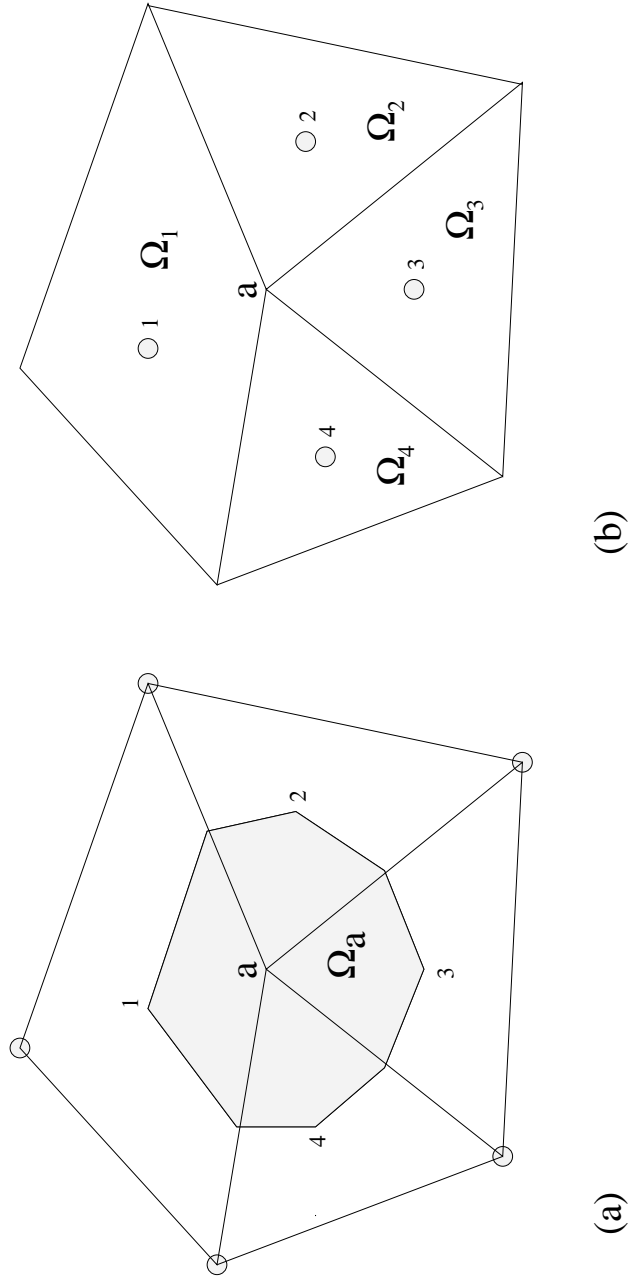


Fig. 2: CV structure for an unstructured mesh
 (a) Ω_a represents a vertex-based CV and (b) Ω_1 , Ω_2 , Ω_3 and Ω_4 represent cell centered CVs.

11.3 Discretization

The first step in the discretization of Equation (11.2) using the CC-FVM is to integrate over a CV and then make appropriate approximations for fluxes across the boundary of each CV. In this section, each term in Equation (11.2) will be examined separately.

11.3.1 Transient Term $\partial(\rho\phi)/\partial t$

For the transient term, the discretization of the integral \int_V over the volume of a CV is given by

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{(\rho_P \phi_P - \rho_P^{\text{old}} \phi_P^{\text{old}}) V_P}{\Delta t} \quad (11.3)$$

where ϕ_P represents the average value of ϕ in a CV centered on a point P and the superscript “old” represents the previous time-step value. The value V_P is the volume of the CV and Δt is the time step size.

This term is represented in *FiPy* as

```
>>> TransientTerm(coeff=rho)
```

11.3.2 Convection Term $\nabla \cdot (\vec{u}\phi)$

The discretization for the convection term is given by

$$\begin{aligned} \int_V \nabla \cdot (\vec{u}\phi) dV &= \int_S (\vec{n} \cdot \vec{u}) \phi dS \\ &\simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f \end{aligned} \quad (11.4)$$

where we have used the divergence theorem to transform the integral over the CV volume \int_V into an integral over the CV surface \int_S . The summation over the faces of a CV is denoted by \sum_f and A_f is the area of each face. The vector \vec{n} is the normal to the face pointing out of the CV into an adjacent CV centered on point A . When using a first order approximation, the value of ϕ_f must depend on the average value in adjacent cell ϕ_A and the average value in the cell of interest ϕ_P , such that

$$\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A.$$

The weighting factor α_f is determined by the convection scheme, described in *Numerical Schemes*.

This term is represented in *FiPy* as

```
>>> <SpecificConvectionTerm>(coeff=u)
```

where `<SpecificConvectionTerm>` can be any of `CentralDifferenceConvectionTerm`, `ExponentialConvectionTerm`, `HybridConvectionTerm`, `PowerLawConvectionTerm`, `UpwindConvectionTerm`, `ExplicitUpwindConvectionTerm`, or `VanLeerConvectionTerm`. The differences between these convection schemes are described in Section *Numerical Schemes*. The velocity coefficient `u` must be a rank-1 *FaceVariable*, or a constant vector in the form of a *Python* list or tuple, e.g. `((1,), (2,))` for a vector in 2D.

11.3.3 Diffusion Term $\nabla \cdot (\Gamma_1 \nabla \phi)$

The discretization for the diffusion term is given by

$$\begin{aligned} \int_V \nabla \cdot (\Gamma \nabla \{\dots\}) dV &= \int_S \Gamma (\vec{n} \cdot \nabla \{\dots\}) dS \\ &\simeq \sum_f \Gamma_f (\vec{n} \cdot \nabla \{\dots\})_f A_f \end{aligned} \quad (11.5)$$

$\{\dots\}$ indicates recursive application of the specified operation on ϕ , depending on the order of the diffusion term. The estimation for the flux, $(\vec{n} \cdot \nabla \{\dots\})_f$, is obtained via

$$(\vec{n} \cdot \nabla \{\dots\})_f \simeq \frac{\{\dots\}_A - \{\dots\}_P}{d_{AP}}$$

where the value of d_{AP} is the distance between neighboring cell centers. This estimate relies on the orthogonality of the mesh, and becomes increasingly inaccurate as the non-orthogonality increases. Correction terms have been derived to improve this error but are not currently included in *FiPy* [15].

This term is represented in *FiPy* as

```
>>> DiffusionTerm(coeff=Gamma1)
```

or

```
>>> ExplicitDiffusionTerm(coeff=Gamma1)
```

ExplicitDiffusionTerm is provided primarily for illustrative purposes, although *examples.diffusion.mesh1D* demonstrates its use in Crank-Nicolson time stepping. *ImplicitDiffusionTerm* is almost always preferred (*DiffusionTerm* is a synonym for *ImplicitDiffusionTerm* to reinforce this preference). One can also create an explicit diffusion term with

```
>>> (Gamma1 * phi.faceGrad).divergence
```

Higher order diffusion

Higher order diffusion expressions, such as $\nabla^4 \phi$ or $\nabla \cdot (\Gamma_1 \nabla (\nabla \cdot (\Gamma_2 \nabla \phi)))$ for Cahn-Hilliard are represented as

```
>>> DiffusionTerm(coeff=(Gamma1, Gamma2))
```

The number of elements supplied for `coeff` determines the order of the term.

11.3.4 Source Term

Any term that cannot be written in one of the previous forms is considered a source S_ϕ . The discretization for the source term is given by,

$$\int_V S_\phi dV \simeq S_\phi V_P. \quad (11.6)$$

Including any negative dependence of S_ϕ on ϕ increases solution stability. The dependence can only be included in a linear manner so Equation (11.6) becomes

$$V_P(S_0 + S_1 \phi_P),$$

where S_0 is the source which is independent of ϕ and S_1 is the coefficient of the source which is linearly dependent on ϕ .

A source term is represented in *FiPy* essentially as it appears in mathematical form, *e.g.*, $3\kappa^2 + b \sin \theta$ would be written

```
>>> 3 * kappa**2 + b * numerix.sin(theta)
```

Note: Functions like `sin()` can be obtained from the `fipy.tools.numerix` module.

Warning: Generally, things will not work as expected if the equivalent function is used from the *NumPy* or *SciPy* library.

If, however, the source depends on the variable that is being solved for, it can be advantageous to linearize the source and cast part of it as an implicit source term, *e.g.*, $3\kappa^2 + \phi \sin \theta$ might be written as

```
>>> 3 * kappa**2 + ImplicitSourceTerm(coeff=sin(theta))
```

11.4 Linear Equations

The aim of the discretization is to reduce the continuous general equation to a set of discrete linear equations that can then be solved to obtain the value of the dependent variable at each CV center. This results in a sparse linear system that requires an efficient iterative scheme to solve. The iterative schemes available to *FiPy* are currently encapsulated in the *Pysparse* and *PyTrilinos* suites of solvers and include most common solvers such as the conjugate gradient method and LU decomposition.

Combining Equations (11.3), (11.4), (11.5) and (11.6), the complete discretization for equation (11.2) can now be written for each CV as

$$\begin{aligned} \frac{\rho_P(\phi_P - \phi_P^{\text{old}})V_P}{\Delta t} + \sum_f (\vec{n} \cdot \vec{u})_f A_f [\alpha_f \phi_P + (1 - \alpha_f) \phi_A] \\ = \sum_f \Gamma_f A_f \frac{(\phi_A - \phi_P)}{d_{AP}} + V_P(S_0 + S_1 \phi_P). \end{aligned}$$

Equation (11.7) is now in the form of a set of linear combinations between each CV value and its neighboring values and can be written in the form

$$a_P \phi_P = \sum_f a_A \phi_A + b_P, \quad (11.7)$$

where

$$\begin{aligned} a_P &= \frac{\rho_P V_P}{\Delta t} + \sum_f (a_A + F_f) - V_P S_1, \\ a_A &= D_f - (1 - \alpha_f) F_f, \\ b_P &= V_P S_0 + \frac{\rho_P V_P \phi_P^{\text{old}}}{\Delta t}. \end{aligned}$$

The face coefficients, F_f and D_f , represent the convective strength and diffusive conductance respectively, and are given by

$$\begin{aligned} F_f &= A_f (\vec{u} \cdot \vec{n})_f, \\ D_f &= \frac{A_f \Gamma_f}{d_{AP}}. \end{aligned}$$

11.5 Numerical Schemes

The coefficients of equation (11.7) must remain positive, since an increase in a neighboring value must result in an increase in ϕ_P to obtain physically realistic solutions. Thus, the inequalities $a_A > 0$ and $a_A + F_f > 0$ must be satisfied. The Péclet number $P_f \equiv F_f/D_f$ is the ratio between convective strength and diffusive conductance. To achieve physically realistic solutions, the inequality

$$\frac{1}{1 - \alpha_f} > P_f > -\frac{1}{\alpha_f} \quad (11.8)$$

must be satisfied. The parameter α_f is defined by the chosen scheme, depending on Equation (11.8). The various differencing schemes are:

the central differencing scheme, where

$$\alpha_f = \frac{1}{2} \quad (11.9)$$

so that $|P_f| < 2$ satisfies Equation (11.8). Thus, the central differencing scheme is only numerically stable for a low values of P_f .

the upwind scheme, where

$$\alpha_f = \begin{cases} 1 & \text{if } P_f > 0, \\ 0 & \text{if } P_f < 0. \end{cases} \quad (11.10)$$

Equation (11.10) satisfies the inequality in Equation (11.8) for all values of P_f . However the solution over predicts the diffusive term leading to excessive numerical smearing (“false diffusion”).

the exponential scheme, where

$$\alpha_f = \frac{(P_f - 1) \exp(P_f) + 1}{P_f(\exp(P_f) - 1)}. \quad (11.11)$$

This formulation can be derived from the exact solution, and thus, guarantees positive coefficients while not over-predicting the diffusive terms. However, the computation of exponentials is slow and therefore a faster scheme is generally used, especially in higher dimensions.

the hybrid scheme, where

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 2, \\ \frac{1}{2} & \text{if } |P_f| < 2, \\ -\frac{1}{P_f} & \text{if } P_f < -2. \end{cases} \quad (11.12)$$

The hybrid scheme is formulated by allowing $P_f \rightarrow \infty$, $P_f \rightarrow 0$ and $P_f \rightarrow -\infty$ in the exponential scheme. The hybrid scheme is an improvement on the upwind scheme, however, it deviates from the exponential scheme at $|P_f| = 2$.

the power law scheme, where

$$\alpha_f = \begin{cases} \frac{P_f - 1}{P_f} & \text{if } P_f > 10, \\ \frac{(P_f - 1) + (1 - P_f/10)^5}{P_f} & \text{if } 0 < P_f < 10, \\ \frac{(1 - P_f/10)^5 - 1}{P_f} & \text{if } -10 < P_f < 0, \\ -\frac{1}{P_f} & \text{if } P_f < -10. \end{cases} \quad (11.13)$$

The power law scheme overcomes the inaccuracies of the hybrid scheme, while improving on the computational time for the exponential scheme.

Warning: *VanLeerConvectionTerm* not mentioned and no discussion of explicit forms.

All of the numerical schemes presented here are available in *FiPy* and can be selected by the user.

Design and Implementation

The goal of *FiPy* is to provide a highly customizable, open source code for modeling problems involving coupled sets of PDEs. *FiPy* allows users to select and customize modules from within the framework. *FiPy* has been developed to address model problems in materials science such as poly-crystals, dendritic growth and electrochemical deposition. These applications all contain various combinations of PDEs with differing forms in conjunction with other unusual physics (over varying length scales) and unique solution procedures. The philosophy of *FiPy* is to enable customization while providing a library of efficient modules for common objects and data types.

12.1 Design

12.1.1 Numerical Approach

The solution algorithms given in the *FiPy* examples involve combining sets of PDEs while tracking an interface where the parameters of the problem change rapidly. The phase field method and the level set method are specialized techniques to handle the solution of PDEs in conjunction with a deforming interface. *FiPy* contains several examples of both methods.

FiPy uses the well-known Finite Volume Method (FVM) to reduce the model equations to a form tractable to linear solvers.

12.1.2 Object Oriented Structure

FiPy is programmed in an object-oriented manner. The benefit of object oriented programming mainly lies in encapsulation and inheritance. Encapsulation refers to the tight integration between certain pieces of data and methods that act on that data. Encapsulation allows parts of the code to be separated into clearly defined independent modules that can be re-applied or extended in new ways. Inheritance allows code to be reused, overridden, and new capabilities to be added without altering the original code. An object is treated by its users as an abstraction; the details of its implementation and behavior are internal.

12.1.3 Test Based Development

FiPy has been developed with a large number of test cases. These test cases are in two categories. The lower level tests operate on the core modules at the individual method level. The aim is that every method within the core installation has a test case. The high level test cases operate in conjunction with example solutions and serve to test global solution algorithms and the interaction of various modules.

With this two-tiered battery of tests, at any stage in code development, the test cases can be executed and errors can be identified. A comprehensive test base provides reassurance that any code breakages will be clearly demonstrated with a broken test case. A test base also aids dissemination of the code by providing simple examples and knowledge of whether the code is working on a particular computer environment.

12.1.4 Open Source

In recent years, there has been a movement to release software under open source and associated unrestricted licenses, especially within the scientific community. These licensing terms allow users to develop their own applications with complete access to the source code and then either contribute back to the main source repository or freely distribute their new adapted version.

As a product of the National Institute of Standards and Technology, the *FiPy* framework is placed in the public domain as a matter of U. S. Federal law. Furthermore, *FiPy* is built upon existing open source tools. Others are free to use *FiPy* as they see fit and we welcome contributions to make *FiPy* better.

12.1.5 High-Level Scripting Language

Programming languages can be broadly lumped into two categories: compiled languages and interpreted (or scripting) languages. Compiled languages are converted from a human-readable text source file to a machine-readable binary application file by a sequence of operations generally referred to as “compiling” and “linking.” The binary application can then be run as many times as desired, but changes will provoke a new cycle of compiling and linking. Interpreted languages are converted from human-readable to machine-readable on the fly, each time the script is executed. Because the conversion happens every time¹, interpreted code is usually slower when running than compiled code. On the other hand, code development and debugging tends to be much easier and fluid when it’s not necessary to wait for compile and link cycles after every change. Furthermore, because the conversion happens in real time, it is possible to have interactive sessions in a scripting language that are not generally possible in compiled languages.

Another distinction, somewhat orthogonal, but closely related, to that between compiled and interpreted languages, is between low-level languages and high-level languages. Low-level languages describe actions in simple terms that are closer to the way the computer actually functions. High-level languages describe actions in more complex and abstract terms that are closer to the way the programmer thinks about the problem at hand. This increased complexity in the meaning of an expression renders simpler code, because the details of the implementation are hidden away in the language internals or in an external library. For example, a low-level matrix multiplication written in C might be rendered as

```
if (Acols != Brows)
    error "these matrix shapes cannot be multiplied";

C = (float *) malloc(sizeof(float) * Bcols * Arows);

for (i = 0; i < Bcols; i++) {
    for (j = 0; j < Arows; j++) {
        C[i][j] = 0;
        for (k = 0; k < Acols; k++) {
```

(continues on next page)

¹ ... neglecting such common optimizations as byte-code interpreters.

(continued from previous page)

```

        C[i][j] += A[i][k] * B[k][j];
    }
}

```

Note that the dimensions of the arrays must be supplied externally, as C provides no intrinsic mechanism for determining the shape of an array. An equivalent high-level construction might be as simple as

```
C = A * B
```

All of the error checking, dimension measuring, and space allocation is handled automatically by low-level code that is intrinsic to the high-level matrix multiplication operator. The high-level code “knows” that matrices are involved, how to get their shapes, and to interpret “*” as a matrix multiplier instead of an arithmetic one. All of this allows the programmer to think about the operation of interest and not worry about introducing bugs in low-level code that is not unique to their application.

Although it needn’t be true, for a variety of reasons, compiled languages tend to be low-level and interpreted languages tend to be high-level. Because low-level languages operate closer to the intrinsic “machine language” of the computer, they tend to be faster at running a given task than high-level languages, but programs written in them take longer to write and debug. Because running performance is a paramount concern, most scientific codes are written in low-level compiled languages like FORTRAN or C.

A rather common scenario in the development of scientific codes is that the first draft hard-codes all of the problem parameters. After a few (hundred) iterations of recompiling and relinking the application to explore changes to the parameters, code is added to read an input file containing a list of numbers. Eventually, the point is reached where it is impossible to remember which parameter comes in which order or what physical units are required, so code is added to, for example, interpret a line beginning with “#” as a comment. At this point, the scientist has begun developing a scripting language without even knowing it. Unfortunately for them, very few scientists have actually studied computer science or actually know anything about the design and implementation of script interpreters. Even if they have the expertise, the time spent developing such a language interpreter is time not spent actually doing research.

In contrast, a number of very powerful scripting languages, such as Tcl, Java, Python, Ruby, and even the venerable BASIC, have open source interpreters that can be embedded directly in an application, giving scientific codes immediate access to a high-level scripting language designed by someone who actually knew what they were doing.

We have chosen to go a step further and not just embed a full-fledged scripting language in the *FiPy* framework, but instead to design the framework from the ground up in a scripting language. While runtime performance is unquestionably important, many scientific codes are run relatively little, in proportion to the time spent developing them. If a code can be developed in a day instead of a month, it may not matter if it takes another day to run instead of an hour. Furthermore, there are a variety of mechanisms for diagnosing and optimizing those portions of a code that are actually time-critical, rather than attempting to optimize all of it by using a language that is more palatable to the computer than to the programmer. Thus *FiPy*, rather than taking the approach of writing the fast numerical code first and then dealing with the issue of user interaction, initially implements most modules in high-level scripting language and only translates to low-level compiled code those portions that prove inefficient².

² A discussion of efficiency issues can be found in *Efficiency*.

12.1.6 Python Programming Language

Acknowledging that several scripting languages offer a number, if not all, of the features described above, we have selected *Python* for the implementation of *FiPy*. Python is

- an interpreted language that combines remarkable power with very clear syntax,
- freely usable and distributable, even for commercial use,
- fully object oriented,
- distributed with powerful automated testing tools (*doctest*, *unittest*),
- actively used and extended by other scientists and mathematicians (*SciPy*, *NumPy*, *ScientificPython*, *Pysparse*).
- easily integrated with low-level languages such as C (*weave*, *blitz*, *PyRex*).

12.2 Implementation

The *Python* classes that make up *FiPy* are described in detail in *fipy Package Documentation*, but we give a brief overview here. *FiPy* is based around three fundamental *Python* classes: *Mesh*, *Variable*, and *Term*. Using the terminology of *Theoretical and Numerical Background*:

A ***Mesh* object** represents the domain of interest. *FiPy* contains many different specific mesh classes to describe different geometries.

A ***Variable* object** represents a quantity or field that can change during the problem evolution. A particular type of *Variable*, called a *CellVariable*, represents ϕ at the centers of the cells of the *Mesh*. A *CellVariable* describes the values of the field ϕ , but it is not concerned with their geometry; that role is taken by the *Mesh*.

An important property of *Variable* objects is that they can describe dependency relationships, such that:

```
>>> a = Variable(value = 3)
>>> b = a * 4
```

does not assign the value 12 to *b*, but rather it assigns a multiplication operator object to *b*, which depends on the *Variable* object *a*:

```
>>> b
(Variable(value = 3) * 4)
>>> a.setValue(5)
>>> b
(Variable(value = 5) * 4)
```

The numerical value of the *Variable* is not calculated until it is needed (a process known as “lazy evaluation”):

```
>>> print b
20
```

A ***Term* object** represents any of the terms in Equation (11.2) or any linear combination of such terms. Early in the development of *FiPy*, a distinction was made between *Equation* objects, which represented all of Equation (11.2), and *Term* objects, which represented the individual terms in that equation. The *Equation* object has since been eliminated as redundant. *Term* objects can be single entities such as a *DiffusionTerm* or a linear combination of other *Term* objects that build up to form an expression such as Equation (11.2).

Beyond these three fundamental classes of *Mesh*, *Variable*, and *Term*, *FiPy* is composed of a number of related classes.

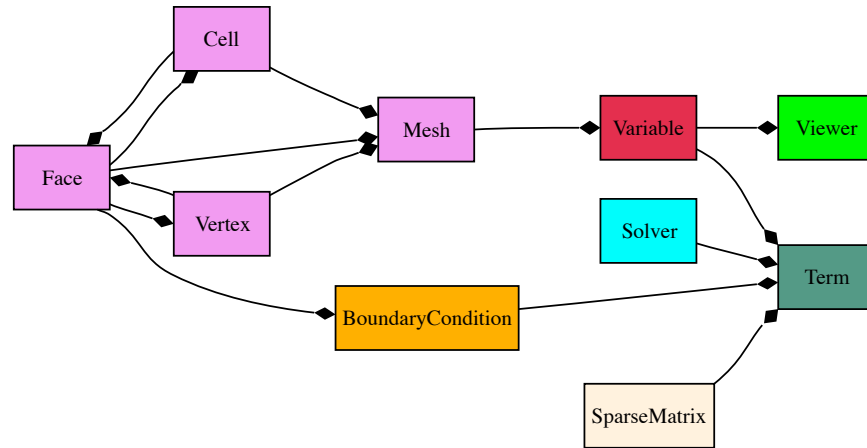


Fig. 1: Primary object relationships in *FiPy*.

A *Mesh* object is composed of cells. Each cell is defined by its bounding faces and each face is defined by its bounding vertices. A *Term* object encapsulates the contributions to the `_SparseMatrix` that defines the solution of an equation. *BoundaryCondition* objects are used to describe the conditions on the boundaries of the *Mesh*, and each *Term* interprets the *BoundaryCondition* objects as necessary to modify the `_SparseMatrix`. An equation constructed from *Term* objects can apply a unique *Solver* to invert its `_SparseMatrix` in the most expedient and stable fashion. At any point during the solution, a *Viewer* can be invoked to display the values of the solved *Variable* objects.

At this point, it will be useful to examine some of the example problems in *Examples*. More classes are introduced in the examples, along with illustrations of their instantiation and use.

Chapter 13

Virtual Kinetics of Materials Laboratory

The VKML is a set of simple *FiPy* examples that simulate basic aspects of kinetics of materials through an interactive Graphical User Interface. The seminal development by Michael Waters and Prof. R. Edwin Garcia of Purdue University includes four examples:

Polycrystalline Growth and Coarsening

simulates the growth, impingement, and coarsening of a random distribution of crystallographically oriented nuclei. The user can control every aspect of the model such as the nuclei radius, the size of the simulation cell, and whether the grains are homogeneously dispersed or only on one wall of the simulation.

Dendritic Growth

simulates the anisotropic solidification of a single solid seed with an N-fold axis of crystallographic symmetry embedded in an undercooled liquid. The user can specify many material aspects of the solidification process, such as the thermal diffusivity and the strength of the surface tension anisotropy. Default values are physical but arbitrary. This model is based on the phase field method and an example shown in the *FiPy* manual.

Two-Dimensional Spinodal Decomposition

simulates the time-dependent segregation of two chemical components and its subsequent coarsening, as presented by John Cahn. The default values are physical but arbitrary.

Three-Dimensional Spinodal Decomposition

has the same functionality as the 2D version, but has an interactive Three-Dimensional viewer.

These modules provide a Graphical User Interface to *FiPy*, and allow you to perform simulations directly through your web browser. This approach to computing removes the need to install the software on your local machine (unless you really want to), allows you to assess current and potential *FiPy* applications and instead you only need a web browser to access it and run it. In other words, you can run these simulations (and simulations like this one) from a Windows machine, a Mac, or a Linux box, and you can also run the modules from Michigan, Boston, Japan, or England: from wherever you are. Moreover, if you close your web browser and leave your calculation running, when you come back a few hours later, your calculation will persist. Additionally, if there is something you want to share with a coworker, wherever he or she might be (e.g., the other side of the planet), you can grant him temporary access to your calculation so that the third party can directly see the output (or specify inputs directly into it, without having to travel to where you are). It is a great way to privately (or publicly) collaborate with other people, especially if the users are in different parts of the world.

The only requirement to run VKML is to register (registration is 100% free) in the [nanoHUB](#).

Contributors

Jon Guyer is a member of the research staff of the Materials Science and Engineering Division in the Material Measurement Laboratory at the National Institute of Standards and Technology. Jon's computational interests are in object-oriented design and in phase field modeling of electrochemistry.

Daniel Wheeler is a guest researcher in the Materials Science and Engineering Division in the Material Measurement Laboratory at the National Institute of Standards and Technology. Daniel's interests are in numerical modeling, finite volume techniques, and level set treatments.

Jim Warren is the leader of the Thermodynamics and Kinetics group in the Materials Science and Engineering Division and Director of the Center for Theoretical and Computational Materials Science of the Material Measurement Laboratory at the National Institute of Standards and Technology. Jim is interested in a variety of problems, including the phase field modeling of solidification, polycrystalline solids, and the electrochemical interface.

Alex Mont developed the *PyxViewer* and the *Gmsh* import and export modules while he was a student at Montgomery Blair High School.

Katie Travis developed the automated *--inline* optimization code for `Variable` objects while she was a SURF student from Smith College.

Max Gibiansky added support for the *Trilinos* solvers while he was a SURF student from Harvey Mudd College

Andrew Reeve added support for anisotropic diffusion coefficients while he was on sabbatical from the University of Maine.

Olivia Buzek worked on adding *Trilinos* parallel computations while she was a SURF student from the University of Maryland

Daniel Stiles worked on adding *Trilinos* parallel computations while he was a student at Montgomery Blair High School.

James O'Beirne added full mesh partitioning using *Gmsh*. James also greatly improved the *Gmsh-FiPy* pipeline. Other contributions include updating *FiPy* to use properties pervasively, deployment of a *Buildbot* server to automate *FiPy* testing and a full refactor of the `Mesh` classes.

Chapter 15

Publications

Attention: If you use FiPy in your research, please cite:

J. E. Guyer, D. Wheeler & J. A. Warren, “FiPy: Partial Differential Equations with Python,” *Computing in Science & Engineering* **11** (3) pp. 6-15 (2009), doi:[10.1109/MCSE.2009.52](https://doi.org/10.1109/MCSE.2009.52). (pdf)

Other publications that have used FiPy. Please contact us to add your work to this list.

- D. Wheeler, and J. A. Warren & W. J. Boettinger, “Modeling the early stages of reactive wetting” *Physical Review E* **82** (5) pp. 051601 (2010), doi:[10.1103/PhysRevE.82.051601](https://doi.org/10.1103/PhysRevE.82.051601).
- R. R. Mohanty, J. E. Guyer & Y. H. Sohn, “Diffusion under temperature gradient: A phase-field model study” *Journal of Applied Physics* **106** (3) pp. 034912 (2009), doi:[10.1063/1.3190607](https://doi.org/10.1063/1.3190607).
- J. A. Warren, T. Pusztai, L. Környei & L. Gránásy, “Phase field approach to heterogeneous crystal nucleation in alloys,” *Physical Review B* **79** 014204 (2009), doi:[10.1103/PhysRevB.79.014204](https://doi.org/10.1103/PhysRevB.79.014204).
- T. P. Moffat, D. Wheeler, S.-K. Kim & D. Josell, “Curvature enhanced adsorbate coverage mechanism for bottom-up superfilling and bump control in Damascene processing,” *Electrochimica Acta* **53** (1) pp. 145-154 (2007), doi:[10.1016/j.electacta.2007.03.025](https://doi.org/10.1016/j.electacta.2007.03.025).
- W. J. Boettinger, J. E. Guyer, C. E. Campbell, G. B. McFadden, “Computation of the Kirkendall velocity and displacement fields in a one-dimensional binary diffusion couple with a moving interface,” *Proceedings of the Royal Society A: Mathematical, Physical & Engineering Sciences* **463** (2088) pp. 3347-3373 (2007), doi:[10.1098/rspa.2007.1904](https://doi.org/10.1098/rspa.2007.1904).
- T. Cickovski, K. Aras, M. Swat, R. M. H. Merks, T. Glimm, H. G. E. Hentschel, M. S. Alber, J. A. Glazier, S. A. Newman & J. A. Izaguirre, “From Genes to Organisms Via the Cell: A Problem-Solving Environment for Multicellular Development,” *Computing in Science & Engineering* **9** (4) pp. 50-60 (2007), doi:[10.1109/MCSE.2007.74](https://doi.org/10.1109/MCSE.2007.74).
- L. Gránásy, T. Pusztai, D. Saylor & J. A. Warren, “Phase Field Theory of Heterogeneous Crystal Nucleation,” *Physical Review Letters* **98** 035703 (2007) [10.1103/PhysRevLett.98.035703](https://doi.org/10.1103/PhysRevLett.98.035703).
- J. Mazur, “Numerical Simulation of Temperature Field in Soil Generated by Solar Radiation,” *Journal de Physique IV France* **137** pp. 317-320 (2006), doi:[10.1051/jp4:2006137061](https://doi.org/10.1051/jp4:2006137061).
- T. P. Moffat, D. Wheeler, S. K. Kim & D. Josell, “Curvature enhanced adsorbate coverage model for electrodeposition,” *Journal of The Electrochemical Society* **153** (2) pp. C127-C132 (2006), [10.1149/1.2165580](https://doi.org/10.1149/1.2165580).

- D. Josell, D. Wheeler & T. P. Moffat, “Gold superfill in submicrometer trenches: Experiment and prediction,” *Journal of The Electrochemical Society* **153** (1) pp. C11-C18 (2006), [10.1149/1.2128765](https://doi.org/10.1149/1.2128765).

Chapter 16

Presentations

We were honored to be invited to deliver a keynote presentation on “[Modeling of Materials with Python](#)” at the [2009 Python for Scientific Computing Conference](#) at Caltech, August 2009.

Other invited talks about FiPy:

- “FiPy: An Open Source Finite Volume PDE Solver Implemented in Python” by J. E. Guyer at the George Mason University Department of Mathematical Sciences, October 2009.
- “FiPy: An Open-Source PDE Solver for Materials Science” by J. E. Guyer at the Center for Devices and Radiological Health of the Food and Drug Administration, June 2009.
- “FiPy: An Open-Source PDE Solver for Materials Science” by J. E. Guyer at GE Global Research, June 2009.
- “FiPy: A PDE Solver for Materials Science” by J. E. Guyer at the SIAM Conference on Computational Science and Engineering, March 2009.
- “FiPy: An Open Source Finite Volume PDE Solver Implemented in Python” by J. E. Guyer in the Open Source Tools for Materials Research and Engineering session of the TMS 2009 Annual Meeting, February 2009.
- “FiPy: A Finite Volume PDE Solver Implemented in Python” by J. E. Guyer in the Computational Materials Research and Education Luncheon Roundtable of the TMS Annual Meeting, February 2009.
- “FiPy - An Object-Oriented Tool for Phase Transformation Simulations Using Python” by J. E. Guyer at Microstructology III, Birmingham, AL, May 2005.
- “FiPy - An Object-Oriented Tool for Phase Transformation Simulations Using Python” by J. E. Guyer at the 2004 MRS Fall Meeting, November 2004.

Chapter 17

Change Log

17.1 Version 3.4.1+5.g14449c291

This release is primarily for compatibility with NumPy 1.18.

17.1.1 Pulls

- Fix documentation ([#711](#))
- build(nix): fix broken plm_rsh_agent error ([#710](#))
- CIs error on deprecation warning ([#708](#))

17.1.2 Fixes

- [#703](#): FORTRAN array ordering is deprecated

17.2 Version 3.4 - 2020-02-06

This release adds support for the *PETSc* solvers for *Solving in Parallel*.

17.2.1 Pulls

- Add support for PETSc solvers ([#701](#))
- Assorted fixes while supporting PETSc ([#700](#)) - Fix print statements for Py3k - Resolve Gmsh issues - Dump only on processor 0 - Only write *timetests* on processor 0 - Fix conda-forge link - Upload PDF - Document *print* option of *FIPY_DISPLAY_MATRIX* - Use legacy numpy formatting when testing individual modules - Switch to matplotlib's built-in symlog scaling - Clean up tests
- Assorted fixes for benchmark 8 ([#699](#)) - Stipulate *-force* option for *conda remove fipy* - Update Miniconda installation url - Replace *_CellVolumeAverageVariable* class with *Variable* expression - Fix output for bad call stack
- Make CircleCI build docs on Py3k ([#698](#))

- Fix link to Nick Croft's thesis (#681)
- Fix NIST header footer (#680)
- Use Nixpkgs version of FiPy expression (#661)
- Update the Nix recipe (#658)

17.2.2 Fixes

- #692: Can't copy example scripts with the command line
- #669: input() deadlock on parallel runs
- #643: Automate release process

17.3 Version 3.3 - 2019-06-28

This release brings support for Python 2 and Python 3 from the same source, without any translation. Thanks to @pya and @woodscn for getting things started.

17.3.1 Pulls

- Automate spell check (#657)
- Fix gmsh on windows (#648)
- Fix sphinx documentation (#647)
- Migrate to Py3k (#645)
- *gmshMesh.py* compatibility with Gmsh > 3.0.6 (#644) Thanks to @xfong.

17.3.2 Fixes

- #655: When Python 2 and 3 are installed, Mayavi wont work. Thanks to @Hendrik410.
- #646: Deprecate develop branch
- #643: Automate release process
- #601: `contents.rst` and `manual.rst` are a recursive mess
- #597: Use GitHub link for the compressed archive in documentation
- #557: *faceGradAverage* is stupid
- #552: documentation integration
- #458: Documentation wrong for precedence of *Lx* and *dx* for *NonUniformGrids*
- #457: Special methods are not included in Sphinx documentation
- #432: Python 3 issues
- #340: Don't upload packages to PyPI, just add the master url

17.4 Version 3.2 - 2019-04-22

This is predominantly a [DevOps](#) release. The focus has been on making FiPy easier to install with [conda](#). It's also possible to install a minimal set of prerequisites with [pip](#). Further, *FiPy* is automatically tested on all major platforms using cloud-based [Continuous Integration](#) ([linux](#) with [CircleCI](#), [macOS](#) with [TravisCI](#), and [Windows](#) with [AppVeyor](#)).

17.4.1 Pulls

- Make badges work in GitHub and pdf (#636)
- Fix Robin errors (#615)
- Issue555 inclusive license (#613)
- Update CIs (#607)
- Add CHANGELOG and tool to generate from issues and pull requests (#600)
- Explain where to get examples (#596)
- spelling corrections using en_US dictionary (#594)
- Remove *SmoothedAggregationSolver* (#593)
- Nix recipe for FiPy (#585)
- Point PyPI to github master tarball (#582)
- Revise Navier-Stokes expression in the viscous limit (#580)
- Update *stokesCavity.py* (#579) Thanks to @Rowin.
- Add *-inline* to TravisCI tests (#578)
- Add support for binder (#577)
- Fix *epetra vector not numarray* (#574)
- add Codacy badge (#572)
- Fix output when PyTrilinos or PyTrilinos version is unavailable (#570) Thanks to @shwina.
- Fix check for PyTrilinos (#569) Thanks to @shwina.
- Adding support for GPU solvers via pyamgx (#567) Thanks to @shwina.
- revise dedication to the public domain (#556)
- Fix tests that don't work in parallel (#550)
- add badges to index and readme (#546)
- Ensure vector is *dtype* float before matrix multiply (#544)
- Revert "Issue534 physical field mishandles compound units" (#536)
- Document boundary conditions (#532)
- Deadlocks and races (#524)
- Make max/min global (#520)
- Add a Gitter chat badge to README.rst (#516) Thanks to @gitter-badger.
- Add TravisCI build recipe (#489)

17.4.2 Fixes

- #631: Clean up `INSTALLATION.rst`
- #628: Problems with the viewer
- #627: Document `OMP_NUM_THREADS`
- #625: `setup.py` should not import `fipy`
- #623: Start using *versioneer*
- #621: Plot *FaceVariable* with `matplotlib`
- #617: Pick 1st Value and last Value of 1D *CellVariable* while running in parallel
- #611: The coefficient cannot be a *FaceVariable* ??
- #610: Anisotropy example: Contour plot displaying in legend of figure !?
- #608: `var.mesh: Property` object not callable...?
- #603: Can't run basic test or examples
- #602: Revise build and release documentation
- #592: is `resources.rst` useful?
- #590: No module named *pyAMGSolver*
- #584: Viewers don't animate in jupyter notebook
- #566: Support for GPU solvers using `pyamgx`
- #565: `pip` install does not work on empty env
- #564: Get green boxes across the board
- #561: Cannot cast array data from `dtype('int64')` to `dtype('int32')` according to the rule *safe*
- #555: inclusive license
- #551: Sphinx spews many warnings:
- #545: Many Py3k failures
- #543: Epetra Vector can't be integer
- #539: `examples/diffusion/explicit/mixedElement.py` is a mess
- #538: badges
- #534: *PhysicalField* mishandles compound units
- #533: `pip` or `conda` installation don't make clear where to get examples
- #531: `drop_tol` argument to `scipy.sparse.linalg.splu` is gone
- #530: `conda` installation instructions not explicit about python version
- #528: `scipy` 1.0.0 incompatibilities
- #525: `conda guyer/pysparse` doesn't run on `osx`
- #513: Stokes example gives wrong equation
- #510: Weave, Scipy and *-inline*
- #509: Unable to use `conda` for installing FiPy in Windows
- #506: Error using spatially varying anisotropic diffusion coefficient

- [#488](#): Gmsh 2.11 breaks *GmshGrids*
- [#435](#): *pip install pyparse* fails with “fatal error: ‘spmatrix.h’ file not found”
- [#434](#): *pip install fipy* fails with “ImportError: No module named ez_setup”

17.5 Version 3.1.3 - 2017-01-17

17.5.1 Fixes

- [#502](#): *gmane* is defunct

17.6 Version 3.1.2 - 2016-12-24

17.6.1 Pulls

- remove *recvobj* from calls to *allgather*, require *sendobj* ([#492](#))
- restore trailing whitespace to expected output of pyparse matrix tests ([#485](#))
- Format version string for pep 440 ([#483](#))
- Provide some documentation for what *_faceToCellDistanceRatio* is and why it’s scalar ([#481](#))
- Strip all trailing white spaces and empty lines at EOF for *.py* and *.r*? ([#479](#)) Thanks to [@pya](#).
- *fipy/meshes/uniformGrid3D.py*: fix *_cellToCellIDs* and more *concatenate()* calls ([#478](#)) Thanks to [@pkgw](#).
- Remove incorrect *axis* argument to *concatenate* ([#477](#))
- Updated to NumPy 1.10 ([#472](#)) Thanks to [@pya](#).
- Some spelling corrections ([#471](#)) Thanks to [@pkgw](#).
- Sort entry points by package name before testing. ([#469](#))
- Update import syntax in examples ([#466](#))
- Update links to prerequisites ([#465](#))
- Correct implementation of *examples.cahnHilliard.mesh2DCoupled*. Fixes ? ([#463](#))
- Fix typeset analytical solution ([#460](#))
- Clear *pdflatex* build errors by removing *Python* from heading ([#459](#))
- purge gist from viewers and optional module lists in *setup.py* ([#456](#))
- Remove deprecated methods that duplicate NumPy ufuncs ([#454](#))
- Remove deprecated Gmsh importers ([#452](#))
- Remove deprecated getters and setters ([#450](#))
- Update links for FiPy developers ([#448](#))
- Render appropriately if in IPython notebook ([#447](#))
- Plot contour in proper axes ([#446](#))
- Robust Gmsh version checking with *distutils.version.StrictVersion* ([#442](#))

- compare gmsh versions as tuples, not floats (#441)
- Corrected two tests (#439) Thanks to @alfrenardi.
- Issue426 fix robin example typo (#431) Thanks to @raybsmith.
- Issue426 fix robin example analytical solution (#429) Thanks to @raybsmith.
- Force *MatplotlibViewer* to display (#428)
- Allow for 2 periodic axes in 3D (#424)
- Bug with Matplotlib 1.4.0 is fixed (#419)

17.6.2 Fixes

- #498: nonlinear source term
- #496: *scipy.LinearBicgstabSolver* doesn't take arguments
- #494: Gmsh call errors
- #493: *Reviewable.io* has read-only access, can't leave comments
- #491: *globalValue* raises error from *mpi4py*
- #484: Pysparse tests fail
- #482: FiPy development version string not compliant with PEP 440
- #476: *setuptools* 18.4 breaks test suite
- #475: *Grid3D* broken by numpy 1.10
- #470: *Mesh3D cellToCellIDs* is broken
- #467: Out-of-sequence *Viewer* imports
- #462: GMSH version ≥ 2.10 incorrectly read by *gmshMesh.py*
- #455: *setup.py* gist warning
- #445: *DendriteViewer* puts contours over color bar
- #443: *MatplotlibViewer* still has problems in IPython notebook
- #440: Use github API to get nicely formatted list of issues
- #438: Failed tests on Mac OS X
- #437: Figure misleading in *examples.cahnHilliard.mesh2DCoupled*
- #433: Links to prerequisites are broken
- #430: Make develop the default branch on Github
- #427: *MatplotlibViewer* don't display
- #425: Links for Warren and Guyer are broken on the web page
- #421: The "limits" argument for *Matplotlib2DGridViewer* does not function
- #416: Updates to reflect move to Github

17.7 Version 3.1.1 - 2015-12-17

17.7.1 Fixes

- #415: *MatplotlibGrid2DViewer* error with Matplotlib version 1.4.0
- #414: *PeriodicGrid3D* supports Only 1 axes of periodicity or all 3, not 2
- #413: Remind users of different types of conservation equations
- #412: Pickling Communicators is unnecessary for Grids
- #408: Implement *PeriodicGrid3D*
- #407: Strange deprecation loop in *reshape()*
- #404: package never gets uploaded to PyPI
- #401: Vector equations are broken when *sweep* is used instead of *solve*.
- #295: Gmsh version must be ≥ 2.0 errors on *zizou*

17.8 Version 3.1 - 2013-09-30

The significant changes since version 3.0 are:

- Level sets are now handled by *LSMLIB* or *Scikit-fmm* solver libraries. These libraries are orders of magnitude faster than the original, *Python*-only prototype.
- The *Matplotlib* *streamplot()* function can be used to display vector fields.
- Version control was switched to the *Git* distributed version control system. This system should make it much easier for *FiPy* users to participate in development.

17.8.1 Fixes

- #398: Home page needs out-of-NIST redirects
- #397: Switch to *sphinxcontrib-bibtex*
- #396: enable google analytics
- #395: Documentation change for Ubuntu install
- #393: *CylindricalNonUniformGrid2D* doesn't make a *FaceVariable* for *exteriorFaces*
- #392: *exit_nist.cgi* deprecated
- #391: Péclet inequalities have the wrong sign
- #388: Windows 64 and numpy's *dtype=int*
- #384: Add support for Matplotlib *streamplot*
- #382: Neumann boundary conditions not clearly documented
- #381: numpy 1.7.1 test failures with *physicalField.py*
- #377: *VanLeerConvectionTerm* MinMod slope limiter is broken
- #376: testing *CommitTicketUpdater*

- #375: NumPy 1.7.0 doesn't have *_formatInteger*
- #373: Bug with numpy 1.7.0
- #372: convection problem with cylindrical grid
- #371: *examples/phase/binary.py* has problems
- #370: FIPY_DISPLAY_MATRIX is broken
- #368: Viewers don't inline well in IPython notebook
- #367: Change documentation to promote use of stackoverflow
- #366: *unOps* can't be pickled
- #365: Rename communicator instances
- #364: Parallel bug in non-uniform grids and conflicting mesh class and factory function names
- #360: NIST CSS changed
- #356: link to mailing list is wrong
- #353: Update Ohloh to point at git repo
- #352: *getVersion()* fails on Py3k
- #350: Gmsh importer can't read mesh elements with no tags
- #347: Include mailing list activity frame on front page
- #339: Fix for test failures on *loki*
- #337: Clean up interaction between dependencies and installation process
- #336: *fipy.test()* and *fipy/test.py* clash
- #334: Make the citation links go to the DOI links
- #333: Web page links seem to be broken
- #331: Assorted errors
- #330: *faceValue* as *FaceCenters* gives inline failures
- #329: Gmsh background mesh doesn't work in parallel
- #326: *Gmsh2D* does not respect background mesh
- #323: *getFaceCenters()* should return a *FaceVariable*
- #319: Explicit convection terms should fail when the equation has no *TransientTerm* (*dt=None*)
- #318: FiPy will not import
- #311: LSMLIB refactor
- #305: *mpirun -np 2 python -Wd setup.py test --trilinos* hanging on sandbox under buildbot
- #297: Remove deprecated gist and gnuplot support
- #291: *efficiency_test* chokes on *liquidVapor2D.py*
- #289: *diffusionTerm._test()* requires Pysparse
- #287: move FiPy to distributed version control
- #275: *mpirun -np 2 python setup.py test --no-pysparse* hangs on *bunter*
- #274: Epetra *Norm2* failure in parallel

- #272: Error adding meshes
- #269: Rename *GridXD*
- #255: numpy 1.5.1 and masked arrays
- #253: Move the mail archive link to a more prominent place on web page.
- #245: Fix *fipy.terms._BinaryTerm* test failure in parallel
- #228: *-pysparse* configuration should never attempt MPI imports
- #225: Windows interactive plotting mostly broken
- #209: add Rhie-Chow correction term in stokes cavity example
- #180: broken arithmetic face to cell distance calculations
- #128: Trying to “solve” an integer *CellVariable* should raise an error
- #123: *numerix.dot* doesn’t support tensors
- #103: *subscriber()._markStale()* *AttributeError*
- #61: Move *ImplicitDiffusionTerm().solve(var) == 0* “failure” from *examples.phase.simple* to *examples.diffusion.mesh1D*?

17.9 Version 3.0.1 - 2012-10-03

17.9.1 Fixes

- #346: text in *trunk/examples/convection/source.py* is out of date
- #342: sign issues for equation with transient, convection and implicit terms
- #338: SvnToGit clean up

17.10 Version 3.0 - 2012-08-16

The bump in major version number reflects more on the substantial increase in capabilities and ease of use than it does on a break in compatibility with FiPy 2.x. Few, if any, changes to your existing scripts should be necessary.

The significant changes since version 2.1 are:

- *Coupled and Vector Equations* are now supported.
- A more robust mechanism for specifying *Boundary Conditions* is now used.
- Most *Meshes* can be partitioned by *Meshing with Gmsh*.
- *PyAMG* and *SciPy* have been added to the *Solvers*.
- FiPy is capable of running under *Python 3*.
- “getter” and “setter” methods have been pervasively changed to Python properties.
- The test suite now runs much faster.
- Tests can now be run on a full install using *fipy.test()*.
- The functions of the *numerix* module are no longer included in the *fipy* namespace. See *examples.updating.update2_0to3_0* for details.

- Equations containing a *TransientTerm*, must specify the timestep by passing a `dt=` argument when calling `solve()` or `sweep()`.

Warning: *FiPy* 3 brought unavoidable syntax changes from *FiPy* 2. Please see [examples.updating.update2_0to3_0](#) for guidance on the changes that you will need to make to your *FiPy* 2.x scripts.

17.10.1 Fixes

- #332: Inline failure on Ubuntu x86_64
- #324: constraining values with *ImplicitSourceTerm* not documented?
- #317: *gmshImport* tests fail on Windows due to shared file
- #316: changes to *gmshImport.py* caused *-inline* problems
- #313: Gmsh I/O
- #307: Failures on sandbox under buildbot
- #306: Add in parallel buildbot testing on more than 2 processors
- #302: *CellVariable.min()* broken in parallel
- #301: *Epetra.PyComm()* broken on Debian
- #300: *examples/cahnHilliard/mesh2D.py* broken with *-trilinos*
- #299: Viewers not working when plotting meshes with zero cells in parallel
- #298: Memory consumption growth with repeated meshing, especially with Gmsh
- #294: *-pysparse -inline* failures
- #293: *python examples/cahnHilliard/sphere.py -inline* segfaults on OS X
- #292: two *-scipy* failures
- #290: Improve test reporting to avoid inconsequential buildbot failures
- #288: gmsh importer and gmsh tests don't clean up after themselves
- #286: get running in Py3k
- #285: remove deprecated *viewers.make()*
- #284: remove deprecated *Variable.transpose()*
- #281: remove deprecated *NthOrderDiffusionTerm*
- #280: remove deprecated *diffusionTerm=* argument to *ConvectionTerm*
- #277: remove deprecated *steps=* from Solver
- #273: Make *DiffusionTermNoCorrection* the default
- #270: tests take *too* long!!!
- #267: Reduce the run times for chemotaxis tests
- #264: HANG in parallel test of *examples/chemotaxis/input2D.py* on some configurations
- #261: *GmshImport* should read element colors
- #260: *GmshImport* should support all element types

- #259: Introduce *mesh.x* as shorthand for *mesh.cellCenters[0]* etc
- #258: *GmshExport* is not tested and does not work
- #252: Include Benny's improved interpolation patch
- #250: TeX is wrong in *examples.phase.quaternary*
- #247: *diffusionTerm(var=var1).solver(var=var0)* should fail sensibly
- #243: close out reconstrain branch
- #242: update documentation
- #240: Profile and merge reconstrain branch
- #237: *-Trilinos -no-pysparse* uses Pysparse?!?
- #236: anisotropic diffusion and constraints don't mix
- #235: changed constraints don't propagate
- #231: *factoryMeshes.py* not up to date with respect to keyword arguments
- #223: mesh in FiPy name space
- #218: Absence of *enthought.tvtk* causes test failures
- #216: Fresh FiPy gives "*ImportError: No viewers found*"
- #213: PyPI is failing
- #206: *gnuplot1d* gives error on plot of *FaceVariable*
- #205: wrong cell to cell normal in periodic meshes
- #203: Give helpful error on - or / of meshes
- #202: mesh manipulation of periodic meshes leads to errors
- #201: Use physical velocity in the manual/FAQ
- #200: FAQ gives bad guidance for anisotropic diffusion
- #195: term multiplication changes result
- #163: Default time steps should be infinite
- #162: remove ones and zeros from *numerix.py*
- #130: tests should be run with *fipy.tests()*
- #86: Grids should take *Lx*, *Ly*, *Lz* arguments
- #77: *CellVariable.hasOld()* should set *self.old*
- #44: Navier-Stokes

17.11 Version 2.1.3 - 2012-01-17

17.11.1 Fixes

- #282: remove deprecated getters and setters
- #279: remove deprecated *fipy.meshes.numMesh* submodule
- #278: remove deprecated forms of Gmsh meshes
- #268: Set up *Zizou* as a working slave
- #262: issue with solvers
- #256: *Grid1D(dx=(1,2,3))* failure
- #251: parallel is broken
- #241: Set *Sandbox* up as a working slave
- #238: *_BinaryTerm.var* is not predictable
- #233: coupled convection-diffusion always treated as Upwind
- #224: “matrices are not aligned” errors in example test suite
- #222: Non-uniform *Grid3D* fails to `__add__`
- #221: Problem with fipy and gmsh
- #219: matforge css is hammer-headed
- #208: numpy 2.0: *arrays have a dot method*
- #207: numpy 2.0: *masked arrays cast right of product to ndarray*
- #196: Pysparse won’t import in Python 2.6.5 on Windows
- #152: (Re)Implement SciPy solvers
- #138: FAQ on boundary conditions
- #100: testing from the Windows dist using the `ipython` command line
- #80: Windows - testing - `idle -ipython`
- #46: Variable needs to consider boundary conditions
- #45: Slicing a vector Variable should produce a scalar Variable

17.12 Version 2.1.2 - 2011-04-20

The significant changes since version 2.1.1 are:

- *Trilinos* efficiency improvements
- Diagnostics of the parallel environment

17.12.1 Fixes

- #232: Mayavi broken on windows because it has no *SIGHUP*.
- #230: *factoryMeshes.py* not up to date with respect to keyword arguments
- #226: *MatplotlibViewer* fails if backend doesn't support *flush_events()*
- #225: Windows interactive plotting mostly broken
- #217: Gmsh *CellVariables* can't be unpickled
- #191: *sphereDaemon.py* missing in FiPy 2.1 and from trunk
- #187: Concatenated *Mesh* garbled by *dump.write/read*

17.13 Version 2.1.1 - 2010-10-05

The significant changes since version 2.1 are:

- *MatplotlibViewer* can display into an existing set of Matplotlib axes.
- *Pysparse* and *Trilinos* are now completely independent.

17.13.1 Fixes

- #199: dummy viewer results in “*NotImplementedError: can't instantiate abstract base class*”
- #198: bug problem with *CylindricalGrid1D*
- #197: How to tell if parallel is configured properly?
- #194: *FIPY_DISPLAY_MATRIX* on empty matrix with large b-vector throws *ValueError*
- #193: *FIPY_DISPLAY_MATRIX* raises *ImportError* in FiPy 2.1 and trunk
- #192: *FIPY_DISPLAY_MATRIX=terms* raises *TypeError* in FiPy 2.1 and trunk

17.14 Version 2.1 - 2010-04-01

The relatively small change in version number belies significant advances in *FiPy* capabilities. This release did not receive a “full” version increment because it is completely (er...¹) compatible with older scripts.

The significant changes since version 2.0.2 are:

- *FiPy* can use *Trilinos* for *Solving in Parallel*.
- We have switched from *MayaVi* 1 to *Mayavi* 2. This *Viewer* is an independent process that allows interaction with the display while a simulation is running.
- Documentation has been switched to *Sphinx*, allowing the entire manual to be available on the web and for our documentation to link to the documentation for packages such as *numpy*, *scipy*, *matplotlib*, and for *Python* itself.

¹ Only two examples from *FiPy* 2.0 fail when run with *FiPy* 2.1:

- *examples/phase/symmetry.py* fails because *Mesh* no longer provides a *getCell* method. The mechanism for enforcing symmetry in the updated example is both clearer and faster.
- *examples.levelSet.distanceFunction.circle* fails because of a change in the comparison of masked values.

Both of these are subtle issues unlikely to affect very many *FiPy* users.

17.14.1 Fixes

- #190: “matplotlib: list index out of range” when no title given, but only sometimes
- #182: `~binOp` doesn’t work on branches/version-2_0
- #180: broken arithmetic face to cell distance calculations
- #179: `easy_install` instructions for Mac OS X are broken
- #177: broken `setuptools` url with python 2.6
- #169: The FiPy webpage seems to be broken on Internet Explorer
- #156: update the mayavi viewer to use mayavi 2
- #153: Switch documentation to use `:math:` directive

17.15 Version 2.0.3 - 2010-03-17

17.15.1 Fixes

- #188: *SMTPSenderRefused: (553, “5.1.8 <trac@matdl-osi.org>... Domain of sender address trac@matdl-osi.org does not exist”, u““FiPy” <trac@matdl-osi.org>”)*
- #184: `gmshExport.exportAsMesh()` doesn’t work
- #183: FiPy 2.0.2 `LinearJORSolver.__init__` calls *Solver* rather than *PysparseSolver*
- #181: Navier-Stokes again
- #151: update mayavi viewer to use mayavi2
- #13: Mesh refactor

17.16 Version 2.0.2 - 2009-06-11

17.16.1 Fixes

- #176: Win32 distribution test error
- #175: `Grid3D.getFaceCenters` incorrect when mesh is offset
- #170: `Variable` doesn’t implement `__invert__`

17.17 Version 2.0.1 - 2009-04-23

17.17.1 Fixes

- #154: Update manuals

17.18 Version 2.0 - 2009-02-09

Warning: *FiPy* 2 brings unavoidable syntax changes. Please see [examples.updating.update1_0to2_0](#) for guidance on the changes that you will need to make to your *FiPy* 1.x scripts.

The significant changes since version 1.2 are:

- *CellVariable* and *FaceVariable* objects can hold values of any rank.
- Much simpler syntax for specifying *Cells* for initial conditions and *Faces* for boundary conditions.
- Automated determination of the Péclet number and partitioning of *ImplicitSourceTerm* coefficients between the matrix diagonal and the right-hand-side-vector.
- Simplified *Viewer* syntax.
- Support for the *Trilinos* solvers.
- Support for anisotropic diffusion coefficients.
- #167: example showing how to go from 1.2 to 2.0
- #166: Still references to *VectorCell* and *VectorFace Variable* in manual
- #165: Edit the what's new section of the manual
- #149: Test viewers
- #143: Document syntax changes
- #141: enthought toolset?
- #140: easy_install fipy
- #136: Document anisotropic diffusion
- #135: Trilinos documentation
- #127: Examples can be very fragile with respect to floating point

17.19 Version 1.2.3 - 2009-01-0

17.19.1 Fixes

- #54: *python setup.py test* fails

17.20 Version 1.2.2 - 2008-12-30

17.20.1 Fixes

- #161: get pyparse working with python 2.4
- #160: Grid class
- #157: temp files on widows
- #155: fix some of the deprecation warnings appearing in the tests

- #150: PythonXY installation?
- #148: SciPy 0.7.0 solver failures on Macs
- #147: Disable CGS solver in pyparse
- #145: *Viewer* factory fails for rank-1 *CellVariable*
- #144: intermittent failure on *examples/diffusion/explicit/mixedelement.py* *-inline*
- #142: merge Viewers branch
- #139: Get a Windows Bitten build slave
- #137: Backport examples from manuscript
- #131: *MatplotlibViewer* doesn't properly report the supported file extensions
- #126: Variable, float, integer
- #125: Pickled test data embeds obsolete packages
- #124: Can't pickle a *binOp*
- #121: *simpleTrenchSystem.py*
- #120: mayavi display problems
- #118: Automatically handle casting of *Variable* from *int* to *float* when necessary.
- #117: *getFacesBottom*, *getFacesTop* etc. lack clear description in the reference
- #115: viewing 3D Cahn-Hilliard is broken
- #113: OS X (MacBook Pro; Intel) FiPy installation problems
- #112: *stokesCavity.py* doesn't display properly with matplotlib
- #111: Can't display *Grid2D* variables with matplotlib on Linux
- #110: "Numeric array value must be dimensionless" in ElPhF examples
- #109: doctest of *fipy.variables.variable.Variable.__array__*
- #108: *numerix.array * FaceVariable* is broken
- #107: Can't move matplotlib windows on Mac
- #106: Concatenation of *Grid1D* objects doesn't always work
- #105: useless broken *__array__* tests should be removed
- #102: viewer limits should just be set as arguments, rather than as a dict
- #99: *Matplotlib2DGridViewer* cannot update multiple views
- #97: Windows does not seem to handle NaN correctly.
- #96: broken tests with version 2.0 of gmsh
- #95: attached code breaks with *-inline*
- #92: Pygist is dead (it's official)
- #84: Test failures on Intel Mac
- #83: *ZeroDivisionError* for *CellTerm* when calling *getOld()* on its coefficient
- #79: *viewers.make()* to *viewers.Viewer()*
- #67: Mesh viewing and unstructured data.

- #43: *TSVViewer* doesn't always get the right shape for the var
- #34: `float(&infinity&)` issue on windows

17.21 Version 1.2.1 - 2008-02-08

17.21.1 Fixes

- #122: check argument types for meshes
- #119: `max` is broken for Variables
- #116: Linux: failed test, *TypeError: No array interface...* in `solve()`
- #104: Syntax error in *MatplotlibVectorViewer._plot()*
- #101: matplotlib 1D viewer autoscales when a limit is set to 0
- #93: Broken examples
- #91: update the examples to use *from fipy import **
- #76: `solve()` and `sweep()` accept `dt=CellVariable`
- #75: installation of fipy should auto include README as a docstring
- #74: Some combinations of *DiffusionTerm* and *ConvectionTerm* do not work
- #51: `__pos__` doesn't work for terms
- #50: Broken examples
- #39: matplotlib broken on mac with version 0.72.1
- #19: Péclet number
- #15: Boundary conditions and Terms

17.22 Version 1.2 - 2007-02-12

The significant changes since version 1.1 are:

- *-inline* automatically generates C code from *Variable* expressions.
- *FiPy* has been updated to use the *Python NumPy* module. *FiPy* no longer works with the older *Numeric* module.

17.22.1 Fixes

- #98: Windows patch for some broken test cases
- #94: *-inline* error for attached code
- #90: bug in matplotlib 0.87.7: *TypeError: only length-1 arrays can be converted to Python scalars.*
- #72: needless rebuilding of variables
- #66: PDF rendering issues for the guide on various platforms
- #62: fipy guide pdf bug: “*an unrecognized token 13c was found*”
- #55: Error for internal BCs

- #52: *FaceVariable * FaceVectorVariable* memory
- #48: Documentation is not inherited from `&hidden&` classes
- #42: *fipy.models.phase.phase.addOverFacesVariable* is gross
- #41: `EFFICIENCY.txt` example fails to make viewer
- #30: periodic boundary condition support
- #25: make phase field examples more explicit
- #23: sweep control, iterator object, error norms
- #21: Update FiPy to use numpy
- #16: Dimensions
- #12: Refactor viewers
- #1: Gnuplot doesn't display on windows

17.23 Version 1.1 - 2006-06-06

The significant changes since version 1.0 are:

- Memory efficiency has been improved in a number of ways, but most significantly by:
 - not caching all intermediate `Variable` values.
 - introducing `UniformGrid` classes that calculate geometric arrays on the fly.

Details of these improvements are presented in *Efficiency*.

- Installation on Windows has been made considerably easier by constructing executable installers for *FiPy* and its dependencies.
- The arithmetic for `Variable` subclasses now works, and returns sensible answers. For example, `VectorCellVariable * CellVariable` returns a `VectorCellVariable`.
- `PeriodicGrid` meshes have been implemented. Currently, however, there are no examples of their use in the manual.
- Many of the examples have been completely rewritten
 - A basic 1D diffusion problem now serves as a general tutorial for setting up any problem in *FiPy*.
 - Several more phase field examples have been added that should make it clearer how to get from the simple 1D case to the more elaborate multicomponent, multidimensional, and anisotropic models.
 - The “Superfill” examples have been substantially improved with better functionality and documentation.
 - An example of fluid flow with the classic Stokes moving lid has been added.
- A clear distinction has been made between solving an equation via `solve()` and iterating an non-linear equation to solution via `sweep()`. An extensive explanation of the concepts involved has been added to the *Frequently Asked Questions*.
- Added a *MultiViewer* class that automatically groups several viewers together if the variables couldn't be displayed by a single viewer.
- The abbreviated syntax `from fipy import Class` or `from fipy import *` promised in version 1.0 actually works now. The examples all still use the fully qualified names.

- The repository has been converted from a CVS to a [Subversion](#) repository. Details on how to check out the new repository are given in [Installation](#).
- The *FiPy* repository has also been moved from [Sourceforge](#) to the [Materials Digital Library Pathway](#).

17.24 Version 1.0 - 2005-09-16

Numerous changes have been made since *FiPy* 0.1 was released, but the most significant ones are:

- `Equation` objects no longer exist. PDEs are constructed from `Term` objects. `Term` objects can be added, subtracted, and equated to build up an equation.
- A true 1D grid class has been added: `fipy.meshes.grid1D.Grid1D`.
- A generic “factory” method `fipy.viewers.make()` has been added that will do a reasonable job of automatically creating a `Viewer` for the supplied `Variable` objects. The `FIPY_VIEWER` environment variable allows you to specify your preferred viewer.
- A simple `TSVViewer` has been added to allow display or export to a file of your solution data.
- It is no longer necessary to `transpose()` scalar fields in order to multiply them with vector fields.
- Better default choice of solver when convection is present.
- Better examples.
- A number of *NoiseVariable* objects have been added.
- A new viewer based on *Matplotlib* has been added.
- The *PyX* viewer has been removed.
- Considerably simplified the public interface to *FiPy*.
- Support for Python 2.4.
- Improved layout of the manuals.
- `getLaplacian()` method has been removed from `CellVariable` objects. You can obtain the same effect with `getFaceGrad().getDivergence()`, which provides better control.
- An import shorthand has been added that allows for:

```
from fipy import Class
```

instead of:

```
from fipy.some.deeply.nested.module.class import Class
```

This system is still experimental. Please tell us if you find situations that don’t work.

The syntax of *FiPy* 1.0 scripts is incompatible with earlier releases. A tutorial for updating your existing scripts can be found in `examples/updating/update0_1to1_0.py`.

17.24.1 Fixes

- #49: Documentation for many *ConvectionTerms* is wrong
- #47: Terms should throw an error on bad *coeff* type
- #40: broken levelset test case
- #38: multiple BCs on one face broken?
- #37: Better support for periodic boundary conditions
- #36: Gnuplot doesn't display the `~examples/levelSet/electroChem` problem on windows.
- #35: gmsh write problem on windows
- #33: *DiffusionTerm(coeff = CellVariable)* functionality
- #32: `conflict_handler = ignore` not valid in Python 2.4
- #31: Support simple import notation
- #29: periodic boundary conditions are broken
- #28: invoke the `==` for terms
- #26: doctest extraction with python 2.4
- #24: Pysparse windows binaries
- #22: automated `efficiency_test` problems
- #20: Test with Python version 2.4
- #18: Memory leak for the leveling problem
- #17: *distanceVariable* is broken
- #14: Testing mailing list interface
- #11: Reconcile versions of pysparse
- #10: check phase field crystal growth
- #9: implement levelling surfactant equation
- #8: merge *depositionRateVar* and *extensionVelocity*
- #7: Automate FiPy efficiency test
- #6: FiPy breaks on windows with Numeric 23.6
- #5: axisymmetric 2D mesh
- #4: Windows installation wizard
- #3: Windows installation instructions
- #2: Some tests fail on windows XP

17.25 Version 0.1.1

17.26 Version 0.1 - 2004-11-05

Original release

Glossary

AppVeyor A cloud-based *Continuous Integration* tool. See <https://www.appveyor.com>.

Buildbot The Buildbot is a system to automate the compile/test cycle required by most software projects to validate code changes. No longer used for *FiPy*. See <http://trac.buildbot.net/>.

CircleCI A cloud-based *Continuous Integration* tool. See <https://circleci.com>.

conda An open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies. Conda easily creates, saves, loads and switches between environments on your local computer. It was created for Python programs, but it can package and distribute software for any language. See <https://conda.io>.

Continuous Integration The practice of frequently testing and integrating one's new or changed code with the existing code repository. See https://en.wikipedia.org/wiki/Continuous_integration.

FiPy The eponymous software package. See <http://www.ctcms.nist.gov/fipy>.

Gmsh A free and Open Source 3D (and 2D!) finite element grid generator. It also has a CAD engine and post-processor that *FiPy* does not make use of. See <http://www.geuz.org/gmsh>.

IPython An improved *Python* shell that integrates nicely with *Matplotlib*. See <http://ipython.scipy.org/>.

linux An operating system. See <http://www.linux.org>.

macOS An operating system. See <http://www.apple.com/macos>.

Matplotlib `matplotlib` *Python* package displays publication quality results. It displays both 1D X-Y type plots and 2D contour plots for structured data. It does not display unstructured 2D data or 3D data. It works on all common platforms and produces publication quality hard copies. See <http://matplotlib.sourceforge.net> and *Matplotlib*.

Mayavi The `mayavi` Data Visualizer is a free, easy to use scientific data visualizer. It displays 1D, 2D and 3D data. It is the only *FiPy* viewer available for 3D data. Other viewers are probably better for 1D or 2D viewing. See <http://code.enthought.com/projects/mayavi> and *Mayavi*.

MayaVi The predecessor to *Mayavi*. Yes, it's confusing.

MPI The Message Passing Interface is a standard that allows the use of multiple processors. See <http://www.mpi-forum.org>

mpi4py MPI for Python provides bindings of the Message Passing Interface (*MPI*) standard for the Python programming language, allowing any Python program to exploit multiple processors. For *Solving in Parallel*, *FiPy* requires `mpi4py`, in addition to *PETSc* or *Trilinos*. See <https://mpi4py.readthedocs.io>.

numarray An archaic predecessor to *NumPy*.

Numeric An archaic predecessor to *NumPy*.

NumPy The `numpy` *Python* package provides array arithmetic facilities. See <http://www.scipy.org/NumPy>.

OpenMP The Open Multi-Processing architecture is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. See <https://www.openmp.org>.

pandas “Python Data Analysis Library” provides high-performance data structures for flexible, extensible analysis. See <http://pandas.pydata.org>.

PETSc The Portable, Extensible Toolkit for Scientific Computation is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations. See <https://www.mcs.anl.gov/petsc> and *PETSc*.

petsc4py *Python* wrapper for *PETSc*. See <https://petsc4py.readthedocs.io/>.

pip “pip installs python” is a tool for installing and managing Python packages, such as those found in *PyPI*. See <http://www.pip-installer.org>.

PyAMG A suite of python-based preconditioners. See <http://code.google.com/p/pyamg/> and *PyAMG*.

pyamgx a *Python* interface to the NVIDIA *AMGX* library, which can be used to construct complex solvers and preconditioners to solve sparse sparse linear systems on the GPU. See <https://pyamgx.readthedocs.io/> and *pyamgx*.

PyPI The Python Package Index is a repository of software for the *Python* programming language. See <http://pypi.python.org/pypi>.

Pyrex A mechanism for mixing C and Python code. See <http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/>.

Pysparse The `pysparse` *Python* package provides sparse matrix storage, solvers, and linear algebra routines. See <http://pysparse.sourceforge.net> and *Pysparse*.

Python The programming language that *FiPy* (and your scripts) are written in. See <http://www.python.org/>.

Python 3 The (likely) future of the *Python* programming language. Third-party packages are slowly being adapted, but many that *FiPy* uses are not yet available. See <http://docs.python.org/py3k/> and **PEP 3000**.

PyTrilinos *Python* wrapper for *Trilinos*. See <http://trilinos.sandia.gov/packages/pytrilinos/>.

PyxViewer A now defunct python viewer.

ScientificPython A collection of useful utilities for scientists. See <http://dirac.cnrs-orleans.fr/plone/software/scientificpython>.

SciPy The `scipy` package provides a wide range of scientific and mathematical operations. *FiPy* can use *SciPy*’s solver suite for linear solutions. See <http://www.scipy.org/>. and *SciPy*.

Sphinx The tools used to generate the *FiPy* documentation. See <http://sphinx.pocoo.org/>.

TravisCI A cloud-based *Continuous Integration* tool. See <https://travis-ci.org>.

Trilinos This package provides sparse matrix storage, solvers, and preconditioners, and can be used instead of *Pysparse*. *Trilinos* preconditioning allows for iterative solutions to some difficult problems that *Pysparse* cannot solve. See <http://trilinos.sandia.gov> and *Trilinos*.

Weave The `weave` package can enhance performance with C language inlining. See <https://github.com/scipy/weave>.

Windows An operating system. See <http://www.microsoft.com/windows>.

Part II

Examples

Note: Any given module “example.something.input” can be found in the file “examples/something/input.py”.

These examples can be used in at least four ways:

- Each example can be invoked individually to demonstrate an application of *FiPy*:

```
$ python examples/something/input.py
```

- Each example can be invoked such that when it has finished running, you will be left in an interactive *Python* shell:

```
$ python -i examples/something/input.py
```

At this point, you can enter *Python* commands to manipulate the model or to make queries about the example’s variable values. For instance, the interactive *Python* sessions in the example documentation can be typed in directly to see that the expected results are obtained.

- Alternatively, these interactive *Python* sessions, known as *doctest* blocks, can be invoked as automatic tests:

```
$ python setup.py test --examples
```

In this way, the documentation and the code are always certain to be consistent.

- Finally, and most importantly, the examples can be used as a templates to design your own problem scripts.

Note: The examples shown in this manual have been written with particular emphasis on serving as both documentation and as comprehensive tests of the FiPy framework. As explained at the end of `examples/diffusion/steadyState/mesh1D.py`, your own scripts can be much more succinct, if you wish, and include only the text that follows the “>>>” and “...” prompts shown in these examples.

To obtain a copy of an example, containing just the script instructions, type:

```
$ python setup.py copy_script --From x.py --To y.py
```

In addition to those presented in this manual, there are dozens of other files in the `examples/` directory, that demonstrate other uses of FiPy. If these examples do not help you construct your own problem scripts, please [contact us](#).

Chapter 19

Diffusion Examples

<code>examples.diffusion.mesh1D</code>	Solve a one-dimensional diffusion equation under different conditions.
<code>examples.diffusion.coupled</code>	Solve the biharmonic equation as a coupled pair of diffusion equations.
<code>examples.diffusion.mesh20x20</code>	Solve a two-dimensional diffusion problem in a square domain.
<code>examples.diffusion.circle</code>	Solve the diffusion equation in a circular domain meshed with triangles.
<code>examples.diffusion.electrostatics</code>	Solve the Poisson equation in one dimension.
<code>examples.diffusion.nthOrder. input4thOrder1D</code>	Solve a fourth-order diffusion problem.
<code>examples.diffusion.anisotropy</code>	Solve the diffusion equation with an anisotropic diffusion coefficient.

19.1 `examples.diffusion.mesh1D`

Solve a one-dimensional diffusion equation under different conditions.

To run this example from the base *FiPy* directory, type:

```
$ python examples/diffusion/mesh1D.py
```

at the command line. Different stages of the example should be displayed, along with prompting messages in the terminal.

This example takes the user through assembling a simple problem with *FiPy*. It describes different approaches to a 1D diffusion problem with constant diffusivity and fixed value boundary conditions such that,

$$\frac{\partial \phi}{\partial t} = D \nabla^2 \phi. \quad (19.1)$$

The first step is to define a one dimensional domain with 50 solution points. The `Grid1D` object represents a linear structured grid. The parameter `dx` refers to the grid spacing (set to unity here).

```
>>> from fipy import Variable, FaceVariable, CellVariable, Grid1D, \
↳ ExplicitDiffusionTerm, TransientTerm, DiffusionTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> nx = 50
>>> dx = 1.
>>> mesh = Grid1D(nx=nx, dx=dx)
```

FiPy solves all equations at the centers of the cells of the mesh. We thus need a *CellVariable* object to hold the values of the solution, with the initial condition $\phi = 0$ at $t = 0$,

```
>>> phi = CellVariable(name="solution variable",
...                     mesh=mesh,
...                     value=0.)
```

We'll let

```
>>> D = 1.
```

for now.

The set of boundary conditions are given to the equation as a Python tuple or list (the distinction is not generally important to *FiPy*). The boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 1, \\ 1 & \text{at } x = 0. \end{cases}$$

are formed with a value

```
>>> valueLeft = 1
>>> valueRight = 0
```

and a set of faces over which they apply.

Note: Only faces around the exterior of the mesh can be used for boundary conditions.

For example, here the exterior faces on the left of the domain are extracted by `mesh.facesLeft`. The boundary conditions is applied using `phi.constrain()` with these faces and a value (`valueLeft`).

```
>>> phi.constrain(valueRight, mesh.facesRight)
>>> phi.constrain(valueLeft, mesh.facesLeft)
```

Note: If no boundary conditions are specified on exterior faces, the default boundary condition is equivalent to a zero gradient, equivalent to $\vec{n} \cdot \nabla \phi|_{\text{someFaces}} = 0$.

If you have ever tried to numerically solve Eq. (19.1), you most likely attempted “explicit finite differencing” with code something like:

```
for step in range(steps):
    for j in range(cells):
        phi_new[j] = phi_old[j] \
            + (D * dt / dx**2) * (phi_old[j+1] - 2 * phi_old[j] + phi_old[j-1])
    time += dt
```

plus additional code for the boundary conditions. In *FiPy*, you would write

```
>>> eqX = TransientTerm() == ExplicitDiffusionTerm(coeff=D)
```

The largest stable timestep that can be taken for this explicit 1D diffusion problem is $\Delta t \leq \Delta x^2 / (2D)$.

We limit our steps to 90% of that value for good measure

```
>>> timeStepDuration = 0.9 * dx**2 / (2 * D)
>>> steps = 100
```

If we're running interactively, we'll want to view the result, but not if this example is being run automatically as a test. We accomplish this by having Python check if this script is the “__main__” script, which will only be true if we explicitly launched it and not if it has been imported by another script such as the automatic tester. The factory function *Viewer()* returns a suitable viewer depending on available viewers and the dimension of the mesh.

```
>>> phiAnalytical = CellVariable(name="analytical value",
...                               mesh=mesh)
```

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(phi, phiAnalytical),
...                         datamin=0., datamax=1.)
...     viewer.plot()
```

In a semi-infinite domain, the analytical solution for this transient diffusion problem is given by $\phi = 1 - \text{erf}(x/2\sqrt{Dt})$. If the *SciPy* library is available, the result is tested against the expected profile:

```
>>> x = mesh.cellCenters[0]
>>> t = timeStepDuration * steps
```

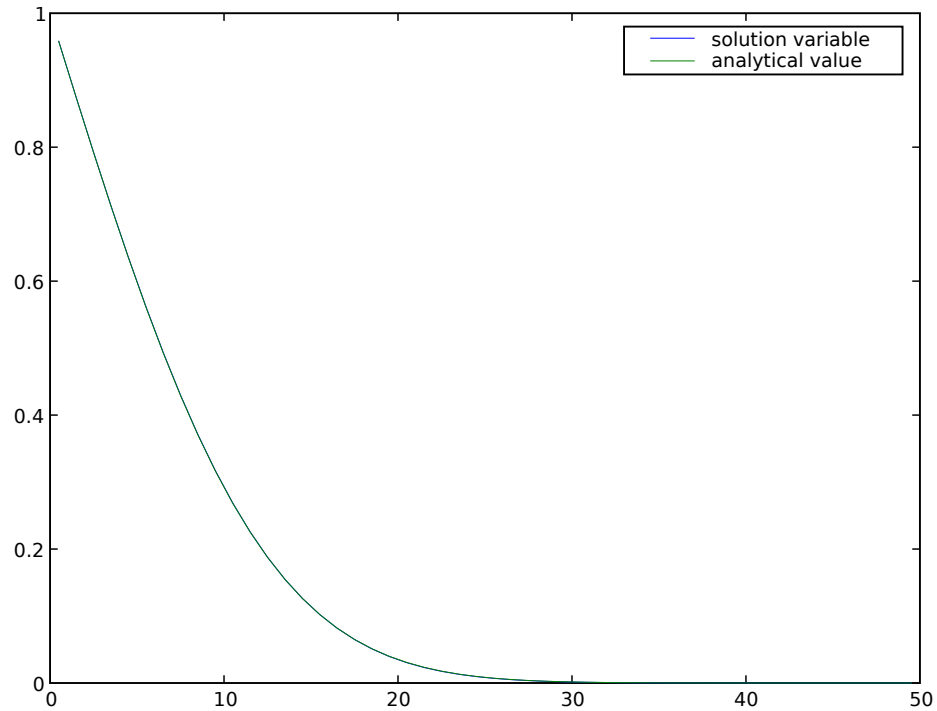
```
>>> try:
...     from scipy.special import erf
...     phiAnalytical.setValue(1 - erf(x / (2 * numerix.sqrt(D * t))))
... except ImportError:
...     print("The SciPy library is not available to test the solution to \
... the transient diffusion equation")
```

We then solve the equation by repeatedly looping in time:

```
>>> from builtins import range
>>> for step in range(steps):
...     eqX.solve(var=phi,
...               dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
```

```
>>> print(phi.allclose(phiAnalytical, atol = 7e-4))
1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Explicit transient diffusion. Press <return> to proceed...")
```



Although explicit finite differences are easy to program, we have just seen that this 1D transient diffusion problem is limited to taking rather small time steps. If, instead, we represent Eq. (19.1) as:

```
phi_new[j] = phi_old[j] \
    + (D * dt / dx**2) * (phi_new[j+1] - 2 * phi_new[j] + phi_new[j-1])
```

it is possible to take much larger time steps. Because `phi_new` appears on both the left and right sides of the equation, this form is called “implicit”. In general, the “implicit” representation is much more difficult to program than the “explicit” form that we just used, but in *FiPy*, all that is needed is to write

```
>>> eqI = TransientTerm() == DiffusionTerm(coeff=D)
```

reset the problem

```
>>> phi.setValue(valueRight)
```

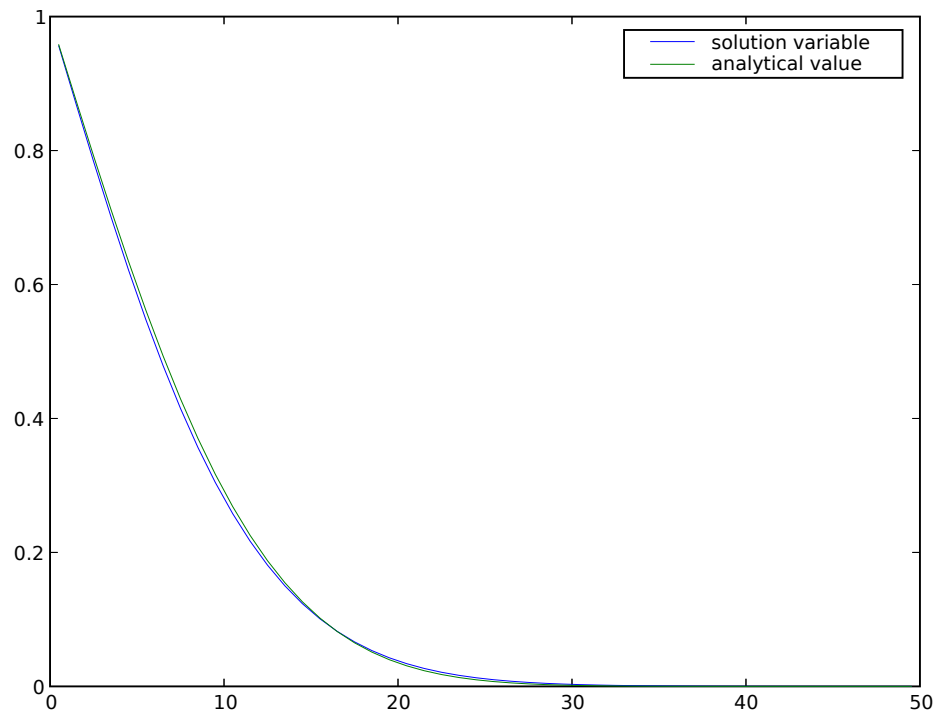
and rerun with much larger time steps

```
>>> timeStepDuration *= 10
>>> steps //= 10
>>> from builtins import range
>>> for step in range(steps):
...     eqI.solve(var=phi,
...               dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
```



```
>>> print(phi.allclose(phiAnalytical, atol = 2e-2))
1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Implicit transient diffusion. Press <return> to proceed...")
```



Note that although much larger *stable* timesteps can be taken with this implicit version (there is, in fact, no limit to how large an implicit timestep you can take for this particular problem), the solution is less *accurate*. One way to achieve a compromise between *stability* and *accuracy* is with the Crank-Nicholson scheme, represented by:

```
phi_new[j] = phi_old[j] + (D * dt / (2 * dx**2)) * \
    ((phi_new[j+1] - 2 * phi_new[j] + phi_new[j-1])
     + (phi_old[j+1] - 2 * phi_old[j] + phi_old[j-1]))
```

which is essentially an average of the explicit and implicit schemes from above. This can be rendered in *FiPy* as easily as

```
>>> eqCN = eqX + eqI
```

We again reset the problem

```
>>> phi.setValue(valueRight)
```

and apply the Crank-Nicholson scheme until the end, when we apply one step of the fully implicit scheme to drive down the error (see, *e.g.*, section 19.2 of [23]).

```
>>> from builtins import range
>>> for step in range(steps - 1):
...     eqCN.solve(var=phi,
...                 dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
>>> eqI.solve(var=phi,
...            dt=timeStepDuration)
>>> if __name__ == '__main__':
...     viewer.plot()
```

```
>>> print(phi.allclose(phiAnalytical, atol = 3e-3))
1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Crank-Nicholson transient diffusion. Press <return> to proceed...")
```

As mentioned above, there is no stable limit to how large a time step can be taken for the implicit diffusion problem. In fact, if the time evolution of the problem is not interesting, it is possible to eliminate the time step altogether by omitting the *TransientTerm*. The steady-state diffusion equation

$$D\nabla^2\phi = 0$$

is represented in *FiPy* by

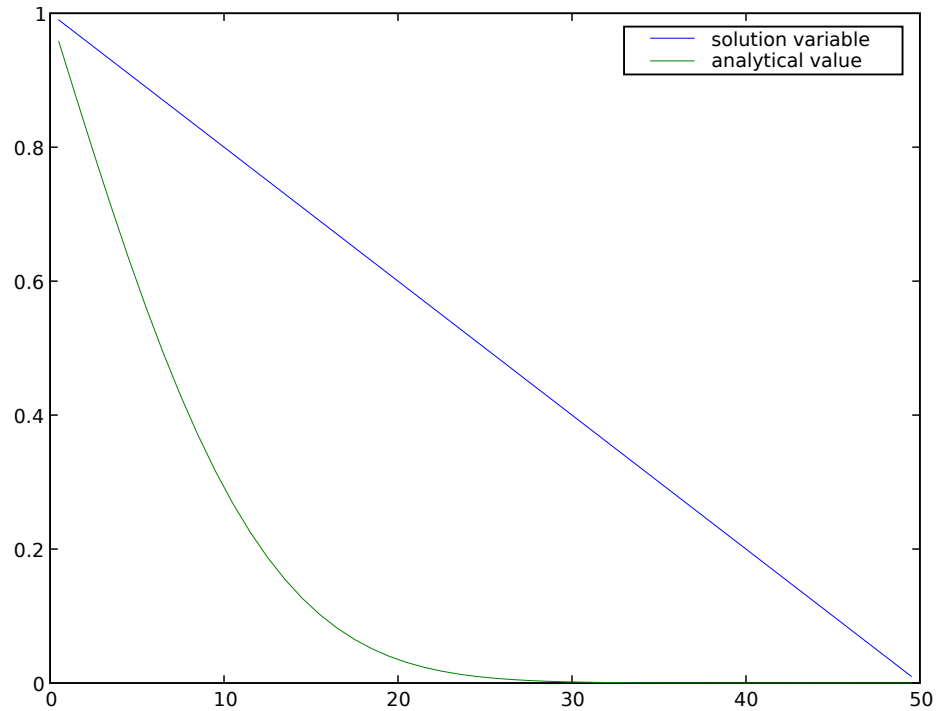
```
>>> DiffusionTerm(coeff=D).solve(var=phi)
```

```
>>> if __name__ == '__main__':
...     viewer.plot()
```

The analytical solution to the steady-state problem is no longer an error function, but simply a straight line, which we can confirm to a tolerance of 10^{-10} .

```
>>> L = nx * dx
>>> print(phi.allclose(valueLeft + (valueRight - valueLeft) * x / L,
...                     rtol = 1e-10, atol = 1e-10))
1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Implicit steady-state diffusion. Press <return> to proceed...")
```



Often, boundary conditions may be functions of another variable in the system or of time.

For example, to have

$$\phi = \begin{cases} (1 + \sin t)/2 & \text{on } x = 0 \\ 0 & \text{on } x = L \end{cases}$$

we will need to declare time t as a *Variable*

```
>>> time = Variable()
```

and then declare our boundary condition as a function of this *Variable*

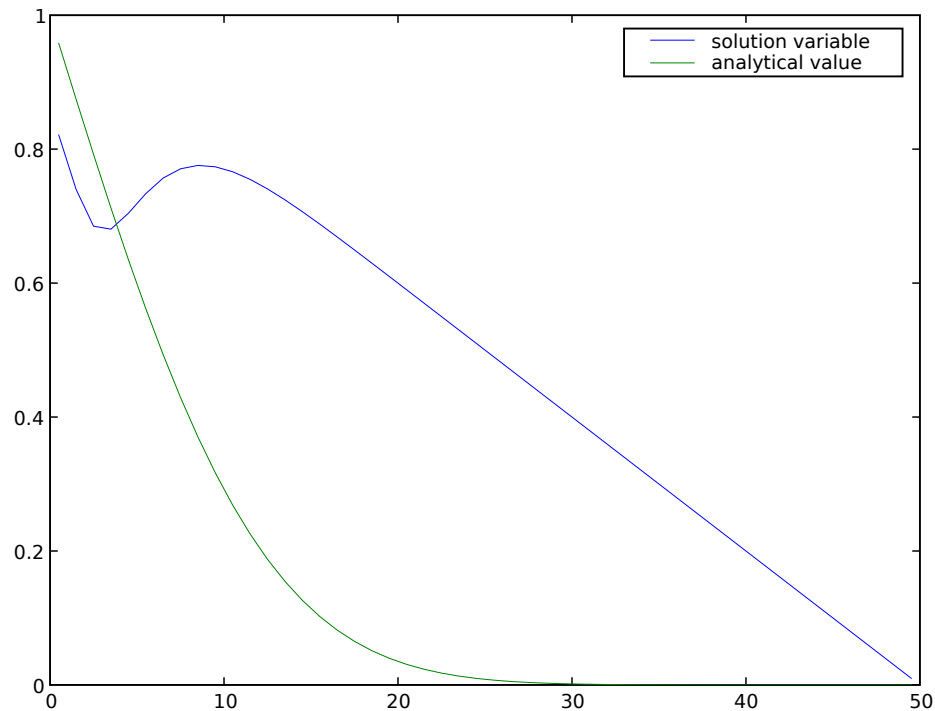
```
>>> del phi.faceConstraints
>>> valueLeft = 0.5 * (1 + numerix.sin(time))
>>> phi.constrain(valueLeft, mesh.facesLeft)
>>> phi.constrain(0., mesh.facesRight)
```

```
>>> eqI = TransientTerm() == DiffusionTerm(coeff=D)
```

When we update time at each timestep, the left-hand boundary condition will automatically update,

```
>>> dt = .1
>>> while time() < 15:
...     time.setValue(time() + dt)
...     eqI.solve(var=phi, dt=dt)
...     if __name__ == '__main__':
...         viewer.plot()
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Time-dependent boundary condition. Press <return> to proceed...")
```



Many interesting problems do not have simple, uniform diffusivities. We consider a steady-state diffusion problem

$$\nabla \cdot (D \nabla \phi) = 0,$$

with a spatially varying diffusion coefficient

$$D = \begin{cases} 1 & \text{for } 0 < x < L/4, \\ 0.1 & \text{for } L/4 \leq x < 3L/4, \\ 1 & \text{for } 3L/4 \leq x < L, \end{cases}$$

and with boundary conditions $\phi = 0$ at $x = 0$ and $D \frac{\partial \phi}{\partial x} = 1$ at $x = L$, where L is the length of the solution domain. Exact numerical answers to this problem are found when the mesh has cell centers that lie at $L/4$ and $3L/4$, or when the number of cells in the mesh N_i satisfies $N_i = 4i + 2$, where i is an integer. The mesh we've been using thus far is satisfactory, with $N_i = 50$ and $i = 12$.

Because *FiPy* considers diffusion to be a flux from one cell to the next, through the intervening face, we must define the non-uniform diffusion coefficient on the mesh faces

```
>>> D = FaceVariable(mesh=mesh, value=1.0)
>>> X = mesh.faceCenters[0]
>>> D.setValue(0.1, where=(L / 4. <= X) & (X < 3. * L / 4.))
```

The boundary conditions are a fixed value of

```
>>> valueLeft = 0.
```

to the left and a fixed flux of

```
>>> fluxRight = 1.
```

to the right:

```
>>> phi = CellVariable(mesh=mesh)
>>> phi.faceGrad.constrain([fluxRight], mesh.facesRight)
>>> phi.constrain(valueLeft, mesh.facesLeft)
```

We re-initialize the solution variable

```
>>> phi.setValue(0)
```

and obtain the steady-state solution with one implicit solution step

```
>>> DiffusionTerm(coeff = D).solve(var=phi)
```

The analytical solution is simply

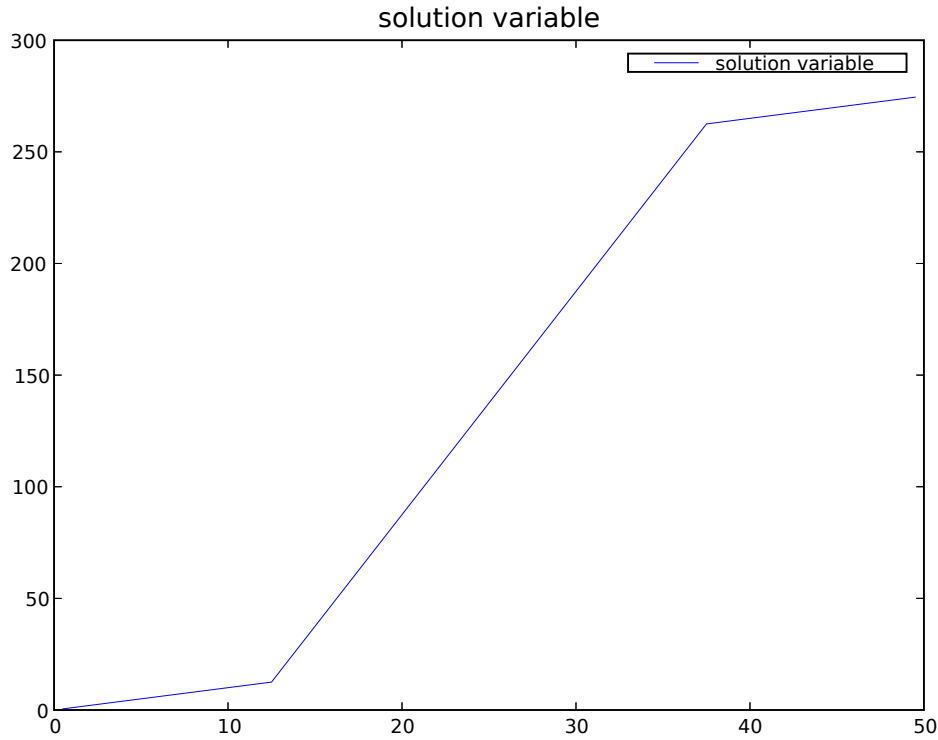
$$\phi = \begin{cases} x & \text{for } 0 < x < L/4, \\ 10x - 9L/4 & \text{for } L/4 \leq x < 3L/4, \\ x + 18L/4 & \text{for } 3L/4 \leq x < L, \end{cases}$$

or

```
>>> x = mesh.cellCenters[0]
>>> phiAnalytical.setValue(x)
>>> phiAnalytical.setValue(10 * x - 9. * L / 4.,
...                        where=(L / 4. <= x) & (x < 3. * L / 4.))
>>> phiAnalytical.setValue(x + 18. * L / 4.,
...                        where=3. * L / 4. <= x)
>>> print(phi.allclose(phiAnalytical, atol = 1e-8, rtol = 1e-8))
1
```

And finally, we can plot the result

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     Viewer(vars=(phi, phiAnalytical)).plot()
...     input("Non-uniform steady-state diffusion. Press <return> to proceed...")
```



Note that for problems involving heat transfer and other similar conservation equations, it is important to ensure that we begin with the correct form of the equation. For example, for heat transfer with ϕ representing the temperature,

$$\frac{\partial}{\partial t} (\rho \hat{C}_p \phi) = \nabla \cdot [k \nabla \phi].$$

With constant and uniform density ρ , heat capacity \hat{C}_p and thermal conductivity k , this is often written like Eq. (19.1), but replacing D with $\alpha = \frac{k}{\rho \hat{C}_p}$. However, when these parameters vary either in position or time, it is important to be careful with the form of the equation used. For example, if $k = 1$ and

$$\rho \hat{C}_p = \begin{cases} 1 & \text{for } 0 < x < L/4, \\ 10 & \text{for } L/4 \leq x < 3L/4, \\ 1 & \text{for } 3L/4 \leq x < L, \end{cases},$$

then we have

$$\alpha = \begin{cases} 1 & \text{for } 0 < x < L/4, \\ 0.1 & \text{for } L/4 \leq x < 3L/4, \\ 1 & \text{for } 3L/4 \leq x < L, \end{cases}.$$

However, using a `DiffusionTerm` with the same coefficient as that in the section above is incorrect, as the steady state governing equation reduces to $0 = \nabla^2 \phi$, which results in a linear profile in 1D, unlike that for the case above with spatially varying diffusivity. Similar care must be taken if there is time dependence in the parameters in transient problems.

We can illustrate the differences with an example. We define field variables for the correct and incorrect solution

```
>>> phiT = CellVariable(name="correct", mesh=mesh)
>>> phiF = CellVariable(name="incorrect", mesh=mesh)
>>> phiT.faceGrad.constrain([fluxRight], mesh.facesRight)
>>> phiF.faceGrad.constrain([fluxRight], mesh.facesRight)
>>> phiT.constrain(valueLeft, mesh.facesLeft)
>>> phiF.constrain(valueLeft, mesh.facesLeft)
>>> phiT.setValue(0)
>>> phiF.setValue(0)
```

The relevant parameters are

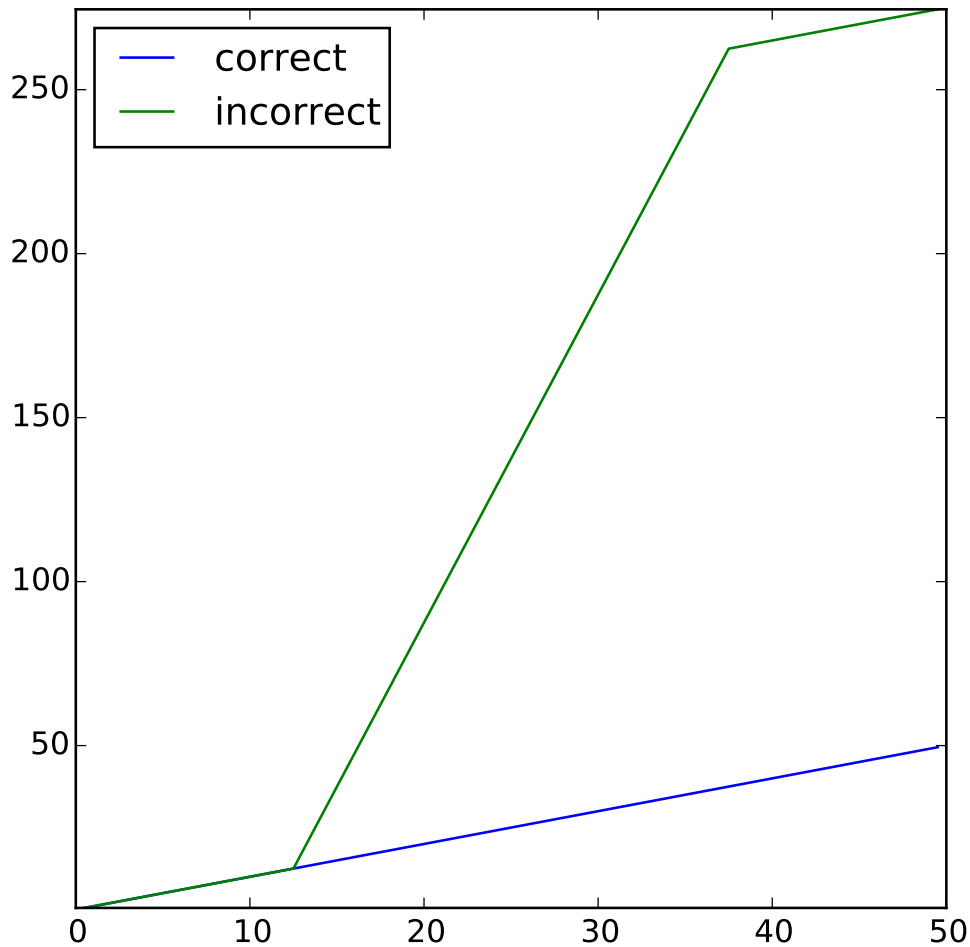
```
>>> k = 1.
>>> alpha_false = FaceVariable(mesh=mesh, value=1.0)
>>> X = mesh.faceCenters[0]
>>> alpha_false.setValue(0.1, where=(L / 4. <= X) & (X < 3. * L / 4.))
>>> eqF = 0 == DiffusionTerm(coeff=alpha_false)
>>> eqT = 0 == DiffusionTerm(coeff=k)
>>> eqF.solve(var=phiF)
>>> eqT.solve(var=phiT)
```

Comparing to the correct analytical solution, $\phi = x$

```
>>> x = mesh.cellCenters[0]
>>> phiAnalytical.setValue(x)
>>> print(phiT.allclose(phiAnalytical, atol = 1e-8, rtol = 1e-8))
1
```

and finally, plot

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     Viewer(vars=(phiT, phiF)).plot()
...     input("Non-uniform thermal conductivity. Press <return> to proceed...")
```



Often, the diffusivity is not only non-uniform, but also depends on the value of the variable, such that

$$\frac{\partial \phi}{\partial t} = \nabla \cdot [D(\phi) \nabla \phi]. \quad (19.2)$$

With such a non-linearity, it is generally necessary to “sweep” the solution to convergence. This means that each time step should be calculated over and over, using the result of the previous sweep to update the coefficients of the equation, without advancing in time. In *FiPy*, this is accomplished by creating a solution variable that explicitly retains its “old” value by specifying `hasOld` when you create it. The variable does not move forward in time until it is explicitly told to `updateOld()`. In order to compare the effects of different numbers of sweeps, let us create a list of variables: `phi[0]` will be the variable that is actually being solved and `phi[1]` through `phi[4]` will display the result of taking the corresponding number of sweeps (`phi[1]` being equivalent to not sweeping at all).

```
>>> valueLeft = 1.  
>>> valueRight = 0.  
>>> phi = [  
...     CellVariable(name="solution variable",
```

(continues on next page)

(continued from previous page)

```

...         mesh=mesh,
...         value=valueRight,
...         hasOld=1),
...     CellVariable(name="1 sweep",
...                   mesh=mesh),
...     CellVariable(name="2 sweeps",
...                   mesh=mesh),
...     CellVariable(name="3 sweeps",
...                   mesh=mesh),
...     CellVariable(name="4 sweeps",
...                   mesh=mesh)
... ]

```

If, for example,

$$D = D_0(1 - \phi)$$

we would simply write Eq. (19.2) as

```

>>> D0 = 1.
>>> eq = TransientTerm() == DiffusionTerm(coeff=D0 * (1 - phi[0]))

```

Note: Because of the non-linearity, the Crank-Nicholson scheme does not work for this problem.

We apply the same boundary conditions that we used for the uniform diffusivity cases

```

>>> phi[0].constrain(valueRight, mesh.facesRight)
>>> phi[0].constrain(valueLeft, mesh.facesLeft)

```

Although this problem does not have an exact transient solution, it can be solved in steady-state, with

$$\phi(x) = 1 - \sqrt{\frac{x}{L}}$$

```

>>> x = mesh.cellCenters[0]
>>> phiAnalytical.setValue(1. - numerix.sqrt(x/L))

```

We create a viewer to compare the different numbers of sweeps with the analytical solution from before.

```

>>> if __name__ == '__main__':
...     viewer = Viewer(vars=phi + [phiAnalytical],
...                     datamin=0., datamax=1.)
...     viewer.plot()

```

As described above, an inner “sweep” loop is generally required for the solution of non-linear or multiple equation sets. Often a conditional is required to exit this “sweep” loop given some convergence criteria. Instead of using the `solve()` method equation, when sweeping, it is often useful to call `sweep()` instead. The `sweep()` method behaves the same way as `solve()`, but returns the residual that can then be used as part of the exit condition.

We now repeatedly run the problem with increasing numbers of sweeps.

```

>>> from fipy import input
>>> from builtins import range
>>> for sweeps in range(1, 5):

```

(continues on next page)

(continued from previous page)

```

...     phi[0].setValue(valueRight)
...     for step in range(steps):
...         # only move forward in time once per time step
...         phi[0].updateOld()
...
...         # but "sweep" many times per time step
...         for sweep in range(sweeps):
...             res = eq.sweep(var=phi[0],
...                             dt=timeStepDuration)
...             if __name__ == '__main__':
...                 viewer.plot()
...
...         # copy the final result into the appropriate display variable
...         phi[sweeps].setValue(phi[0])
...         if __name__ == '__main__':
...             viewer.plot()
...             input("Implicit variable diffusivity. %d sweep(s). \
... Residual = %f. Press <return> to proceed..." % (sweeps, (abs(res))))

```

As can be seen, sweeping does not dramatically change the result, but the “residual” of the equation (a measure of how accurately it has been solved) drops about an order of magnitude with each additional sweep.

Attention: Choosing an optimal balance between the number of time steps, the number of sweeps, the number of solver iterations, and the solver tolerance is more art than science and will require some experimentation on your part for each new problem.

Finally, we can increase the number of steps to approach equilibrium, or we can just solve for it directly

```
>>> eq = DiffusionTerm(coeff=D0 * (1 - phi[0]))
```

```

>>> phi[0].setValue(valueRight)
>>> res = 1e+10
>>> while res > 1e-6:
...     res = eq.sweep(var=phi[0],
...                     dt=timeStepDuration)

```

```

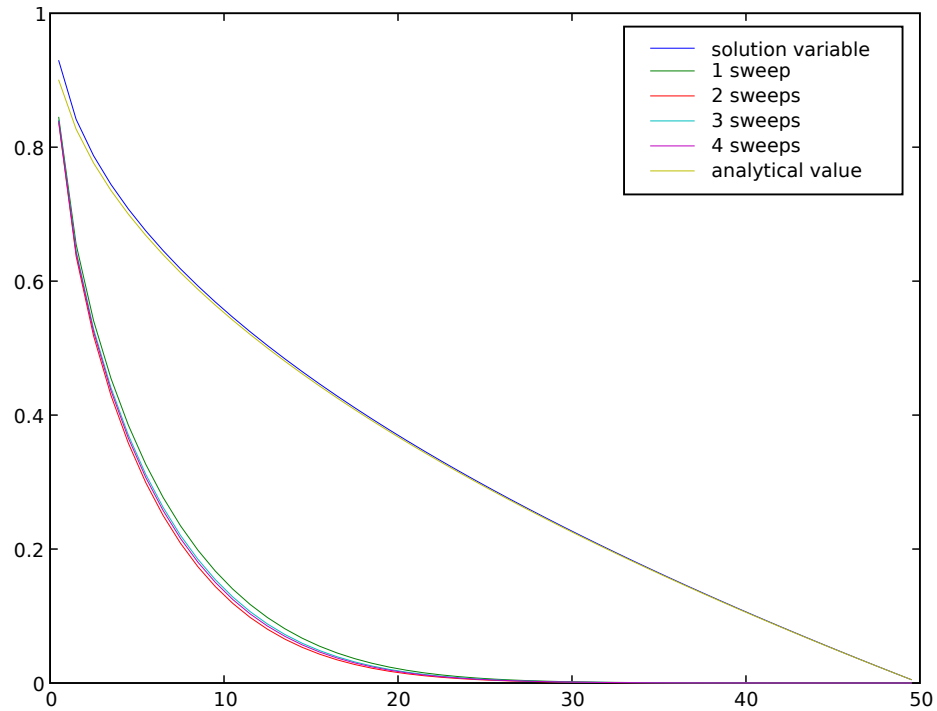
>>> print(phi[0].allclose(phiAnalytical, atol = 1e-1))
1

```

```

>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Implicit variable diffusivity - steady-state. \
... Press <return> to proceed...")

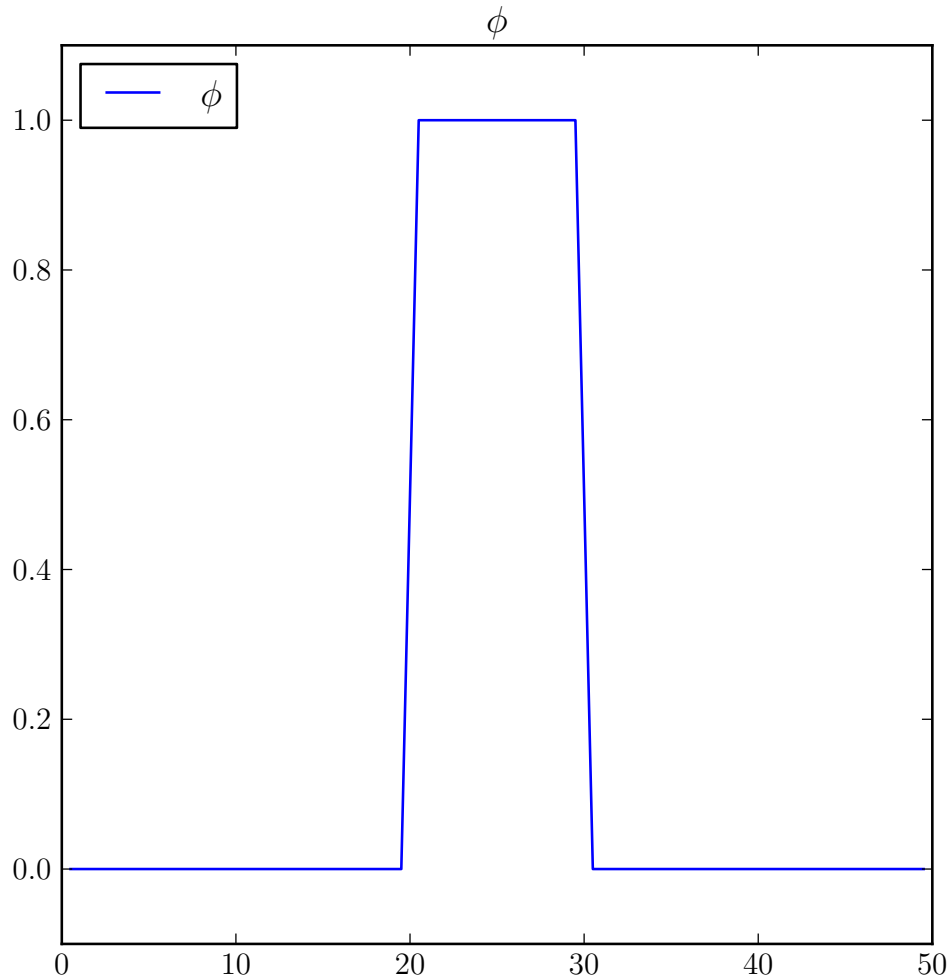
```



Fully implicit solutions are not without their pitfalls, particularly in steady state. Consider a localized block of material diffusing in a closed box.

```
>>> phi = CellVariable(mesh=mesh, name=r"$\phi$")
```

```
>>> phi.value = 0.
>>> phi.setValue(1., where=(x > L/2. - L/10.) & (x < L/2. + L/10.))
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=phi, datamin=-0.1, datamax=1.1)
```



We assign no explicit boundary conditions, leaving the default no-flux boundary conditions, and solve

$$\partial\phi/\partial t = \nabla \cdot (D\nabla\phi)$$

```
>>> D = 1.
>>> eq = TransientTerm() == DiffusionTerm(D)
```

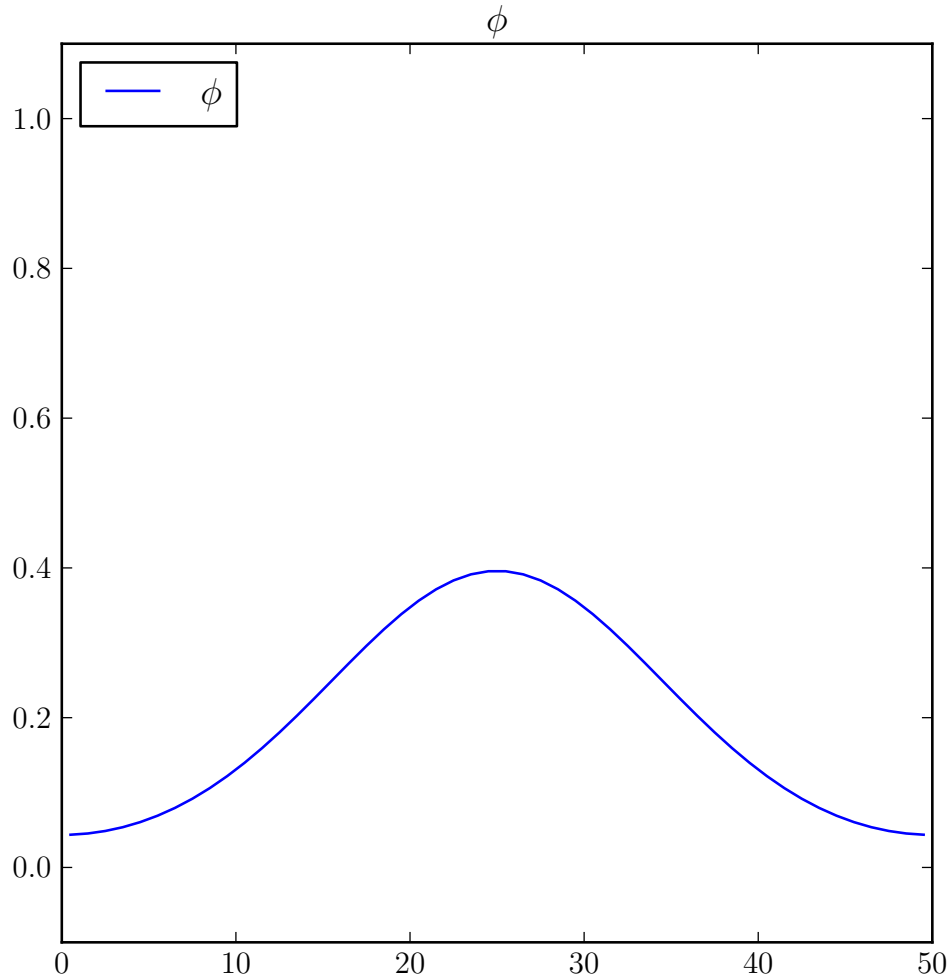
```
>>> dt = 10. * dx**2 / (2 * D)
>>> steps = 200
```

```
>>> from builtins import range
>>> for step in range(steps):
...     eq.solve(var=phi, dt=dt)
...     if __name__ == '__main__':
...         viewer.plot()
>>> from fipy import input
>>> if __name__ == '__main__':
```

(continues on next page)

(continued from previous page)

```
...     input("No-flux - transient. \n")
...     Press <return> to proceed...")
```



and see that ϕ dissipates to the expected average value of 0.2 with reasonable accuracy.

```
>>> print(numerix.allclose(phi, 0.2, atol=1e-5))
True
```

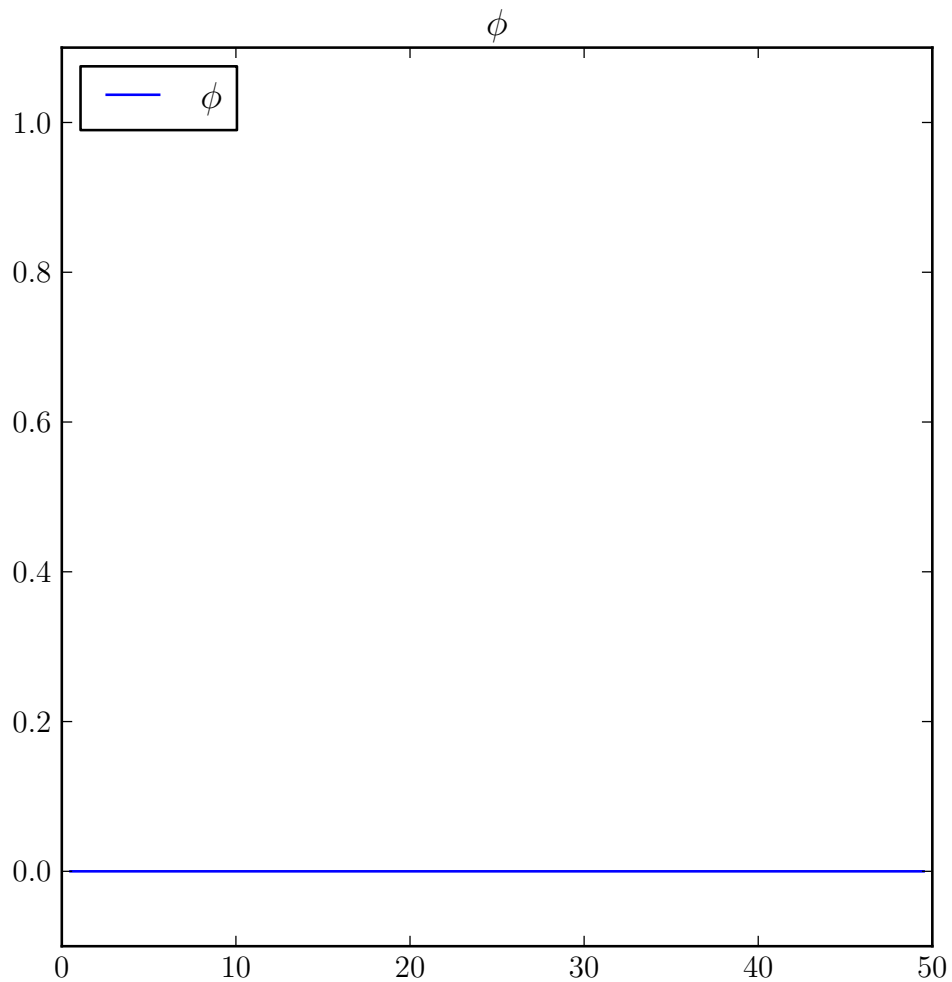
If we reset the initial condition

```
>>> phi.value = 0.
>>> phi.setValue(1., where=(x > L/2. - L/10.) & (x < L/2. + L/10.))
>>> if __name__ == '__main__':
...     viewer.plot()
```

and solve the steady-state problem

```
>>> DiffusionTerm(coeff=D).solve(var=phi)
>>> if __name__ == '__main__':
...     viewer.plot()
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("No-flux - steady-state failure. \
...     Press <return> to proceed...")
```

```
>>> print(numerix.allclose(phi, 0.0))
True
```



we find that the value is uniformly zero! What happened to our no-flux boundary conditions?

The problem is that in the implicit discretization of $\nabla \cdot (D \nabla \phi) = 0$,

$$\begin{vmatrix}
 \frac{D}{\Delta x^2} & -\frac{D}{\Delta x^2} & & & & \\
 & \ddots & \ddots & \ddots & & \\
 & & -\frac{D}{\Delta x^2} & \frac{2D}{\Delta x^2} & -\frac{D}{\Delta x^2} & \\
 & & & -\frac{D}{\Delta x^2} & \frac{2D}{\Delta x^2} & -\frac{D}{\Delta x^2} \\
 & & & & -\frac{D}{\Delta x^2} & \frac{2D}{\Delta x^2} & -\frac{D}{\Delta x^2} \\
 & & & & & \ddots & \ddots & \ddots \\
 & & & & & & -\frac{D}{\Delta x^2} & \frac{D}{\Delta x^2}
 \end{vmatrix}
 \begin{vmatrix}
 \phi_0^{\text{new}} \\
 \vdots \\
 \phi_{j-1}^{\text{new}} \\
 \phi_j^{\text{new}} \\
 \phi_{j+1}^{\text{new}} \\
 \vdots \\
 \phi_{N-1}^{\text{new}}
 \end{vmatrix}
 =
 \begin{vmatrix}
 0 \\
 \vdots \\
 0 \\
 0 \\
 0 \\
 \vdots \\
 0
 \end{vmatrix}$$

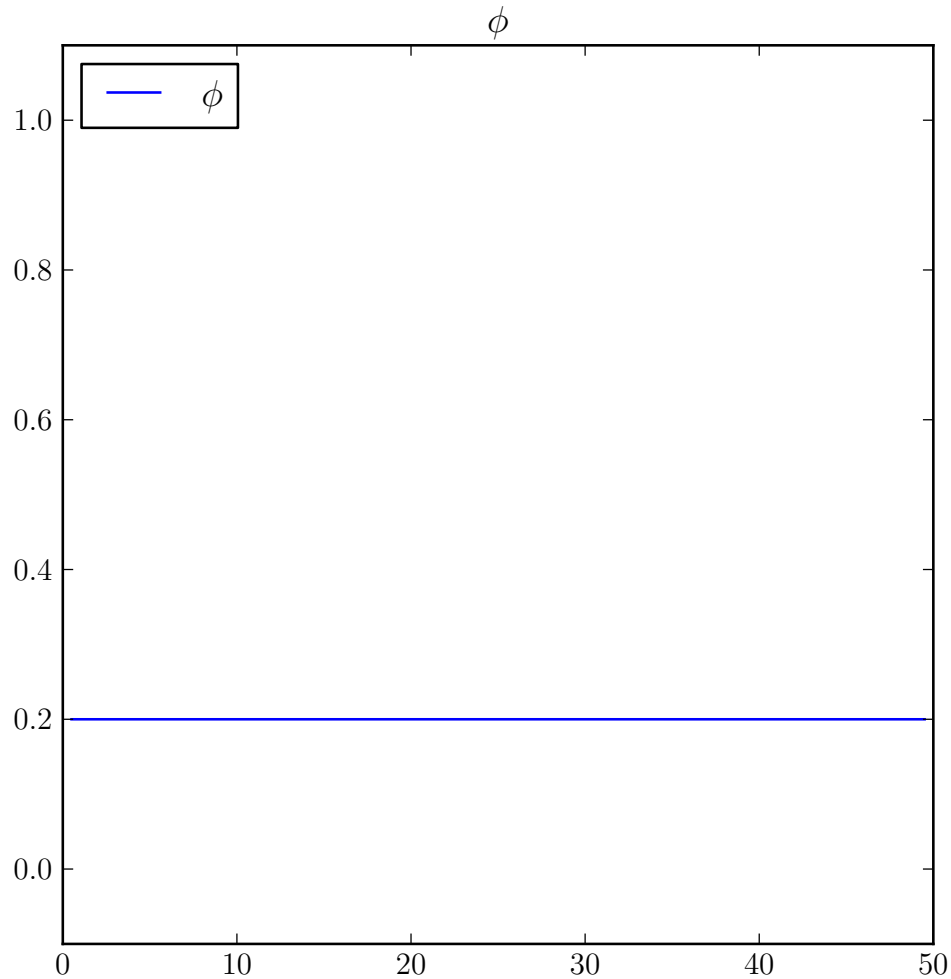
the initial condition ϕ^{old} no longer appears and $\phi = 0$ is a perfectly legitimate solution to this matrix equation.

The solution is to run the transient problem and to take one enormous time step

```
>>> phi.value = 0.
>>> phi.setValue(1., where=(x > L/2. - L/10.) & (x < L/2. + L/10.))
>>> if __name__ == '__main__':
...     viewer.plot()
```

```
>>> (TransientTerm() == DiffusionTerm(D)).solve(var=phi, dt=1e6*dt)
>>> if __name__ == '__main__':
...     viewer.plot()
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("No-flux - steady-state. \
... Press <return> to proceed...")
```

```
>>> print(numerix.allclose(phi, 0.2, atol=1e-5))
True
```



If this example had been written primarily as a script, instead of as documentation, we would delete every line that does not begin with either “>>>” or “. . .”, and then delete those prefixes from the remaining lines, leaving:

```
## This script was derived from
## 'examples/diffusion/mesh1D.py'

nx = 50
dx = 1.
mesh = Grid1D(nx = nx, dx = dx)
phi = CellVariable(name="solution variable",
                  mesh=mesh,
                  value=0)
```

```
eq = DiffusionTerm(coeff=D0 * (1 - phi[0]))
phi[0].setValue(valueRight)
res = 1e+10
while res > 1e-6:
```

(continues on next page)

(continued from previous page)

```

    res = eq.sweep(var=phi[0],
                   dt=timeStepDuration)

print phi[0].allclose(phiAnalytical, atol = 1e-1)
# Expect:
# 1
#
if __name__ == '__main__':
    viewer.plot()
    input("Implicit variable diffusivity - steady-state. \
Press <return> to proceed...")

```

Your own scripts will tend to look like this, although you can always write them as doctest scripts if you choose. You can obtain a plain script like this from some `.../example.py` by typing:

```
$ python setup.py copy_script --From .../example.py --To myExample.py
```

at the command line.

Most of the *FiPy* examples will be a mixture of plain scripts and doctest documentation/tests.

19.2 examples.diffusion.coupled

Solve the biharmonic equation as a coupled pair of diffusion equations.

FiPy has only first order time derivatives so equations such as the biharmonic wave equation written as

$$\frac{\partial^4 v}{\partial x^4} + \frac{\partial^2 v}{\partial t^2} = 0$$

cannot be represented as a single equation. We need to decompose the biharmonic equation into two equations that are first order in time in the following way,

$$\begin{aligned}\frac{\partial^2 v_0}{\partial x^2} + \frac{\partial v_1}{\partial t} &= 0 \\ \frac{\partial^2 v_1}{\partial x^2} - \frac{\partial v_0}{\partial t} &= 0\end{aligned}$$

Historically, *FiPy* required systems of coupled equations to be solved successively, “sweeping” the equations to convergence. As a practical example, we use the following system

$$\begin{aligned}\frac{\partial v_0}{\partial t} &= 0.01 \nabla^2 v_0 - \nabla^2 v_1 \\ \frac{\partial v_1}{\partial t} &= \nabla^2 v_0 + 0.01 \nabla^2 v_1\end{aligned}$$

subject to the boundary conditions

$$\begin{aligned}v_0|_{x=0} &= 0 & v_0|_{x=1} &= 1 \\ v_1|_{x=0} &= 1 & v_1|_{x=1} &= 0\end{aligned}$$

This system closely resembles the pure biharmonic equation, but has an additional diffusion contribution to improve numerical stability. The example system is solved with the following block of code using explicit coupling for the cross-coupled terms.

```
>>> from fipy import Grid1D, CellVariable, TransientTerm, DiffusionTerm, Viewer
```

```
>>> m = Grid1D(nx=100, Lx=1.)
```

```
>>> v0 = CellVariable(mesh=m, hasOld=True, value=0.5)
>>> v1 = CellVariable(mesh=m, hasOld=True, value=0.5)
```

```
>>> v0.constrain(0, m.facesLeft)
>>> v0.constrain(1, m.facesRight)
```

```
>>> v1.constrain(1, m.facesLeft)
>>> v1.constrain(0, m.facesRight)
```

```
>>> eq0 = TransientTerm() == DiffusionTerm(coeff=0.01) - v1.faceGrad.divergence
>>> eq1 = TransientTerm() == v0.faceGrad.divergence + DiffusionTerm(coeff=0.01)
```

```
>>> vi = Viewer((v0, v1))
```

```
>>> from builtins import range
>>> for t in range(100):
...     v0.updateOld()
...     v1.updateOld()
...     res0 = res1 = 1e100
...     while max(res0, res1) > 0.1:
...         res0 = eq0.sweep(var=v0, dt=1e-5)
...         res1 = eq1.sweep(var=v1, dt=1e-5)
...     if t % 10 == 0:
...         vi.plot()
```

The uncoupled method still works, but it can be advantageous to solve the two equations simultaneously. In this case, by coupling the equations, we can eliminate the explicit sources and dramatically increase the time steps:

```
>>> v0.value = 0.5
>>> v1.value = 0.5
```

```
>>> eqn0 = TransientTerm(var=v0) == DiffusionTerm(0.01, var=v0) - DiffusionTerm(1,
↳ var=v1)
>>> eqn1 = TransientTerm(var=v1) == DiffusionTerm(1, var=v0) + DiffusionTerm(0.01,
↳ var=v1)
```

```
>>> eqn = eqn0 & eqn1
```

```
>>> from builtins import range
>>> for t in range(1):
...     v0.updateOld()
...     v1.updateOld()
...     eqn.solve(dt=1.e-3)
...     vi.plot()
```

It is also possible to pose the same equations in vector form:

```
>>> v = CellVariable(mesh=m, hasOld=True, value=[[0.5], [0.5]], elementshape=(2,))
```

```
>>> v.constrain([[0], [1]], m.facesLeft)
>>> v.constrain([[1], [0]], m.facesRight)
```

```
>>> eqn = TransientTerm([[1, 0],
...                      [0, 1]]) == DiffusionTerm([[0.01, -1],
...                                                  [1, 0.01]]))
```

```
>>> vi = Viewer((v[0], v[1]))
```

```
>>> from builtins import range
>>> for t in range(1):
...     v.updateOld()
...     eqn.solve(var=v, dt=1.e-3)
...     vi.plot()
```

Whether you pose your problem in coupled or vector form should be dictated by the underlying physics. If v_0 and v_1 represent the concentrations of two conserved species, then it is natural to write two separate governing equations and to couple them. If they represent two components of a vector field, then the vector formulation is obviously more natural. FiPy will solve the same matrix system either way.

19.3 examples.diffusion.mesh20x20

Solve a two-dimensional diffusion problem in a square domain.

This example solves a diffusion problem and demonstrates the use of applying boundary condition patches.

```
>>> from fipy import CellVariable, Grid2D, Viewer, TransientTerm, DiffusionTerm
>>> from fipy.tools import numerix
```

```
>>> nx = 20
>>> ny = nx
>>> dx = 1.
>>> dy = dx
>>> L = dx * nx
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

We create a *CellVariable* and initialize it to zero:

```
>>> phi = CellVariable(name = "solution variable",
...                     mesh = mesh,
...                     value = 0.)
```

and then create a diffusion equation. This is solved by default with an iterative conjugate gradient solver.

```
>>> D = 1.
>>> eq = TransientTerm() == DiffusionTerm(coeff=D)
```

We apply Dirichlet boundary conditions

```
>>> valueTopLeft = 0
>>> valueBottomRight = 1
```

to the top-left and bottom-right corners. Neumann boundary conditions are automatically applied to the top-right and bottom-left corners.

```
>>> X, Y = mesh.faceCenters
>>> facesTopLeft = ((mesh.facesLeft & (Y > L / 2))
...                 | (mesh.facesTop & (X < L / 2)))
>>> facesBottomRight = ((mesh.facesRight & (Y < L / 2))
...                     | (mesh.facesBottom & (X > L / 2)))
```

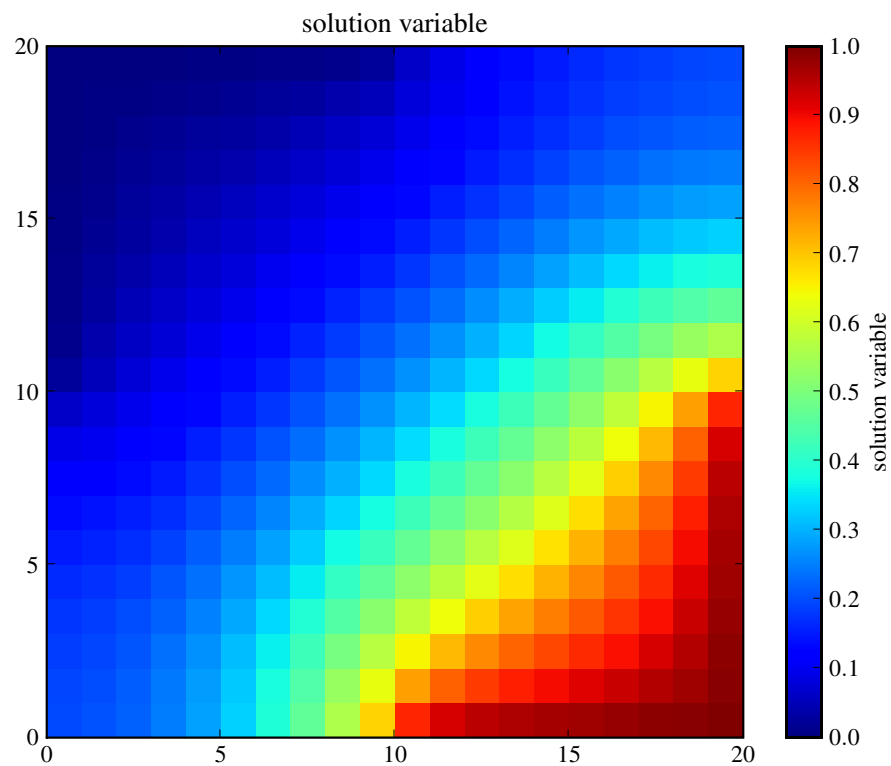
```
>>> phi.constrain(valueTopLeft, facesTopLeft)
>>> phi.constrain(valueBottomRight, facesBottomRight)
```

We create a viewer to see the results

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=phi, datamin=0., datamax=1.)
...     viewer.plot()
```

and solve the equation by repeatedly looping in time:

```
>>> timeStepDuration = 10 * 0.9 * dx**2 / (2 * D)
>>> steps = 10
>>> from builtins import range
>>> for step in range(steps):
...     eq.solve(var=phi,
...              dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
```



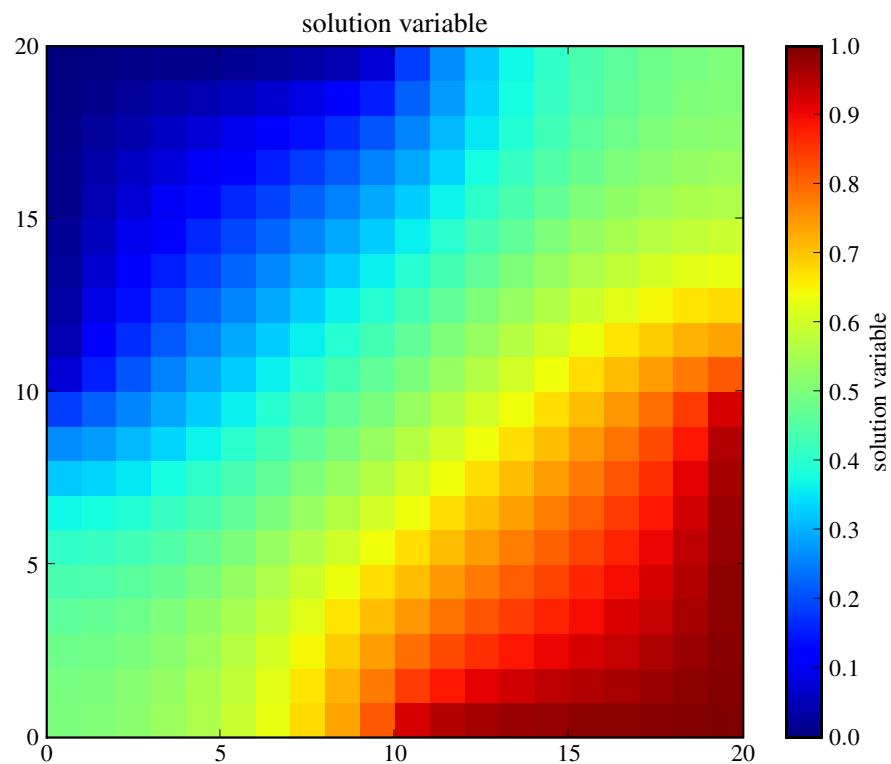
We can test the value of the bottom-right corner cell.

```
>>> print(numerix.allclose(phi((L,),(0,)), valueBottomRight, atol = 1e-2))
1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Implicit transient diffusion. Press <return> to proceed...")
```

We can also solve the steady-state problem directly

```
>>> DiffusionTerm().solve(var=phi)
>>> if __name__ == '__main__':
...     viewer.plot()
```



and test the value of the bottom-right corner cell.

```
>>> print(numerix.allclose(phi((L,),(0,)), valueBottomRight, atol = 1e-2))
1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Implicit steady-state diffusion. Press <return> to proceed...")
```

19.4 examples.diffusion.circle

Solve the diffusion equation in a circular domain meshed with triangles.

This example demonstrates how to solve a simple diffusion problem on a non-standard mesh with varying boundary conditions. The *Gmsh* package is used to create the mesh. Firstly, define some parameters for the creation of the mesh,

```
>>> cellSize = 0.05
>>> radius = 1.
```

The *cellSize* is the preferred edge length of each mesh element and the *radius* is the radius of the circular mesh domain. In the following code section a file is created with the geometry that describes the mesh. For details of how to write such geometry files for *Gmsh*, see the *gmsh manual*.

The mesh created by *Gmsh* is then imported into *FiPy* using the *Gmsh2D* object.

```
>>> from fipy import CellVariable, Gmsh2D, TransientTerm, DiffusionTerm, Viewer
>>> from fipy.tools import numerix
```

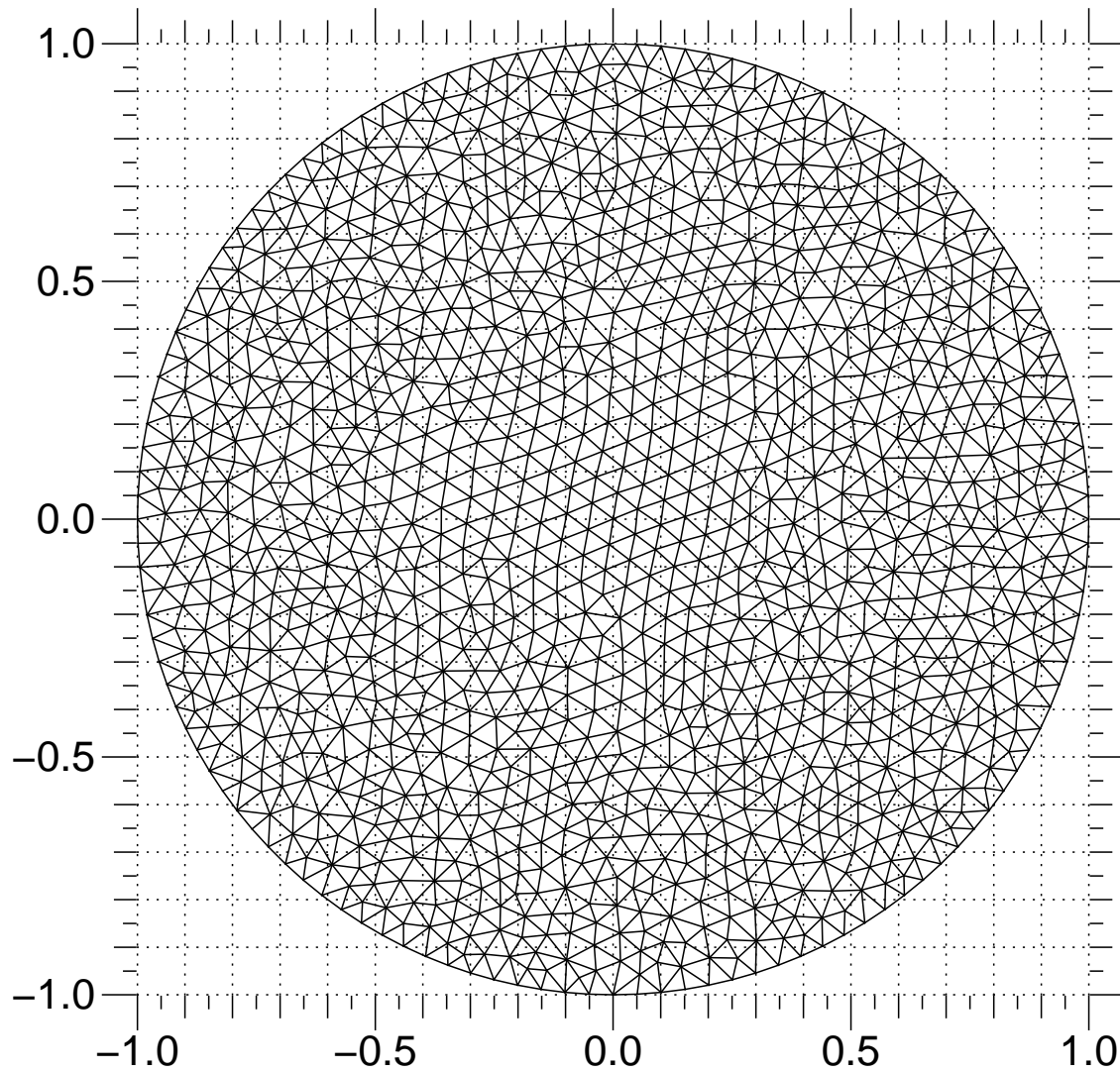
```
>>> mesh = Gmsh2D('''
...     cellSize = %(cellSize)g;
...     radius = %(radius)g;
...     Point(1) = {0, 0, 0, cellSize};
...     Point(2) = {-radius, 0, 0, cellSize};
...     Point(3) = {0, radius, 0, cellSize};
...     Point(4) = {radius, 0, 0, cellSize};
...     Point(5) = {0, -radius, 0, cellSize};
...     Circle(6) = {2, 1, 3};
...     Circle(7) = {3, 1, 4};
...     Circle(8) = {4, 1, 5};
...     Circle(9) = {5, 1, 2};
...     Line Loop(10) = {6, 7, 8, 9};
...     Plane Surface(11) = {10};
...     ''' % locals())
```

Using this mesh, we can construct a solution variable

```
>>> phi = CellVariable(name = "solution variable",
...                     mesh = mesh,
...                     value = 0.)
```

We can now create a *Viewer* to see the mesh

```
>>> viewer = None
>>> from fipy import input
>>> if __name__ == '__main__':
...     try:
...         viewer = Viewer(vars=phi, datamin=-1, datamax=1.)
...         viewer.plotMesh()
...         input("Irregular circular mesh. Press <return> to proceed...")
...     except:
...         print("Unable to create a viewer for an irregular mesh (try_
↪Matplotlib2DViewer or MayaviViewer)")
```



We set up a transient diffusion equation

```
>>> D = 1.
>>> eq = TransientTerm() == DiffusionTerm(coeff=D)
```

The following line extracts the x coordinate values on the exterior faces. These are used as the boundary condition fixed values.

```
>>> X, Y = mesh.faceCenters
```

```
>>> phi.constrain(X, mesh.exteriorFaces)
```

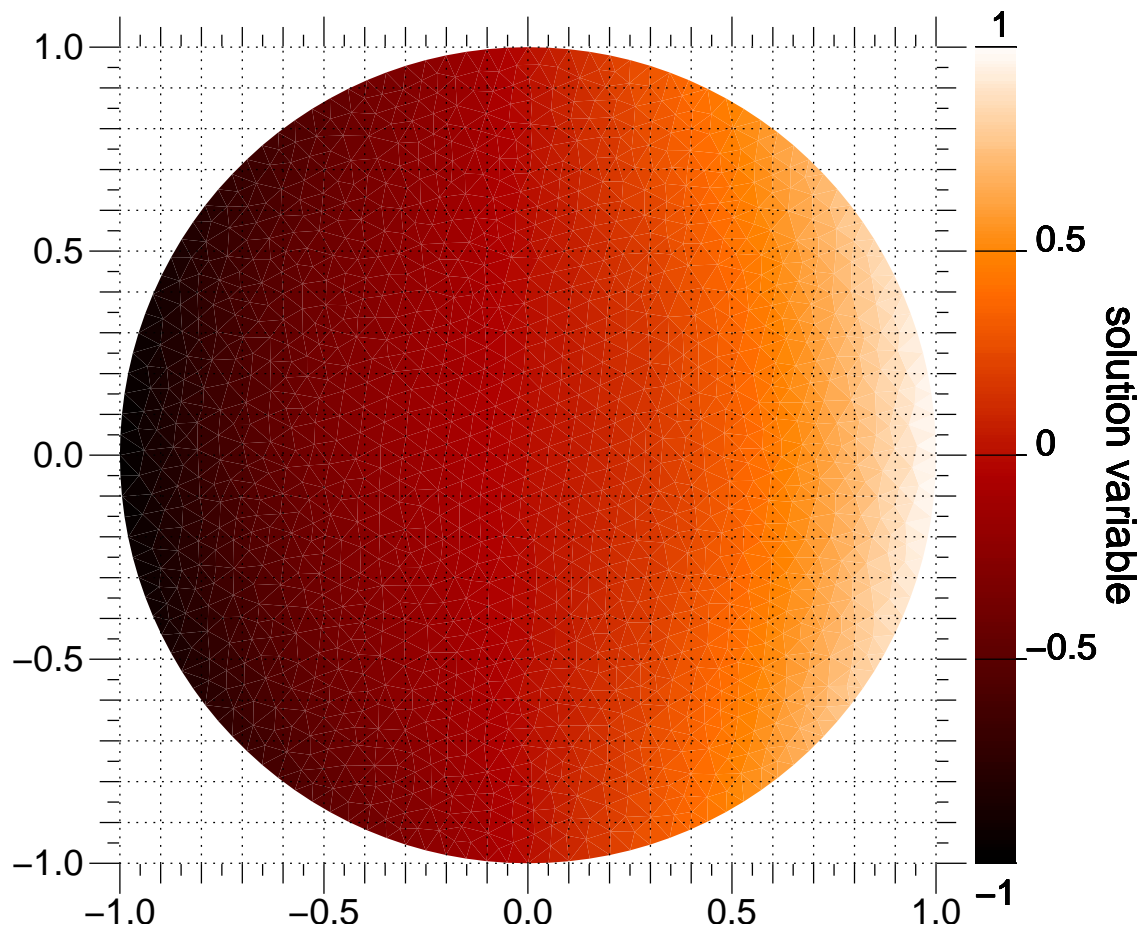
We first step through the transient problem

```
>>> timeStepDuration = 10 * 0.9 * cellSize**2 / (2 * D)
>>> steps = 10
>>> from builtins import range
>>> for step in range(steps):
...     eq.solve(var=phi,
...               dt=timeStepDuration)
```

(continues on next page)

(continued from previous page)

```
... if viewer is not None:
...     viewer.plot()
```



If we wanted to plot or analyze the results of this calculation with another application, we could export tab-separated-values with

```
TSVViewer(vars=(phi, phi.grad)).plot(filename="myTSV.tsv")
```

x	y	solution variable	solution variable_grad_x	solution_
↪variable_grad_y				
0.975559734792414		0.0755414402612554	0.964844363287199	-0.
↪229687917881182		0.00757854476106331		
0.0442864953037566		0.79191893162384	0.0375859836421991	-0.
↪773936613923853		-0.205560697612547		
0.0246775505084069		0.771959648896982	0.020853932412869	-0.
↪723540342405813		-0.182589694334729		
0.223345558247991		-0.807931073108895	0.203035857140125	-0.
↪777466238738658		0.0401235242511506		
-0.00726763301939488		-0.775978916110686	-0.00412895434496877	-0.
↪650055516507232		-0.183112882869288		
-0.0220279064527904		-0.187563765977912	-0.012771874945585	-0.
↪35707168379437		-0.056072788439713		
0.111223320911545		-0.679586798311355	0.0911595298310758	-0.
↪613455176718145		0.0256182541329463		

(continues on next page)

(continued from previous page)

-0.78996770899909	-0.0173672729866294	-0.693887874335319	-1.
↪00671109050419	-0.127611490372511		
-0.703545986179876	-0.435813500559859	-0.635004192597412	-0.
↪896203033957194	-0.00855563518923689		
0.888641841567831	-0.408558914368324	0.877939107374768	-0.
↪32195762184087	-0.22696791637322		
0.38212257821916	-0.51732949653553	0.292889724306196	-0.
↪854466141879776	0.199715815696975		
-0.359068256998365	0.757882581524374	-0.323541041763627	-0.
↪870534227755687	0.0792631912863636		
-0.459673905457569	-0.701526587772079	-0.417577664032421	-0.
↪725460726303266	-0.119132299176163		
-0.338256179134518	-0.523565732643067	-0.254030052182524	-0.
↪923505840608445	-0.192224240688976		
0.87498754712638	0.174119064688993	0.836057900916614	-1.
↪11590500805745	-0.211010116496191		
-0.484106960369249	0.0705987421869745	-0.319827850867342	-0.
↪867894407968447	0.051246727010685		
-0.0221203060940465	-0.216026820080053	-0.0152729438559779	-0.
↪341246696530392	-0.0538476142281317		

The values are listed at the cell centers. Particularly for irregular meshes, no specific ordering should be relied upon. Vector quantities are listed in multiple columns, one for each mesh dimension.

This problem again has an analytical solution that depends on the error function, but it's a bit more complicated due to the varying boundary conditions and the different horizontal diffusion length at different vertical positions

```
>>> x, y = mesh.cellCenters
>>> t = timeStepDuration * steps
```

```
>>> phiAnalytical = CellVariable(name="analytical value",
...                               mesh=mesh)
```

```
>>> x0 = radius * numerix.cos(numerix.arcsin(y))
>>> try:
...     from scipy.special import erf
...     ## This function can sometimes throw nans on OS X
...     ## see http://projects.scipy.org/scipy/scipy/ticket/325
...     phiAnalytical.setValue(x0 * (erf((x0+x) / (2 * numerix.sqrt(D * t)))
...                                   - erf((x0-x) / (2 * numerix.sqrt(D * t)))))
... except ImportError:
...     print("The SciPy library is not available to test the solution to \
... the transient diffusion equation")
```

```
>>> print(phi.allclose(phiAnalytical, atol = 7e-2))
1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Transient diffusion. Press <return> to proceed...")
```

As in the earlier examples, we can also directly solve the steady-state diffusion problem.

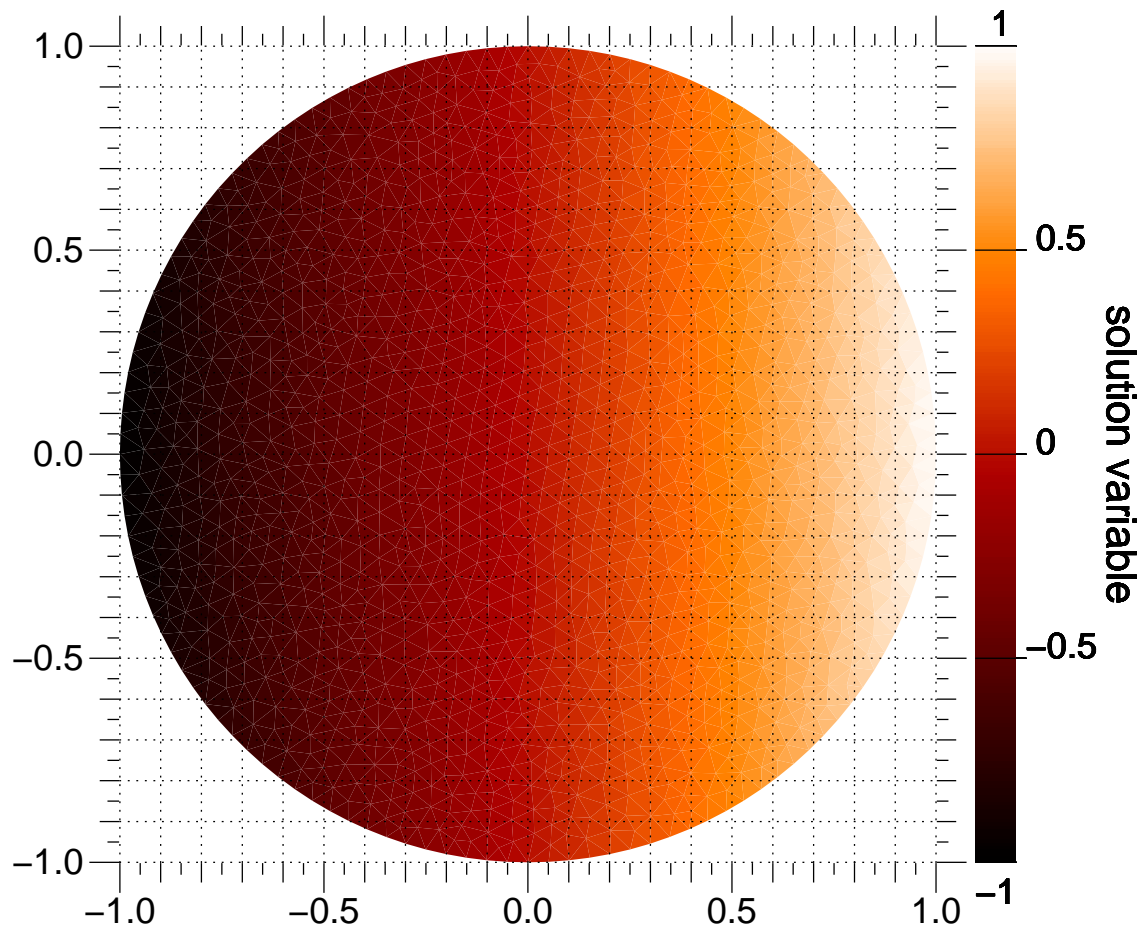
```
>>> DiffusionTerm(coeff=D).solve(var=phi)
```

The values at the elements should be equal to their x coordinate

```
>>> print(phi.allclose(x, atol = 0.03))
1
```

Display the results if run as a script.

```
>>> from fipy import input
>>> if viewer is not None:
...     viewer.plot()
...     input("Steady-state diffusion. Press <return> to proceed...")
```



19.5 examples.diffusion.electrostatics

Solve the Poisson equation in one dimension.

The Poisson equation is a particular example of the steady-state diffusion equation. We examine a few cases in one dimension.

```
>>> from fipy import CellVariable, Grid1D, Viewer, DiffusionTerm
```

```
>>> nx = 200
>>> dx = 0.01
>>> L = nx * dx
>>> mesh = Grid1D(dx = dx, nx = nx)
```

Given the electrostatic potential ϕ ,

```
>>> potential = CellVariable(mesh=mesh, name='potential', value=0.)
```

the permittivity ϵ ,

```
>>> permittivity = 1
```

the concentration C_j of the j^{th} component with valence z_j (we consider only a single component C_{e^-} with valence with $z_{e^-} = -1$)

```
>>> electrons = CellVariable(mesh=mesh, name='e-')
>>> electrons.valence = -1
```

and the charge density ρ ,

```
>>> charge = electrons * electrons.valence
>>> charge.name = "charge"
```

The dimensionless Poisson equation is

$$\nabla \cdot (\epsilon \nabla \phi) = -\rho = -\sum_{j=1}^n z_j C_j$$

```
>>> potential.equation = (DiffusionTerm(coeff = permittivity)
...                       + charge == 0)
```

Because this equation admits an infinite number of potential profiles, we must constrain the solution by fixing the potential at one point:

```
>>> potential.constrain(0., mesh.facesLeft)
```

First, we obtain a uniform charge distribution by setting a uniform concentration of electrons $C_{e^-} = 1$.

```
>>> electrons.setValue(1.)
```

and we solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential)
```

This problem has the analytical solution

$$\psi(x) = \frac{x^2}{2} - 2x$$

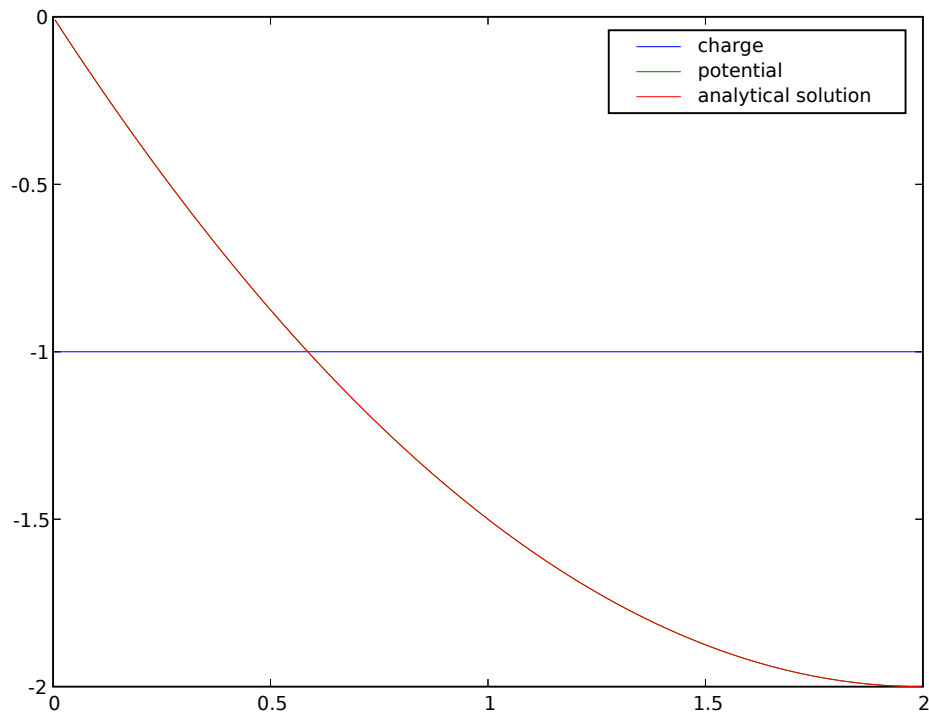
```
>>> x = mesh.cellCenters[0]
>>> analytical = CellVariable(mesh=mesh, name="analytical solution",
...                           value=(x**2)/2 - 2*x)
```

which has been satisfactorily obtained

```
>>> print(potential.allclose(analytical, rtol = 2e-5, atol = 2e-5))
1
```

If we are running the example interactively, we view the result

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(charge, potential, analytical))
...     viewer.plot()
...     input("Press any key to continue...")
```



Next, we segregate all of the electrons to right side of the domain

$$C_{e^-} = \begin{cases} 0 & \text{for } x \leq L/2, \\ 1 & \text{for } x > L/2. \end{cases}$$

```
>>> x = mesh.cellCenters[0]
>>> electrons.setValue(0.)
>>> electrons.setValue(1., where=x > L / 2.)
```

and again solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential)
```

which now has the analytical solution

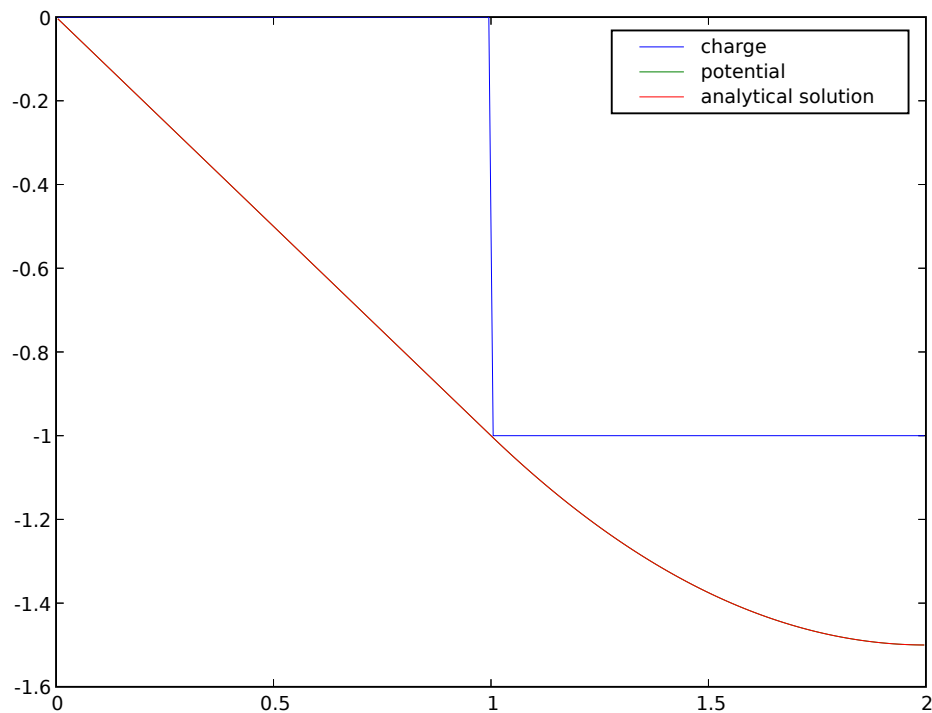
$$\psi(x) = \begin{cases} -x & \text{for } x \leq L/2, \\ \frac{(x-1)^2}{2} - x & \text{for } x > L/2. \end{cases}$$

```
>>> analytical.setValue(-x)
>>> analytical.setValue(((x-1)**2)/2 - x, where=x > L/2)
```

```
>>> print(potential.allclose(analytical, rtol = 2e-5, atol = 2e-5))
1
```

and again view the result

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Press any key to continue...")
```



Finally, we segregate all of the electrons to the left side of the domain

$$C_{e^-} = \begin{cases} 1 & \text{for } x \leq L/2, \\ 0 & \text{for } x > L/2. \end{cases}$$

```
>>> electrons.setValue(1.)
>>> electrons.setValue(0., where=x > L / 2.)
```

and again solve for the electrostatic potential

```
>>> potential.equation.solve(var=potential)
```

which has the analytical solution

$$\psi(x) = \begin{cases} \frac{x^2}{2} - x & \text{for } x \leq L/2, \\ -\frac{1}{2} & \text{for } x > L/2. \end{cases}$$

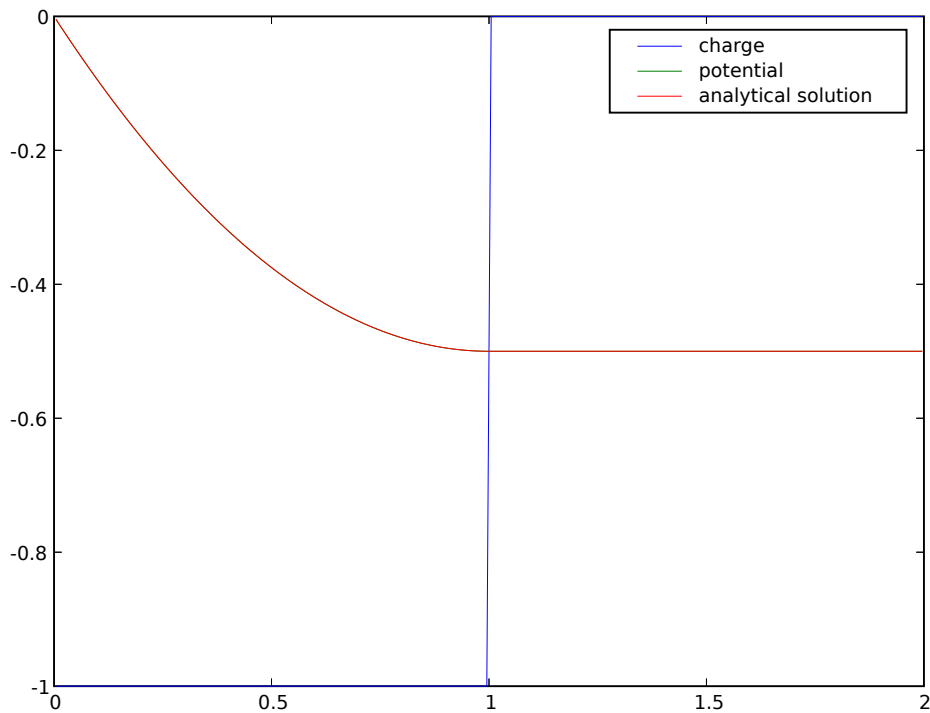
We again verify that the correct equilibrium is attained

```
>>> analytical.setValue((x**2)/2 - x)
>>> analytical.setValue(-0.5, where=x > L / 2)
```

```
>>> print(potential.allclose(analytical, rtol = 2e-5, atol = 2e-5))
1
```

and once again view the result

```
>>> if __name__ == '__main__':
...     viewer.plot()
```



19.6 examples.diffusion.nthOrder.input4thOrder1D

Solve a fourth-order diffusion problem.

This example uses the *DiffusionTerm* class to solve the equation

$$\frac{\partial^4 \phi}{\partial x^4} = 0$$

on a 1D mesh of length

```
>>> L = 1000.
```

We create an appropriate mesh

```
>>> from fipy import CellVariable, Grid1D, NthOrderBoundaryCondition, DiffusionTerm, \
↳ Viewer, GeneralSolver
```

```
>>> nx = 500
>>> dx = L / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

and initialize the solution variable to 0

```
>>> var = CellVariable(mesh=mesh, name='solution variable')
```

For this problem, we impose the boundary conditions:

$$\begin{aligned} \phi &= \alpha_1 & \text{at } x = 0 \\ \frac{\partial \phi}{\partial x} &= \alpha_2 & \text{at } x = L \\ \frac{\partial^2 \phi}{\partial x^2} &= \alpha_3 & \text{at } x = 0 \\ \frac{\partial^3 \phi}{\partial x^3} &= \alpha_4 & \text{at } x = L. \end{aligned}$$

or

```
>>> alpha1 = 2.
>>> alpha2 = 1.
>>> alpha3 = 4.
>>> alpha4 = -3.
```

```
>>> BCs = (NthOrderBoundaryCondition(faces=mesh.facesLeft, value=alpha3, order=2),
...        NthOrderBoundaryCondition(faces=mesh.facesRight, value=alpha4, order=3))
>>> var.faceGrad.constrain([alpha2], mesh.facesRight)
>>> var.constrain(alpha1, mesh.facesLeft)
```

We initialize the steady-state equation

```
>>> eq = DiffusionTerm(coeff=(1, 1)) == 0
```

```
>>> import fipy.solvers.solver
>>> if fipy.solvers.solver == 'petsc':
...     solver = GeneralSolver(precon='lu')
... else:
...     solver = GeneralSolver()
```

We perform one implicit timestep to achieve steady state

```
>>> eq.solve(var=var,  
...         boundaryConditions=BCs,  
...         solver=solver)
```

The analytical solution is:

$$\phi = \frac{\alpha_4}{6}x^3 + \frac{\alpha_3}{2}x^2 + \left(\alpha_2 - \frac{\alpha_4}{2}L^2 - \alpha_3L\right)x + \alpha_1$$

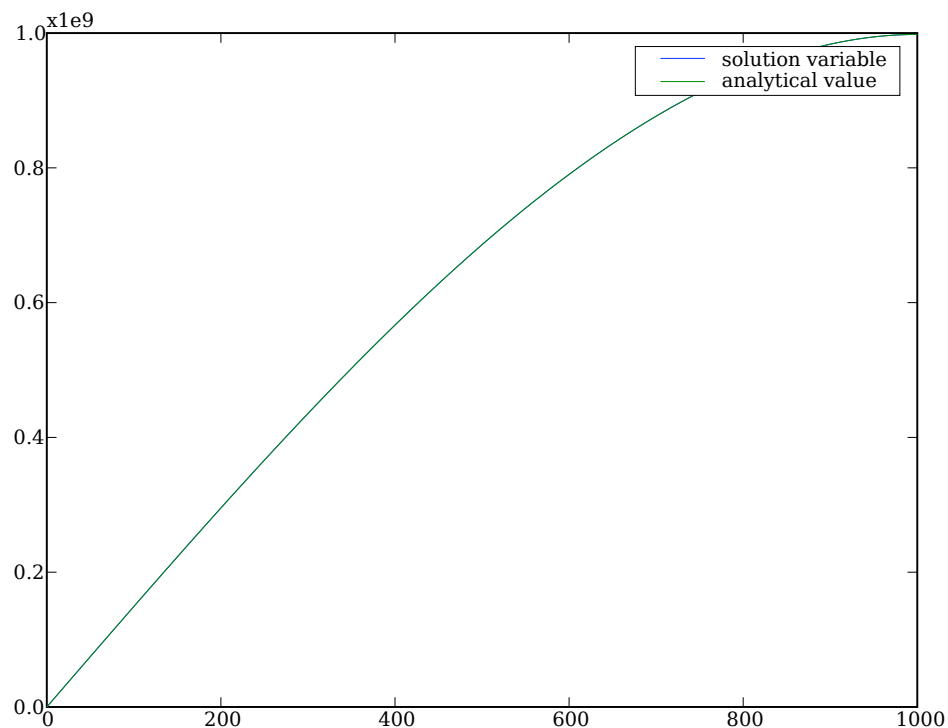
or

```
>>> analytical = CellVariable(mesh=mesh, name='analytical value')  
>>> x = mesh.cellCenters[0]  
>>> analytical.setValue(alpha4 / 6. * x**3 + alpha3 / 2. * x**2 + \  
...                     (alpha2 - alpha4 / 2. * L**2 - alpha3 * L) * x + alpha1)
```

```
>>> print(var.allclose(analytical, rtol=1e-4))  
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':  
...     viewer = Viewer(vars=(var, analytical))  
...     viewer.plot()
```



19.7 examples.diffusion.anisotropy

Solve the diffusion equation with an anisotropic diffusion coefficient.

We wish to solve the problem

$$\frac{\partial \phi}{\partial t} = \partial_j \Gamma_{ij} \partial_i \phi$$

on a circular domain centered at $(0, 0)$. We can choose an anisotropy ratio of 5 such that

$$\Gamma' = \begin{pmatrix} 0.2 & 0 \\ 0 & 1 \end{pmatrix}$$

A new matrix is formed by rotating Γ' such that

$$R = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix}$$

and

$$\Gamma = R \Gamma' R^T$$

In the case of a point source at $(0, 0)$ a reference solution is given by,

$$\phi(X, Y, t) = Q \frac{\exp\left(-\frac{1}{4t} \left(\frac{X^2}{\Gamma'_{00}} + \frac{Y^2}{\Gamma'_{11}}\right)\right)}{4\pi t \sqrt{\Gamma'_{00} \Gamma'_{11}}}$$

where $(X, Y)^T = R(x, y)^T$ and Q is the initial mass.

```
>>> from fipy import CellVariable, Gmsh2D, Viewer, TransientTerm, \
↳ DiffusionTermCorrection
>>> from fipy.tools import serialComm, numerix
```

Import a mesh previously created using *Gmsh*.

```
>>> import os
>>> mesh = Gmsh2D(os.path.splitext(__file__)[0] + '.msh', communicator=serialComm)
```

Set the center-most cell to have a value.

```
>>> var = CellVariable(mesh=mesh, hasOld=1)
>>> x, y = mesh.cellCenters
>>> var[numerix.argmin(x**2 + y**2)] = 1.
```

Choose an orientation for the anisotropy.

```
>>> theta = numerix.pi / 4.
>>> rotationMatrix = numerix.array(((numerix.cos(theta), numerix.sin(theta)), \
...                                  (-numerix.sin(theta), numerix.cos(theta))))
>>> gamma_prime = numerix.array(((0.2, 0.), (0., 1.)))
>>> DOT = numerix.NUMERIX.dot
>>> gamma = DOT(DOT(rotationMatrix, gamma_prime), numerix.transpose(rotationMatrix))
```

Make the equation, viewer and solve.

```
>>> eqn = TransientTerm() == DiffusionTermCorrection((gamma,))
```

```
>>> if __name__ == '__main__':
...     viewer = Viewer(var, datamin=0.0, datamax=0.001)
```

```
>>> mass = float(var.cellVolumeAverage * numerix.sum(mesh.cellVolumes))
>>> time = 0
>>> dt=0.00025
```

```
>>> from builtins import range
>>> for i in range(20):
...     var.updateOld()
...     res = 1.
...
...     while res > 1e-2:
...         res = eqn.sweep(var, dt=dt)
...
...     if __name__ == '__main__':
...         viewer.plot()
...     time += dt
```

Compare with the analytical solution (within 5% accuracy).

```
>>> X, Y = numerix.dot(mesh.cellCenters, CellVariable(mesh=mesh, rank=2,
↳value=rotationMatrix))
>>> solution = mass * numerix.exp(-(X**2 / gamma_prime[0][0] + Y**2 / gamma_
↳prime[1][1]) / (4 * time)) / (4 * numerix.pi * time * numerix.sqrt(gamma_
↳prime[0][0] * gamma_prime[1][1]))
>>> print(max(abs((var - solution) / max(solution))) < 0.08)
True
```

Chapter 20

Convection Examples

<code>examples.convection.exponential1D.mesh1D</code>	Solve the steady-state convection-diffusion equation in one dimension.
<code>examples.convection.exponential1DSource.mesh1D</code>	Solve the steady-state convection-diffusion equation with a constant source.
<code>examples.convection.robin</code>	Solve an advection-diffusion equation with a Robin boundary condition.
<code>examples.convection.source</code>	Solve a convection problem with a source.

20.1 examples.convection.exponential1D.mesh1D

Solve the steady-state convection-diffusion equation in one dimension.

This example solves the steady-state convection-diffusion equation given by

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = 10\hat{x}$, or

```
>>> diffCoeff = 1.  
>>> convCoeff = (10.,)
```

We define a 1D mesh

```
>>> from fipy import CellVariable, Grid1D, DiffusionTerm, ExponentialConvectionTerm,   
↳ Viewer  
>>> from fipy.tools import numerix
```

```
>>> L = 10.  
>>> nx = 10  
>>> mesh = Grid1D(dx=L / nx, nx=nx)
```

```
>>> valueLeft = 0.  
>>> valueRight = 1.
```

The solution variable is initialized to valueLeft:

```
>>> var = CellVariable(mesh=mesh, name="variable")
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 0, \\ 1 & \text{at } x = L, \end{cases}$$

with

```
>>> var.constrain(valueLeft, mesh.facesLeft)
>>> var.constrain(valueRight, mesh.facesRight)
```

The equation is created with the *DiffusionTerm* and *ExponentialConvectionTerm*. The scheme used by the convection term needs to calculate a Péclet number and thus the diffusion term instance must be passed to the convection term.

```
>>> eq = (DiffusionTerm(coeff=diffCoeff)
...       + ExponentialConvectionTerm(coeff=convCoeff))
```

More details of the benefits and drawbacks of each type of convection term can be found in *Numerical Schemes*. Essentially, the *ExponentialConvectionTerm* and *PowerLawConvectionTerm* will both handle most types of convection-diffusion cases, with the *PowerLawConvectionTerm* being more efficient.

We solve the equation

```
>>> eq.solve(var=var)
```

and test the solution against the analytical result

$$\phi = \frac{1 - \exp(-u_x x / D)}{1 - \exp(-u_x L / D)}$$

or

```
>>> axis = 0
>>> x = mesh.cellCenters[axis]
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> print(var.allclose(analyticalArray))
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var)
...     viewer.plot()
```

20.2 examples.convection.exponential1DSource.mesh1D

Solve the steady-state convection-diffusion equation with a constant source.

Like `examples.convection.exponential1D.mesh1D` this example solves a steady-state convection-diffusion equation, but adds a constant source, $S_0 = 1$, such that

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) + S_0 = 0.$$

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,)
>>> sourceCoeff = 1.
```

We define a 1D mesh

```
>>> from fipy import CellVariable, Grid1D, DiffusionTerm, ExponentialConvectionTerm, \
↳ DefaultAsymmetricSolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> nx = 1000
>>> L = 10.
>>> mesh = Grid1D(dx=L / 1000, nx=nx)
```

```
>>> valueLeft = 0.
>>> valueRight = 1.
```

The solution variable is initialized to valueLeft:

```
>>> var = CellVariable(name="variable", mesh=mesh)
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 0, \\ 1 & \text{at } x = L, \end{cases}$$

with

```
>>> var.constrain(valueLeft, mesh.facesLeft)
>>> var.constrain(valueRight, mesh.facesRight)
```

We define the convection-diffusion equation with source

```
>>> eq = (DiffusionTerm(coeff=diffCoeff)
...       + ExponentialConvectionTerm(coeff=convCoeff)
...       + sourceCoeff)
```

```
>>> eq.solve(var=var,
...          solver=DefaultAsymmetricSolver(tolerance=1.e-15, iterations=10000))
```

and test the solution against the analytical result:

$$\phi = -\frac{S_0 x}{u_x} + \left(1 + \frac{S_0 x}{u_x}\right) \frac{1 - \exp(-u_x x/D)}{1 - \exp(-u_x L/D)}$$

or

```
>>> axis = 0
>>> x = mesh.cellCenters[axis]
>>> AA = -sourceCoeff * x / convCoeff[axis]
>>> BB = 1. + sourceCoeff * L / convCoeff[axis]
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = AA + BB * CC / DD
>>> print(var.allclose(analyticalArray, rtol=1e-4, atol=1e-4))
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var)
...     viewer.plot()
```

20.3 examples.convection.robin

Solve an advection-diffusion equation with a Robin boundary condition.

This example demonstrates how to apply a Robin boundary condition to an advection-diffusion equation. The equation we wish to solve is given by,

$$\begin{aligned} 0 &= \frac{\partial^2 C}{\partial x^2} - P \frac{\partial C}{\partial x} - DC & 0 < x < 1 \\ x = 0 : P &= -\frac{\partial C}{\partial x} + PC \\ x = 1 : \frac{\partial C}{\partial x} &= 0 \end{aligned}$$

The analytical solution for this equation is given by,

$$C(x) = \frac{2P \exp\left(\frac{Px}{2}\right) \left[(P+A) \exp\left(\frac{A}{2}(1-x)\right) - (P-A) \exp\left(-\frac{A}{2}(1-x)\right)\right]}{(P+A)^2 \exp\left(\frac{A}{2}\right) - (P-A)^2 \exp\left(-\frac{A}{2}\right)}$$

where

$$A = \sqrt{P^2 + 4D}$$

```
>>> from fipy import CellVariable, FaceVariable, Grid1D, DiffusionTerm, \
...     PowerLawConvectionTerm, ImplicitSourceTerm, Viewer
>>> from fipy.tools import numerix
>>> nx = 100
>>> dx = 1.0 / nx
```

```
>>> mesh = Grid1D(nx=nx, dx=dx)
>>> C = CellVariable(mesh=mesh)
```

```
>>> D = 2.0
>>> P = 3.0
```

```
>>> C.faceGrad.constrain([0], mesh.facesRight)
```

We note that the Robin condition exactly defines the flux on the left, so we introduce a corresponding divergence source to the equation.

Note: Zeroing out the coefficients of the equation at this boundary is probably not necessary due to the default no-flux boundary conditions of cell-centered finite volume, but it's a safe precaution.

```
>>> convectionCoeff = FaceVariable(mesh=mesh, value=[P])
>>> convectionCoeff[... , mesh.facesLeft.value] = 0.
>>> diffusionCoeff = FaceVariable(mesh=mesh, value=1.)
>>> diffusionCoeff[... , mesh.facesLeft.value] = 0.
```

```
>>> eq = (PowerLawConvectionTerm(coeff=convectionCoeff)
...      == DiffusionTerm(coeff=diffusionCoeff) - ImplicitSourceTerm(coeff=D)
...      - (P * mesh.facesLeft).divergence)
```

```
>>> A = numerix.sqrt(P**2 + 4 * D)
```

```
>>> x = mesh.cellCenters[0]
>>> CAnalytical = CellVariable(mesh=mesh)
>>> CAnalytical.setValue(2 * P * numerix.exp(P * x / 2)
...                      * ((P + A) * numerix.exp(A / 2 * (1 - x))
...                          - (P - A) * numerix.exp(-A / 2 * (1 - x)))
...                      / ((P + A)**2 * numerix.exp(A / 2)
...                          - (P - A)**2 * numerix.exp(-A / 2)))
```

```
>>> if __name__ == '__main__':
...     C.name = '$C$'
...     CAnalytical.name = '$C_{analytical}$'
...     viewer = Viewer(vars=(C, CAnalytical))
```

```
>>> if __name__ == '__main__':
...     restol = 1e-5
...     anstol = 1e-3
...     else:
...         restol = 0.5
...         anstol = 0.15
```

```
>>> res = 1e+10
```

```
>>> while res > restol:
...     res = eq.sweep(var=C)
...     if __name__ == '__main__':
...         viewer.plot()
```

```
>>> print(C.allclose(CAnalytical, rtol=anstol, atol=anstol))
True
```

20.4 examples.convection.source

Solve a convection problem with a source.

This example solves the equation

$$\frac{\partial \phi}{\partial x} + \alpha \phi = 0$$

with $\phi(0) = 1$ at $x = 0$. The boundary condition at $x = L$ is an outflow boundary condition requiring the use of an artificial constraint to be set on the right hand side faces. Exterior faces without constraints are considered to have zero outflow. An `ImplicitSourceTerm` object will be used to represent this term. The derivative of ϕ can be represented by a `ConvectionTerm` with a constant unitary velocity field from left to right. The following is an example code that includes a test against the analytical result.

```
>>> from fipy import CellVariable, Grid1D, DiffusionTerm, PowerLawConvectionTerm, \
↳ ImplicitSourceTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 10.
>>> nx = 5000
>>> dx = L / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
>>> phi0 = 1.0
>>> alpha = 1.0
>>> phi = CellVariable(name=r"$\phi$", mesh=mesh, value=phi0)
>>> solution = CellVariable(name=r"solution", mesh=mesh, value=phi0 * numerix.exp(-
↳ alpha * mesh.cellCenters[0]))
```

```
>>> from fipy import input
>>> if __name__ == "__main__":
...     viewer = Viewer(vars=(phi, solution))
...     viewer.plot()
...     input("press key to continue")
```

```
>>> phi.constrain(phi0, mesh.facesLeft)
>>> ## fake outflow condition
>>> phi.faceGrad.constrain([0], mesh.facesRight)
```

```
>>> eq = PowerLawConvectionTerm((1,)) + ImplicitSourceTerm(alpha)
>>> eq.solve(phi)
>>> print(numerix.allclose(phi, phi0 * numerix.exp(-alpha * mesh.cellCenters[0]), \
↳ atol=1e-3))
True
```

```
>>> from fipy import input
>>> if __name__ == "__main__":
...     viewer = Viewer(vars=(phi, solution))
...     viewer.plot()
...     input("finished")
```


Chapter 21

Phase Field Examples

<code>examples.phase.simple</code>	Solve a phase-field (Allen-Cahn) problem in one-dimension.
<code>examples.phase.binaryCoupled</code>	Simultaneously solve a phase-field evolution and solute diffusion problem in one-dimension.
<code>examples.phase.quaternary</code>	Solve a phase-field evolution and diffusion of four species in one-dimension.
<code>examples.phase.anisotropy</code>	Solve a dendritic solidification problem.
<code>examples.phase.impingement.mesh40x1</code>	Solve for the impingement of two grains in one dimension.
<code>examples.phase.impingement.mesh20x20</code>	Solve for the impingement of four grains in two dimensions.
<code>examples.phase.polyxtal</code>	Solve the dendritic growth of nuclei and subsequent grain impingement.
<code>examples.phase.polyxtalCoupled</code>	Simultaneously solve the dendritic growth of nuclei and subsequent grain impingement.

21.1 examples.phase.simple

Solve a phase-field (Allen-Cahn) problem in one-dimension.

To run this example from the base FiPy directory, type `python examples/phase/simple/input.py` at the command line. A viewer object should appear and, after being prompted to step through the different examples, the word `finished` in the terminal.

This example takes the user through assembling a simple problem with FiPy. It describes a steady 1D phase field problem with no-flux boundary conditions such that,

$$\frac{1}{M_\phi} \frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \quad (21.1)$$

For solidification problems, the Helmholtz free energy is frequently given by

$$f(\phi, T) = \frac{W}{2} g(\phi) + L_v \frac{T - T_M}{T_M} p(\phi)$$

where W is the double-well barrier height between phases, L_v is the latent heat, T is the temperature, and T_M is the melting point.

One possible choice for the double-well function is

$$g(\phi) = \phi^2(1 - \phi)^2$$

and for the interpolation function is

$$p(\phi) = \phi^3(6\phi^2 - 15\phi + 10).$$

We create a 1D solution mesh

```
>>> from fipy import CellVariable, Variable, Grid1D, DiffusionTerm, TransientTerm, \
↳ ImplicitSourceTerm, DummySolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 1.
>>> nx = 400
>>> dx = L / nx
```

```
>>> mesh = Grid1D(dx = dx, nx = nx)
```

We create the phase field variable

```
>>> phase = CellVariable(name = "phase",
...                       mesh = mesh)
```

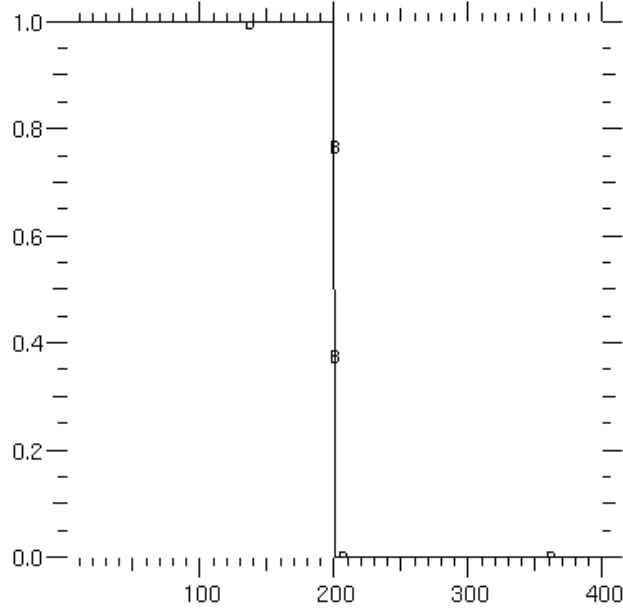
and set a step-function initial condition

$$\phi = \begin{cases} 1 & \text{for } x \leq L/2 \\ 0 & \text{for } x > L/2 \end{cases} \quad \text{at } t = 0$$

```
>>> x = mesh.cellCenters[0]
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

If we are running interactively, we'll want a viewer to see the results

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer = Viewer(vars = (phase,))
...     viewer.plot()
...     input("Initial condition. Press <return> to proceed...")
```



We choose the parameter values,

```
>>> kappa = 0.0025
>>> W = 1.
>>> Lv = 1.
>>> Tm = 1.
>>> T = Tm
>>> enthalpy = Lv * (T - Tm) / Tm
```

We build the equation by assembling the appropriate terms. Since, with $T = T_M$ we are interested in a steady-state solution, we omit the transient term $(1/M_\phi) \frac{\partial \phi}{\partial t}$.

The analytical solution for this steady-state phase field problem, in an infinite domain, is

$$\phi = \frac{1}{2} \left[1 - \tanh \frac{x - L/2}{2\sqrt{\kappa/W}} \right] \quad (21.2)$$

or

```
>>> x = mesh.cellCenters[0]
>>> analyticalArray = 0.5*(1 - numerix.tanh((x - L/2)/(2*numerix.sqrt(kappa/W))))
```

We treat the diffusion term $\kappa_\phi \nabla^2 \phi$ implicitly,

Note: “Diffusion” in *FiPy* is not limited to the movement of atoms, but rather refers to the spontaneous spreading of any quantity (e.g., solute, temperature, or in this case “phase”) by flow “down” the gradient of that quantity.

The source term is

$$\begin{aligned} S &= -\frac{\partial f}{\partial \phi} = -\frac{W}{2} g'(\phi) - L \frac{T - T_M}{T_M} p'(\phi) \\ &= -\left[W\phi(1 - \phi)(1 - 2\phi) + L \frac{T - T_M}{T_M} 30\phi^2(1 - \phi)^2 \right] \\ &= m_\phi \phi(1 - \phi) \end{aligned}$$

where $m_\phi \equiv -[W(1 - 2\phi) + 30\phi(1 - \phi)L \frac{T - T_M}{T_M}]$.

The simplest approach is to add this source explicitly

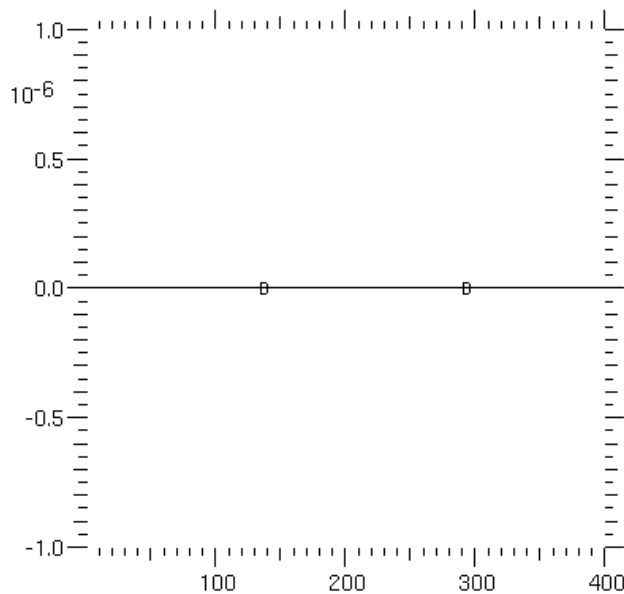
```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> S0 = mPhi * phase * (1 - phase)
>>> eq = S0 + DiffusionTerm(coeff=kappa)
```

After solving this equation

```
>>> eq.solve(var = phase, solver=DummySolver())
```

we obtain the surprising result that ϕ is zero everywhere.

```
>>> print(phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4))
0
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Fully explicit source. Press <return> to proceed...")
```



On inspection, we can see that this occurs because, for our step-function initial condition, $m_\phi = 0$ everywhere, hence we are actually only solving the simple implicit diffusion equation $\kappa_\phi \nabla^2 \phi = 0$, which has exactly the uninteresting solution we obtained.

The resolution to this problem is to apply relaxation to obtain the desired answer, i.e., the solution is allowed to relax in time from the initial condition to the desired equilibrium solution. To do so, we reintroduce the transient term from Equation (21.1)

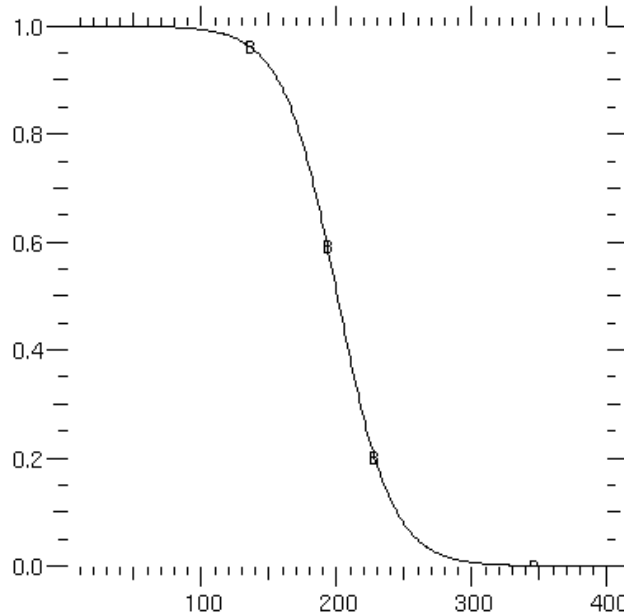
```
>>> eq = TransientTerm() == DiffusionTerm(coeff=kappa) + S0
```

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

```
>>> from builtins import range
>>> for i in range(13):
...     eq.solve(var = phase, dt=1.)
...     if __name__ == '__main__':
...         viewer.plot()
```

After 13 time steps, the solution has converged to the analytical solution

```
>>> print(phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4))
1
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Relaxation, explicit. Press <return> to proceed...")
```



Note: The solution is only found accurate to $\approx 4.3 \times 10^{-5}$ because the infinite-domain analytical solution (21.2) is not an exact representation for the solution in a finite domain of length L .

Setting fixed-value boundary conditions of 1 and 0 would still require the relaxation method with the fully explicit source.

Solution performance can be improved if we exploit the dependence of the source on ϕ . By doing so, we can make the source semi-implicit, improving the rate of convergence over the fully explicit approach. The source can only be semi-implicit because we employ sparse linear algebra routines to solve the PDEs, i.e., there is no fully implicit way to represent a term like ϕ^4 in the linear set of equations $M\vec{\phi} - \vec{b} = 0$.

By linearizing a source as $S = S_0 - S_1\phi$, we make it more implicit by adding the coefficient S_1 to the matrix diagonal. For numerical stability, this linear coefficient must never be negative.

There are an infinite number of choices for this linearization, but many do not converge very well. One choice is that used by Ryo Kobayashi:

```
>>> S0 = mPhi * phase * (mPhi > 0)
>>> S1 = mPhi * ((mPhi < 0) - phase)
>>> eq = DiffusionTerm(coeff=kappa) + S0 \
...     + ImplicitSourceTerm(coeff = S1)
```

Note: Because `mPhi` is a variable field, the quantities `(mPhi > 0)` and `(mPhi < 0)` evaluate to variable *fields* of *True* and *False*, instead of single Boolean values.

This expression converges to the same value given by the explicit relaxation approach, but in only 8 sweeps (note that

because there is no transient term, these sweeps are not time steps, but rather repeated iterations at the same time step to reach a converged solution).

Note: We use `solve()` instead of `sweep()` because we don't care about the residual. Either function would work, but `solve()` is a bit faster.

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

```
>>> from builtins import range
>>> for i in range(8):
...     eq.solve(var = phase)
>>> print(phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4))
1
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Kobayashi, semi-implicit. Press <return> to proceed...")
```

In general, the best convergence is obtained when the linearization gives a good representation of the relationship between the source and the dependent variable. The best practical advice is to perform a Taylor expansion of the source about the previous value of the dependent variable such that $S = S_{\text{old}} + \left. \frac{\partial S}{\partial \phi} \right|_{\text{old}} (\phi - \phi_{\text{old}}) = (S - \left. \frac{\partial S}{\partial \phi} \phi)_{\text{old}} + \left. \frac{\partial S}{\partial \phi} \right|_{\text{old}} \phi$. Now, if our source term is represented by $S = S_0 + S_1 \phi$, then $S_1 = \left. \frac{\partial S}{\partial \phi} \right|_{\text{old}}$ and $S_0 = (S - \left. \frac{\partial S}{\partial \phi} \phi)_{\text{old}} = S_{\text{old}} - S_1 \phi_{\text{old}}$. In this way, the linearized source will be tangent to the curve of the actual source as a function of the dependent variable.

For our source, $S = m_\phi \phi(1 - \phi)$,

$$\frac{\partial S}{\partial \phi} = \frac{\partial m_\phi}{\partial \phi} \phi(1 - \phi) + m_\phi(1 - 2\phi)$$

and

$$\frac{\partial m_\phi}{\partial \phi} = 2W - 30(1 - 2\phi)L \frac{T - T_M}{T_M},$$

or

```
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
>>> eq = DiffusionTerm(coeff=kappa) + S0 \
...     + ImplicitSourceTerm(coeff = S1)
```

Using this scheme, where the coefficient of the implicit source term is tangent to the source, we reach convergence in only 5 sweeps

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

```
>>> from builtins import range
>>> for i in range(5):
...     eq.solve(var = phase)
>>> print(phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4))
1
>>> from fipy import input
```

(continues on next page)

(continued from previous page)

```
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Tangent, semi-implicit. Press <return> to proceed...")
```

Although, for this simple problem, there is no appreciable difference in run-time between the fully explicit source and the optimized semi-implicit source, the benefit of 60% fewer sweeps should be obvious for larger systems and longer iterations.

This example has focused on just the region of the phase field interface in equilibrium. Problems of interest, though, usually involve the dynamics of one phase transforming to another. To that end, let us recast the problem using physical parameters and dimensions. We'll need a new mesh

```
>>> nx = 400
>>> dx = 5e-6 # cm
>>> L = nx * dx
```

```
>>> mesh = Grid1D(dx = dx, nx = nx)
```

and thus must redeclare ϕ on the new mesh

```
>>> phase = CellVariable(name="phase",
...                       mesh=mesh,
...                       hasOld=1)
>>> x = mesh.cellCenters[0]
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

We choose the parameter values appropriate for nickel, given in [28]

```
>>> Lv = 2350 # J / cm**3
>>> Tm = 1728. # K
>>> T = Variable(value=Tm)
>>> enthalpy = Lv * (T - Tm) / Tm # J / cm**3
```

The parameters of the phase field model can be related to the surface energy σ and the interfacial thickness δ by

$$\begin{aligned}\kappa &= 6\sigma\delta \\ W &= \frac{6\sigma}{\delta} \\ M_\phi &= \frac{T_m\beta}{6L\delta}.\end{aligned}$$

We take $\delta \approx \Delta x$.

```
>>> delta = 1.5 * dx
>>> sigma = 3.7e-5 # J / cm**2
>>> beta = 0.33 # cm / (K s)
>>> kappa = 6 * sigma * delta # J / cm
>>> W = 6 * sigma / delta # J / cm**3
>>> Mphi = Tm * beta / (6. * Lv * delta) # cm**3 / (J s)
```

```
>>> if __name__ == '__main__':
...     displacement = L * 0.1
... else:
...     displacement = L * 0.025
```

```
>>> analyticalArray = CellVariable(name="tanh", mesh=mesh,
...                               value=0.5 * (1 - numerix.tanh((x - (L / 2. +
...displacement))
...                               / (2 * delta))))
```

and make a new viewer

```
>>> if __name__ == '__main__':
...     viewer2 = Viewer(vars = (phase, analyticalArray))
...     viewer2.plot()
```

Now we can redefine the transient phase field equation, using the optimal form of the source term shown above

```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
>>> eq = TransientTerm(coeff=1/Mphi) == DiffusionTerm(coeff=kappa) \
...     + S0 + ImplicitSourceTerm(coeff = S1)
```

In order to separate the effect of forming the phase field interface from the kinetics of moving it, we first equilibrate at the melting point. We now use the `sweep()` method instead of `solve()` because we require the residual.

```
>>> timeStep = 1e-6
>>> from builtins import range
>>> for i in range(10):
...     phase.updateOld()
...     res = 1e+10
...     while res > 1e-5:
...         res = eq.sweep(var=phase, dt=timeStep)
>>> if __name__ == '__main__':
...     viewer2.plot()
```

and then quench by 1 K

```
>>> T.setValue(T() - 1)
```

In order to have a stable numerical solution, the interface must not move more than one grid point per time step, we thus set the timestep according to the grid spacing Δx , the linear kinetic coefficient β , and the undercooling $|T_m - T|$. Again we use the `sweep()` method as a replacement for `solve()`.

```
>>> velocity = beta * abs(Tm - T()) # cm / s
>>> timeStep = .1 * dx / velocity # s
>>> elapsed = 0
>>> while elapsed < displacement / velocity:
...     phase.updateOld()
...     res = 1e+10
...     while res > 1e-5:
...         res = eq.sweep(var=phase, dt=timeStep)
...     elapsed += timeStep
...     if __name__ == '__main__':
...         viewer2.plot()
```

A hyperbolic tangent is not an exact steady-state solution given the quintic polynomial we chose for the p function, but it gives a reasonable approximation.

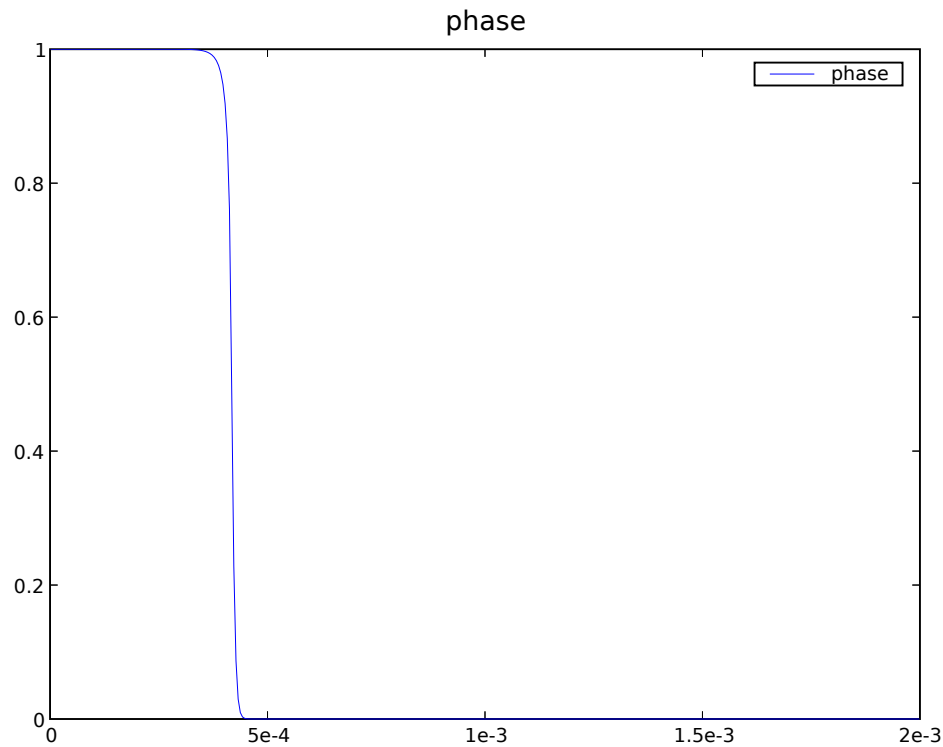

```
>>> print(phase.allclose(analyticalArray, rtol = 5, atol = 2e-3))
1
```

If we had made another common choice of $p(\phi) = \phi^2(3 - 2\phi)$, we would have found much better agreement, as that case does give an exact tanh solution in steady state. If SciPy is available, another way to compare against the expected result is to do a least-squared fit to determine the interface velocity and thickness

```
>>> try:
...     def tanhResiduals(p, y, x, t):
...         V, d = p
...         return y - 0.5 * (1 - numerix.tanh((x - V * t - L / 2.) / (2*d)))
...     from scipy.optimize import leastsq
...     x = mesh.cellCenters[0]
...     (V_fit, d_fit), msg = leastsq(tanhResiduals, [L/2., delta],
...                                   args=(phase.globalValue, x.globalValue,
...                                         elapsed))
... except ImportError:
...     V_fit = d_fit = 0
...     print("The SciPy library is unavailable to fit the interface \
... thickness and velocity")
```

```
>>> print(abs(1 - V_fit / velocity) < 4.1e-2)
True
>>> print(abs(1 - d_fit / delta) < 2e-2)
True
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Dimensional, semi-implicit. Press <return> to proceed...")
```



21.2 examples.phase.binaryCoupled

Simultaneously solve a phase-field evolution and solute diffusion problem in one-dimension.

It is straightforward to extend a phase field model to include binary alloys. As in `examples.phase.simple`, we will examine a 1D problem

```
>>> from fipy import CellVariable, Variable, Grid1D, TransientTerm, DiffusionTerm, \
↳ ImplicitSourceTerm, LinearLUSolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> nx = 400
>>> dx = 5e-6 # cm
>>> L = nx * dx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

The Helmholtz free energy functional can be written as the integral [3] [5] [27]

$$\mathcal{F}(\phi, C, T) = \int_{\mathcal{V}} \left\{ f(\phi, C, T) + \frac{\kappa_{\phi}}{2} |\nabla \phi|^2 + \frac{\kappa_C}{2} |\nabla C|^2 \right\} dV$$

over the volume \mathcal{V} as a function of phase ϕ ¹

¹ We will find that we need to “sweep” this non-linear problem (see *e.g.* the composition-dependent diffusivity example in `examples.diffusion.mesh1D`), so we declare ϕ and C to retain an “old” value.

```
>>> phase = CellVariable(name="phase", mesh=mesh, hasOld=1)
```

composition C

```
>>> C = CellVariable(name="composition", mesh=mesh, hasOld=1)
```

and temperature T^2

```
>>> T = Variable(name="temperature")
```

Frequently, the gradient energy term in concentration is ignored and we can derive governing equations

$$\frac{\partial \phi}{\partial t} = M_\phi \left(\kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \right) \quad (21.3)$$

for phase and

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(M_C \nabla \frac{\partial f}{\partial C} \right) \quad (21.4)$$

for solute.

The free energy density $f(\phi, C, T)$ can be constructed in many different ways. One approach is to construct free energy densities for each of the pure components, as functions of phase, *e.g.*

$$f_A(\phi, T) = p(\phi) f_A^S(T) + (1 - p(\phi)) f_A^L(T) + \frac{W_A}{2} g(\phi)$$

where $f_A^L(T)$, $f_B^L(T)$, $f_A^S(T)$, and $f_B^S(T)$ are the free energy densities of the pure components. There are a variety of choices for the interpolation function $p(\phi)$ and the barrier function $g(\phi)$,

such as those shown in [examples.phase.simple](#)

```
>>> def p(phi):
...     return phi**3 * (6 * phi**2 - 15 * phi + 10)
```

```
>>> def g(phi):
...     return (phi * (1 - phi))**2
```

The desired thermodynamic model can then be applied to obtain $f(\phi, C, T)$, such as for a regular solution,

$$f(\phi, C, T) = (1 - C) f_A(\phi, T) + C f_B(\phi, T) + RT [(1 - C) \ln(1 - C) + C \ln C] + C(1 - C) [\Omega_S p(\phi) + \Omega_L (1 - p(\phi))]$$

where

```
>>> R = 8.314 # J / (mol K)
```

is the gas constant and Ω_S and Ω_L are the regular solution interaction parameters for solid and liquid.

Another approach is useful when the free energy densities $f^L(C, T)$ and $f^S(C, T)$ of the alloy in the solid and liquid phases are known. This might be the case when the two different phases have different thermodynamic models or when one or both is obtained from a Calphad code. In this case, we can construct

$$f(\phi, C, T) = p(\phi) f^S(C, T) + (1 - p(\phi)) f^L(C, T) + \left[(1 - C) \frac{W_A}{2} + C \frac{W_B}{2} \right] g(\phi).$$

² we are going to want to examine different temperatures in this example, so we declare T as a *Variable*

When the thermodynamic models are the same in both phases, both approaches should yield the same result.

We choose the first approach and make the simplifying assumptions of an ideal solution and that

$$f_A^L(T) = 0$$

$$f_A^S(T) - f_A^L(T) = \frac{L_A (T - T_M^A)}{T_M^A}$$

and likewise for component B .

```
>>> LA = 2350. # J / cm**3
>>> LB = 1728. # J / cm**3
>>> TmA = 1728. # K
>>> TmB = 1358. # K
```

```
>>> enthalpyA = LA * (T - TmA) / TmA
>>> enthalpyB = LB * (T - TmB) / TmB
```

This relates the difference between the free energy densities of the pure solid and pure liquid phases to the latent heat L_A and the pure component melting point T_M^A , such that

$$f_A(\phi, T) = \frac{L_A (T - T_M^A)}{T_M^A} p(\phi) + \frac{W_A}{2} g(\phi).$$

With these assumptions

$$\frac{\partial f}{\partial \phi} = (1 - C) \frac{\partial f_A}{\partial \phi} + C \frac{\partial f_B}{\partial \phi}$$

$$= \left\{ (1 - C) \frac{L_A (T - T_M^A)}{T_M^A} + C \frac{L_B (T - T_M^B)}{T_M^B} \right\} p'(\phi) + \left\{ (1 - C) \frac{W_A}{2} + C \frac{W_B}{2} \right\} g'(\phi)$$

and

$$\frac{\partial f}{\partial C} = \left[f_B(\phi, T) + \frac{RT}{V_m} \ln C \right] - \left[f_A(\phi, T) + \frac{RT}{V_m} \ln(1 - C) \right]$$

$$= [\mu_B(\phi, C, T) - \mu_A(\phi, C, T)] / V_m$$

where μ_A and μ_B are the classical chemical potentials for the binary species. $p'(\phi)$ and $g'(\phi)$ are the partial derivatives of p and g with respect to ϕ

```
>>> def pPrime(phi):
...     return 30. * g(phi)
```

```
>>> def gPrime(phi):
...     return 2. * phi * (1 - phi) * (1 - 2 * phi)
```

V_m is the molar volume, which we take to be independent of concentration and phase

```
>>> Vm = 7.42 # cm**3 / mol
```

On comparison with `examples.phase.simple`, we can see that the present form of the phase field equation is identical to the one found earlier, with the source now composed of the concentration-weighted average of the source for either pure component. We let the pure component barriers equal the previous value

```
>>> deltaA = deltaB = 1.5 * dx
>>> sigmaA = 3.7e-5 # J / cm**2
>>> sigmaB = 2.9e-5 # J / cm**2
>>> betaA = 0.33 # cm / (K s)
>>> betaB = 0.39 # cm / (K s)
>>> kappaA = 6 * sigmaA * deltaA # J / cm
>>> kappaB = 6 * sigmaB * deltaB # J / cm
>>> WA = 6 * sigmaA / deltaA # J / cm**3
>>> WB = 6 * sigmaB / deltaB # J / cm**3
```

and define the averages

```
>>> W = (1 - C) * WA / 2. + C * WB / 2.
>>> enthalpy = (1 - C) * enthalpyA + C * enthalpyB
```

We can now linearize the source exactly as before

```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
```

Using the same gradient energy coefficient and phase field mobility

```
>>> kappa = (1 - C) * kappaA + C * kappaB
>>> Mphi = TmA * betaA / (6 * LA * deltaA)
```

we define the phase field equation

```
>>> phaseEq = (TransientTerm(1/Mphi, var=phase) == DiffusionTerm(coeff=kappa,
↳var=phase)
...          + S0 + ImplicitSourceTerm(coeff=S1, var=phase))
```

When coding explicitly, it is typical to simply write a function to evaluate the chemical potentials μ_A and μ_B and then perform the finite differences necessary to calculate their gradient and divergence, e.g.,:

```
def deltaChemPot(phase, C, T):
    return ((Vm * (enthalpyB * p(phase) + WA * g(phase)) + R * T * log(1 - C)) -
            (Vm * (enthalpyA * p(phase) + WA * g(phase)) + R * T * log(C)))

for j in range(faces):
    flux[j] = ((Mc[j+.5] + Mc[j-.5]) / 2) \
        * (deltaChemPot(phase[j+.5], C[j+.5], T) \
            - deltaChemPot(phase[j-.5], C[j-.5], T)) / dx

for j in range(cells):
    diffusion = (flux[j+.5] - flux[j-.5]) / dx
```

where we neglect the details of the outer boundaries ($j = 0$ and $j = N$) or exactly how to translate $j+.5$ or $j-.5$ into an array index, much less the complexities of higher dimensions. FiPy can handle all of these issues automatically, so we could just write:

```
chemPotA = Vm * (enthalpyA * p(phase) + WA * g(phase)) + R * T * log(C)
chemPotB = Vm * (enthalpyB * p(phase) + WB * g(phase)) + R * T * log(1-C)
flux = Mc * (chemPotB - chemPotA).faceGrad
eq = TransientTerm() == flux.divergence
```

Although the second syntax would essentially work as written, such an explicit implementation would be very slow. In order to take advantage of *FiPy*'s implicit solvers, it is necessary to reduce Eq. (21.4) to the canonical form of Eq. (11.2), hence we must expand Eq. (21.5) as

$$\frac{\partial f}{\partial C} = \left[\frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] p(\phi) + \frac{RT}{V_m} [\ln C - \ln(1 - C)] + \frac{W_B - W_A}{2} g(\phi)$$

In either bulk phase, $\nabla p(\phi) = \nabla g(\phi) = 0$, so we can then reduce Eq. (21.4) to

$$\begin{aligned} \frac{\partial C}{\partial t} &= \nabla \cdot \left(M_C \nabla \left\{ \frac{RT}{V_m} [\ln C - \ln(1 - C)] \right\} \right) \\ &= \nabla \cdot \left[\frac{M_C RT}{C(1 - C)V_m} \nabla C \right] \end{aligned}$$

and, by comparison with Fick's second law

$$\frac{\partial C}{\partial t} = \nabla \cdot [D \nabla C],$$

we can associate the mobility M_C with the intrinsic diffusivity D_C by $M_C \equiv D_C C(1 - C)V_m/RT$ and write Eq. (21.4) as

$$\begin{aligned} \frac{\partial C}{\partial t} &= \nabla \cdot (D_C \nabla C) \\ &\quad + \nabla \cdot \left(\frac{D_C C(1 - C)V_m}{RT} \left\{ \left[\frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] \nabla p(\phi) + \frac{W_B - W_A}{2} \nabla g(\phi) \right\} \right) \\ &= \nabla \cdot (D_C \nabla C) \\ &\quad + \nabla \cdot \left(\frac{D_C C(1 - C)V_m}{RT} \left\{ \left[\frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] p'(\phi) + \frac{W_B - W_A}{2} g'(\phi) \right\} \nabla \phi \right). \end{aligned}$$

The first term is clearly a *DiffusionTerm* in C . The second is a *DiffusionTerm* in ϕ with a diffusion coefficient

$$D_\phi(C, \phi) = \frac{D_C C(1 - C)V_m}{RT} \left\{ \left[\frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] p'(\phi) + \frac{W_B - W_A}{2} g'(\phi) \right\},$$

such that

$$\frac{\partial C}{\partial t} = \nabla \cdot (D_C \nabla C) + \nabla \cdot (D_\phi \nabla \phi)$$

or

```
>>> D1 = Variable(value=1e-5) # cm**2 / s
>>> Ds = Variable(value=1e-9) # cm**2 / s
>>> Dc = (Ds - D1) * phase.arithmeticFaceValue + D1
```

```
>>> Dphi = ((Dc * C.harmonicFaceValue * (1 - C.harmonicFaceValue) * Vm / (R * T))
...         * ((enthalpyB - enthalpyA) * pPrime(phase.arithmeticFaceValue)
...         + 0.5 * (WB - WA) * gPrime(phase.arithmeticFaceValue)))
```

```
>>> diffusionEq = (TransientTerm(var=C)
...               == DiffusionTerm(coeff=Dc, var=C)
...               + DiffusionTerm(coeff=Dphi, var=phase))
```

```
>>> eq = phaseEq & diffusionEq
```

We initialize the phase field to a step function in the middle of the domain

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=mesh.cellCenters[0] > L/2.)
```

and start with a uniform composition field $C = 1/2$

```
>>> C.setValue(0.5)
```

In equilibrium, $\mu_A(0, C_L, T) = \mu_A(1, C_S, T)$ and $\mu_B(0, C_L, T) = \mu_B(1, C_S, T)$ and, for ideal solutions, we can deduce the liquidus and solidus compositions as

$$C_L = \frac{1 - \exp\left(-\frac{L_A(T - T_M^A)}{T_M^A} \frac{V_m}{RT}\right)}{\exp\left(-\frac{L_B(T - T_M^B)}{T_M^B} \frac{V_m}{RT}\right) - \exp\left(-\frac{L_A(T - T_M^A)}{T_M^A} \frac{V_m}{RT}\right)}$$

$$C_S = \exp\left(-\frac{L_B(T - T_M^B)}{T_M^B} \frac{V_m}{RT}\right) C_L$$

```
>>> Cl = ((1. - numerix.exp(-enthalpyA * Vm / (R * T)))
...       / (numerix.exp(-enthalpyB * Vm / (R * T)) - numerix.exp(-enthalpyA * Vm /
...       (R * T))))
>>> Cs = numerix.exp(-enthalpyB * Vm / (R * T)) * Cl
```

The phase fraction is predicted by the lever rule

```
>>> Cavg = C.cellVolumeAverage
>>> fraction = (Cl - Cavg) / (Cl - Cs)
```

For the special case of $\text{fraction} = \text{Cavg} = 0.5$, a little bit of algebra reveals that the temperature that leaves the phase fraction unchanged is given by

```
>>> T.setValue((LA + LB) * TmA * TmB / (LA * TmB + LB * TmA))
```

In this simple, binary, ideal solution case, we can derive explicit expressions for the solidus and liquidus compositions. In general, this may not be possible or practical. In that event, the root-finding facilities in SciPy can be used.

We'll need a function to return the two conditions for equilibrium

$$0 = \mu_A(1, C_S, T) - \mu_A(0, C_L, T) = \frac{L_A(T - T_M^A)}{T_M^A} V_m + RT \ln(1 - C_S) - RT \ln(1 - C_L)$$

$$0 = \mu_B(1, C_S, T) - \mu_B(0, C_L, T) = \frac{L_B(T - T_M^B)}{T_M^B} V_m + RT \ln C_S - RT \ln C_L$$

```
>>> def equilibrium(C):
...     return [numerix.array(enthalpyA * Vm
...                             + R * T * numerix.log(1 - C[0])
...                             - R * T * numerix.log(1 - C[1])),
...             numerix.array(enthalpyB * Vm
...                             + R * T * numerix.log(C[0])
...                             - R * T * numerix.log(C[1]))]
```

and we'll have much better luck if we also supply the Jacobian

$$\begin{bmatrix} \frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_S} & \frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_L} \\ \frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_S} & \frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_L} \end{bmatrix} = RT \begin{bmatrix} -\frac{1}{1-C_S} & \frac{1}{1-C_L} \\ \frac{1}{C_S} & -\frac{1}{C_L} \end{bmatrix}$$

```
>>> def equilibriumJacobian(C):
...     return R * T * numerix.array([[ -1. / (1 - C[0]), 1. / (1 - C[1])],
...                                   [ 1. / C[0], -1. / C[1]])
```

```
>>> try:
...     from scipy.optimize import fsolve
...     CsRoot, ClRoot = fsolve(func=equilibrium, x0=[0.5, 0.5],
...                             fprime=equilibriumJacobian)
... except ImportError:
...     ClRoot = CsRoot = 0
...     print("The SciPy library is not available to calculate the solidus and ..._
↳ liquidus concentrations")
```

```
>>> print(Cl.allclose(ClRoot))
1
>>> print(Cs.allclose(CsRoot))
1
```

We plot the result against the sharp interface solution

```
>>> sharp = CellVariable(name="sharp", mesh=mesh)
>>> x = mesh.cellCenters[0]
>>> sharp.setValue(Cs, where=x < L * fraction)
>>> sharp.setValue(Cl, where=x >= L * fraction)
```

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(phase, C, sharp),
...                       datamin=0., datamax=1.)
...     viewer.plot()
```

Because the phase field interface will not move, and because we've seen in earlier examples that the diffusion problem is unconditionally stable, we need take only one very large timestep to reach equilibrium

```
>>> dt = 1.e5
```

Because the phase field equation is coupled to the composition through enthalpy and W and the diffusion equation is coupled to the phase field through $\text{phaseTransformationVelocity}$, it is necessary sweep this non-linear problem to convergence. We use the “residual” of the equations (a measure of how well they think they have solved the given set of linear equations) as a test for how long to sweep.

We now use the “`sweep()`” method instead of “`solve()`” because we require the residual.

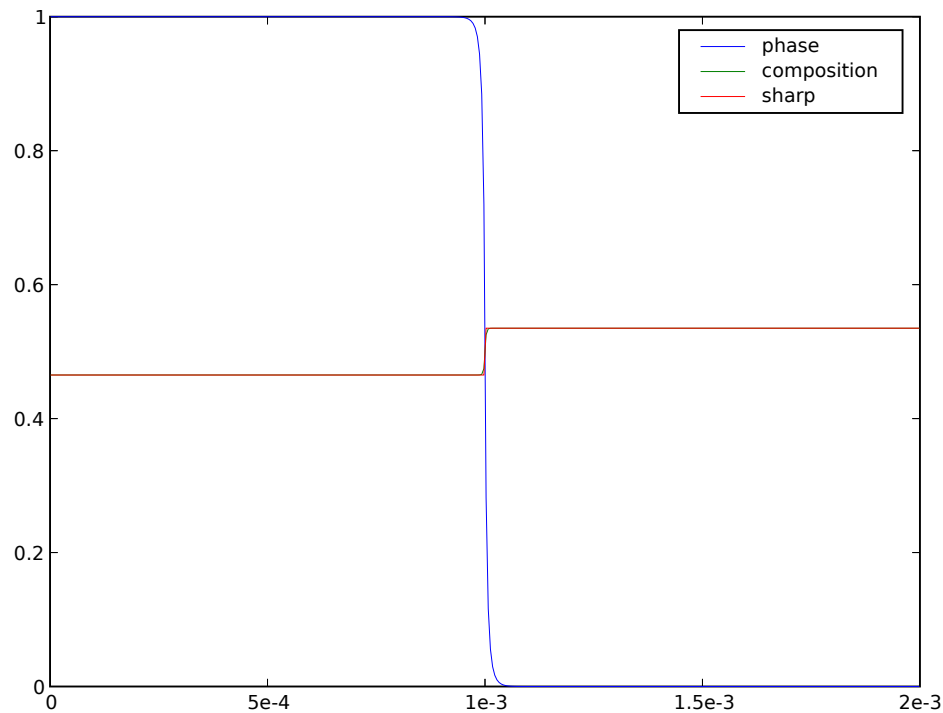
```
>>> solver = LinearLUSolver(tolerance=1e-10)
```

```
>>> phase.updateOld()
>>> C.updateOld()
>>> res = 1.
>>> initialRes = None
>>> sweep = 0
```



```
>>> while res > 1e-4 and sweep < 20:
...     res = eq.sweep(dt=dt, solver=solver)
...     if initialRes is None:
...         initialRes = res
...     res = res / initialRes
...     sweep += 1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Stationary phase field. Press <return> to proceed...")
```



We verify that the bulk phases have shifted to the predicted solidus and liquidus compositions

```
>>> X = mesh.faceCenters[0]
>>> print(Cs.allclose(C.faceValue[X.value==0], atol=1e-2))
True
>>> print(Cl.allclose(C.faceValue[X.value==L], atol=1e-2))
True
```

and that the phase fraction remains unchanged

```
>>> print(fraction.allclose(phase.cellVolumeAverage, atol=2e-4))
1
```

while conserving mass overall

```
>>> print(Cavg.allclose(0.5, atol=1e-8))
1
```

We now quench by ten degrees

```
>>> T.setValue(T() - 10.) # K
```

```
>>> sharp.setValue(Cs, where=x < L * fraction)
>>> sharp.setValue(Cl, where=x >= L * fraction)
```

Because this lower temperature will induce the phase interface to move (solidify), we will need to take much smaller timesteps (the time scales of diffusion and of phase transformation compete with each other).

The CFL limit requires that no interface should advect more than one grid spacing in a timestep. We can get a rough idea for the maximum timestep we can take by looking at the velocity of convection induced by phase transformation in Eq. (21.5) (even though there is no explicit convection in the coupled form used for this example, the principle remains the same). If we assume that the phase changes from 1 to 0 in a single grid spacing, that the diffusivity is Dl at the interface, and that the term due to the difference in barrier heights is negligible:

$$\begin{aligned}\vec{u}_\phi &= \frac{D_\phi}{C} \nabla \phi \\ &\approx \frac{Dl \frac{1}{2} V_m}{RT} \left[\frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] \frac{1}{\Delta x} \\ &\approx \frac{Dl \frac{1}{2} V_m}{RT} (L_B + L_A) \frac{T_M^A - T_M^B}{T_M^A + T_M^B} \frac{1}{\Delta x} \\ &\approx 0.28 \text{ cm/s}\end{aligned}$$

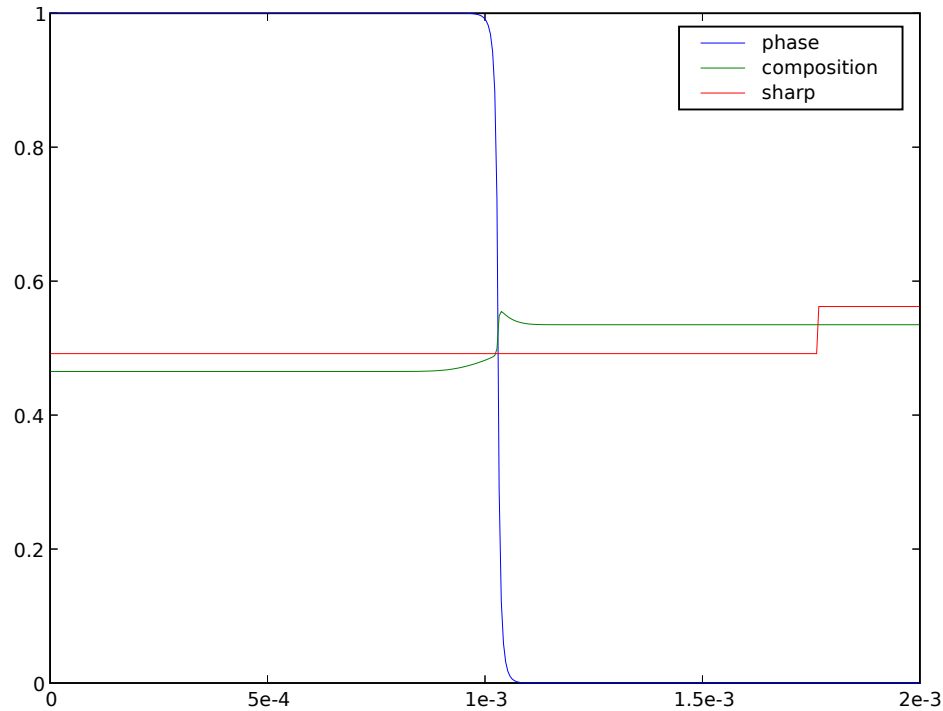
To get a CFL = $\vec{u}_\phi \Delta t / \Delta x < 1$, we need a time step of about 10^{-5} s.

```
>>> dt = 1.e-5
```

```
>>> if __name__ == '__main__':
...     timesteps = 100
... else:
...     timesteps = 10
```

```
>>> from builtins import range
>>> for i in range(timesteps):
...     phase.updateOld()
...     C.updateOld()
...     res = 1e+10
...     sweep = 0
...     while res > 1e-3 and sweep < 20:
...         res = eq.sweep(dt=dt, solver=solver)
...         sweep += 1
...     if __name__ == '__main__':
...         viewer.plot()
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Moving phase field. Press <return> to proceed...")
```



We see that the composition on either side of the interface approach the sharp-interface solidus and liquidus, but it will take a great many more timesteps to reach equilibrium. If we waited sufficiently long, we could again verify the final concentrations and phase fraction against the expected values.

21.3 examples.phase.quaternary

Solve a phase-field evolution and diffusion of four species in one-dimension.

The same procedure used to construct the two-component phase field diffusion problem in `examples.phase.binary` can be used to build up a system of multiple components. Once again, we'll focus on 1D.

```
>>> from fipy import CellVariable, Grid1D, TransientTerm, DiffusionTerm, \
↳ ImplicitSourceTerm, PowerLawConvectionTerm, DefaultAsymmetricSolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> nx = 400
>>> dx = 0.01
>>> L = nx * dx
>>> mesh = Grid1D(dx = dx, nx = nx)
```

We consider a free energy density $f(\phi, C_0, \dots, C_N, T)$ that is a function of phase ϕ

```
>>> phase = CellVariable(mesh=mesh, name='phase', value=1., hasOld=1)
```

interstitial components $C_0 \dots C_M$

```
>>> interstitials = [
...     CellVariable(mesh=mesh, name='C0', hasOld=1)
... ]
```

substitutional components $C_{M+1} \dots C_{N-1}$

```
>>> substitutionals = [
...     CellVariable(mesh=mesh, name='C1', hasOld=1),
...     CellVariable(mesh=mesh, name='C2', hasOld=1),
... ]
```

a “solvent” C_N that is constrained by the concentrations of the other substitutional species, such that $C_N = 1 - \sum_{j=M}^{N-1} C_j$,

```
>>> solvent = 1
>>> for Cj in substitutionals:
...     solvent -= Cj
>>> solvent.name = 'CN'
```

and temperature T

```
>>> T = 1000
```

The free energy density of such a system can be written as

$$f(\phi, C_0, \dots, C_N, T) = \sum_{j=0}^N C_j \left[\mu_j^\circ(\phi, T) + RT \ln \frac{C_j}{\rho} \right]$$

where

```
>>> R = 8.314 # J / (mol K)
```

is the gas constant. As in the binary case,

$$\mu_j^\circ(\phi, T) = p(\phi) \mu_j^{\circ S}(T) + (1 - p(\phi)) \mu_j^{\circ L}(T) + \frac{W_j}{2} g(\phi)$$

is constructed with the free energies of the pure components in each phase, given the “tilting” function

```
>>> def p(phi):
...     return phi**3 * (6 * phi**2 - 15 * phi + 10)
```

and the “double well” function

```
>>> def g(phi):
...     return (phi * (1 - phi))**2
```

We consider a very simplified model that has partial molar volumes $\bar{V}_0 = \dots = \bar{V}_M = 0$ for the “interstitials” and $\bar{V}_{M+1} = \dots = \bar{V}_N = 1$ for the “substitutionals”. This approximation has been used in a number of models where density effects are ignored, including the treatment of electrons in electrodeposition processes [29] [30]. Under these

constraints

$$\begin{aligned}\frac{\partial f}{\partial \phi} &= \sum_{j=0}^N C_j \frac{\partial f_j}{\partial \phi} \\ &= \sum_{j=0}^N C_j \left[\mu_j^{\circ SL}(T) p'(\phi) + \frac{W_j}{2} g'(\phi) \right] \\ \frac{\partial f}{\partial C_j} &= \left[\mu_j^{\circ}(\phi, T) + RT \ln \frac{C_j}{\rho} \right] \\ &= \mu_j(\phi, C_j, T) \quad \text{for } j = 0 \dots M\end{aligned}$$

and

$$\begin{aligned}\frac{\partial f}{\partial C_j} &= \left[\mu_j^{\circ}(\phi, T) + RT \ln \frac{C_j}{\rho} \right] - \left[\mu_N^{\circ}(\phi, T) + RT \ln \frac{C_N}{\rho} \right] \\ &= [\mu_j(\phi, C_j, T) - \mu_N(\phi, C_N, T)] \quad \text{for } j = M + 1 \dots N - 1\end{aligned}$$

where $\mu_j^{\circ SL}(T) \equiv \mu_j^{\circ S}(T) - \mu_j^{\circ L}(T)$ and where μ_j is the classical chemical potential of component j for the binary species and $\rho = 1 + \sum_{j=0}^M C_j$ is the total molar density.

```
>>> rho = 1.
>>> for Cj in interstitials:
...     rho += Cj
```

$p'(\phi)$ and $g'(\phi)$ are the partial derivatives of p and g with respect to ϕ

```
>>> def pPrime(phi):
...     return 30. * g(phi)
```

```
>>> def gPrime(phi):
...     return 2. * phi * (1 - phi) * (1 - 2 * phi)
```

We “cook” the standard potentials to give the desired solid and liquid concentrations, with a solid phase rich in interstitials and the solvent and a liquid phase rich in the two substitutional species.

```
>>> interstitials[0].S = 0.3
>>> interstitials[0].L = 0.4
>>> substitutionals[0].S = 0.4
>>> substitutionals[0].L = 0.3
>>> substitutionals[1].S = 0.2
>>> substitutionals[1].L = 0.1
>>> solvent.S = 1.
>>> solvent.L = 1.
>>> for Cj in substitutionals:
...     solvent.S -= Cj.S
...     solvent.L -= Cj.L
```

```
>>> rhoS = rhoL = 1.
>>> for Cj in interstitials:
...     rhoS += Cj.S
...     rhoL += Cj.L
```

```
>>> for Cj in interstitials + substitutionals + [solvent]:
...     Cj.standardPotential = R * T * (numerix.log(Cj.L/rhoL)
...     - numerix.log(Cj.S/rhoS))
```

```
>>> for Cj in interstitials:
...     Cj.diffusivity = 1.
...     Cj.barrier = 0.
```

```
>>> for Cj in substitutionals:
...     Cj.diffusivity = 1.
...     Cj.barrier = R * T
```

```
>>> solvent.barrier = R * T
```

We create the phase equation

$$\frac{1}{M_\phi} \frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \sum_{j=0}^N C_j \left[\mu_j^{\circ SL}(T) p'(\phi) + \frac{W_j}{2} g'(\phi) \right]$$

with a semi-implicit source just as in `examples.phase.simple` and `examples.phase.binary`

```
>>> enthalpy = 0.
>>> barrier = 0.
>>> for Cj in interstitials + substitutionals + [solvent]:
...     enthalpy += Cj * Cj.standardPotential
...     barrier += Cj * Cj.barrier
```

```
>>> mPhi = -((1 - 2 * phase) * barrier + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * barrier - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
```

```
>>> phase.mobility = 1.
>>> phase.gradientEnergy = 25
>>> phase.equation = TransientTerm(coeff=1/phase.mobility) \
...     == DiffusionTerm(coeff=phase.gradientEnergy) \
...     + S0 + ImplicitSourceTerm(coeff = S1)
```

We could construct the diffusion equations one-by-one, in the manner of `examples.phase.binary`, but it is better to take advantage of the full scripting power of the Python language, where we can easily loop over components or even make “factory” functions if we desire. For the interstitial diffusion equations, we arrange in canonical form as before:

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}} + \underbrace{D_j \nabla \cdot \frac{C_j}{1 + \sum_{\substack{k=0 \\ k \neq j}}^M C_k}}_{\text{convection}} \left\{ \underbrace{\frac{\rho}{RT} \left[\mu_j^{\circ SL} \nabla p(\phi) + \frac{W_j}{2} \nabla g(\phi) \right]}_{\text{phase transformation}} - \underbrace{\sum_{\substack{i=0 \\ i \neq j}}^M \nabla C_i}_{\text{counter diffusion}} \right\}$$

```

>>> for Cj in interstitials:
...     phaseTransformation = (rho.harmonicFaceValue / (R * T)) \
...         * (Cj.standardPotential * p(phase).faceGrad
...           + 0.5 * Cj.barrier * g(phase).faceGrad)
...
...     CkSum = CellVariable(mesh=mesh, value=0.)
...     for Ck in [Ck for Ck in interstitials if Ck is not Cj]:
...         CkSum += Ck
...
...     counterDiffusion = CkSum.faceGrad
...
...     convectionCoeff = counterDiffusion + phaseTransformation
...     convectionCoeff *= (Cj.diffusivity
...                          / (1. + CkSum.harmonicFaceValue))
...
...     Cj.equation = (TransientTerm()
...                    == DiffusionTerm(coeff=Cj.diffusivity)
...                    + PowerLawConvectionTerm(coeff=convectionCoeff))
    
```

The canonical form of the substitutional diffusion equations is

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}} + \underbrace{D_j \nabla \cdot \frac{C_j}{1 - \sum_{\substack{k=M+1 \\ k \neq j}}^{N-1}} C_k}_{\text{convection}} \left\{ \overbrace{\frac{C_N}{RT} \left[(\mu_j^{\circ SL} - \mu_N^{\circ SL}) \nabla p(\phi) + \frac{W_j - W_N}{2} \nabla g(\phi) \right]}^{\text{phase transformation}} + \overbrace{\sum_{\substack{i=M+1 \\ i \neq j}}^{N-1} \nabla C_i}_{\text{counter diffusion}} \right\}$$

```

>>> for Cj in substitutionals:
...     phaseTransformation = (solvent.harmonicFaceValue / (R * T)) \
...         * ((Cj.standardPotential - solvent.standardPotential) * p(phase).faceGrad
...           + 0.5 * (Cj.barrier - solvent.barrier) * g(phase).faceGrad)
...
...     CkSum = CellVariable(mesh=mesh, value=0.)
...     for Ck in [Ck for Ck in substitutionals if Ck is not Cj]:
...         CkSum += Ck
...
...     counterDiffusion = CkSum.faceGrad
...
...     convectionCoeff = counterDiffusion + phaseTransformation
...     convectionCoeff *= (Cj.diffusivity
...                          / (1. - CkSum.harmonicFaceValue))
...
...     Cj.equation = (TransientTerm()
...                    == DiffusionTerm(coeff=Cj.diffusivity)
...                    + PowerLawConvectionTerm(coeff=convectionCoeff))
    
```

We start with a sharp phase boundary

$$\xi = \begin{cases} 1 & \text{for } x \leq L/2, \\ 0 & \text{for } x > L/2, \end{cases}$$

```
>>> x = mesh.cellCenters[0]
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L / 2)
```

and with uniform concentration fields, initially equal to the average of the solidus and liquidus concentrations

```
>>> for Cj in interstitials + substitutionals:
...     Cj.setValue((Cj.S + Cj.L) / 2.)
```

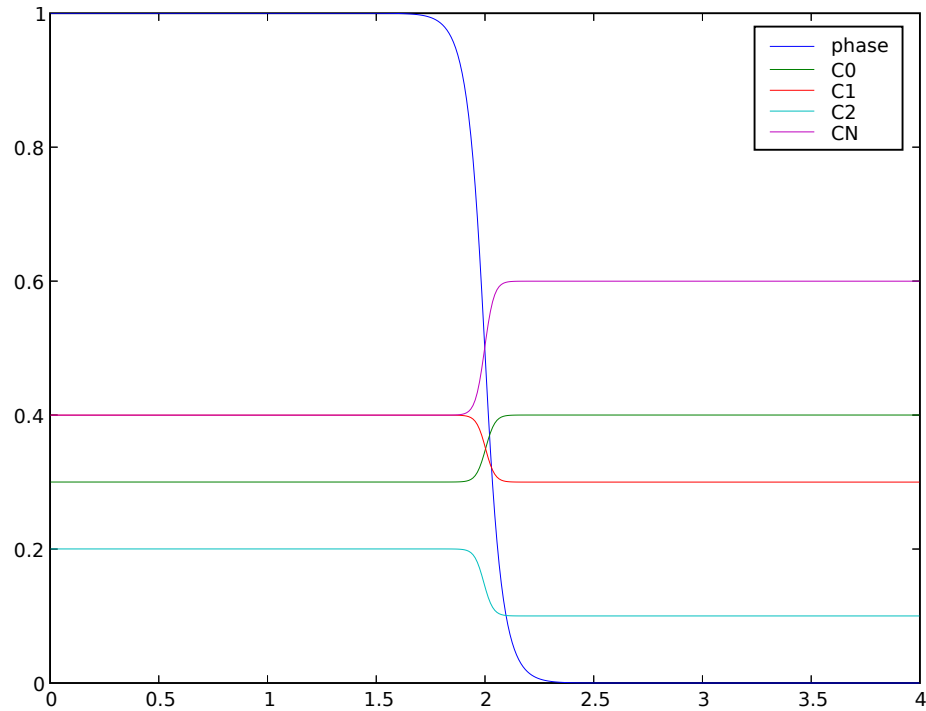
If we're running interactively, we create a viewer

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(phase
...                           + interstitials + substitutionals
...                           + [solvent]),
...                     datamin=0, datamax=1)
...     viewer.plot()
```

and again iterate to equilibrium

```
>>> solver = DefaultAsymmetricSolver(tolerance=1e-10)
```

```
>>> dt = 10000
>>> from builtins import range
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
...     phase.equation.solve(var = phase, dt = dt)
...     for field in substitutionals + interstitials:
...         field.equation.solve(var = field,
...                               dt = dt,
...                               solver = solver)
...     if __name__ == '__main__':
...         viewer.plot()
```

We can confirm that the far-field phases have remained separated

```
>>> X = mesh.faceCenters[0]
>>> print(numerix.allclose(phase.faceValue[X.value==0], 1.0, rtol = 1e-5, atol = 1e-5))
True
>>> print(numerix.allclose(phase.faceValue[X.value==L], 0.0, rtol = 1e-5, atol = 1e-5))
True
```

and that the concentration fields have appropriately segregated into their equilibrium values in each phase

```
>>> equilibrium = True
>>> for Cj in interstitials + substitutionals:
...     equilibrium &= numerix.allclose(Cj.faceValue[X.value==0], Cj.S, rtol = 3e-3, atol = 3e-3).value
...     equilibrium &= numerix.allclose(Cj.faceValue[X.value==L], Cj.L, rtol = 3e-3, atol = 3e-3).value
>>> print(equilibrium)
True
```

21.4 examples.phase.anisotropy

Solve a dendritic solidification problem.

To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as “undercooling” or “supercooling”). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns. Warren, Kobayashi, Lobkovsky and Carter [10] have described a phase field model (“Allen-Cahn”, “non-conserved Ginsberg-Landau”, or “model A” of Hohenberg & Halperin) of such a system, including the effects of discrete crystalline orientations (anisotropy).

We start with a regular 2D Cartesian mesh

```
>>> from fipy import Variable, CellVariable, Grid2D, TransientTerm, DiffusionTerm, \
↳ ImplicitSourceTerm, Viewer, Matplotlib2DGridViewer
>>> from fipy.tools import numerix
>>> dx = dy = 0.025
>>> if __name__ == '__main__':
...     nx = ny = 500
... else:
...     nx = ny = 20
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

and we’ll take fixed timesteps

```
>>> dt = 5e-4
```

We consider the simultaneous evolution of a “phase field” variable ϕ (taken to be 0 in the liquid phase and 1 in the solid)

```
>>> phase = CellVariable(name=r'$\phi$', mesh=mesh, hasOld=True)
```

and a dimensionless undercooling ΔT ($\Delta T = 0$ at the melting point)

```
>>> dT = CellVariable(name=r'$\Delta T$', mesh=mesh, hasOld=True)
```

The `hasOld` flag causes the storage of the value of variable from the previous timestep. This is necessary for solving equations with non-linear coefficients or for coupling between PDEs.

The governing equation for the temperature field is the heat flux equation, with a source due to the latent heat of solidification

$$\frac{\partial \Delta T}{\partial t} = D_T \nabla^2 \Delta T + \frac{\partial \phi}{\partial t}$$

```
>>> DT = 2.25
>>> heatEq = (TransientTerm()
...           == DiffusionTerm(DT)
...           + (phase - phase.old) / dt)
```

The governing equation for the phase field is

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \phi + \phi(1 - \phi)m(\phi, \Delta T)$$

where

$$m(\phi, \Delta T) = \phi - \frac{1}{2} - \frac{\kappa_1}{\pi} \arctan(\kappa_2 \Delta T)$$

represents a source of anisotropy. The coefficient D is an anisotropic diffusion tensor in two dimensions

$$D = \alpha^2 (1 + c\beta) \begin{bmatrix} 1 + c\beta & -c\frac{\partial\beta}{\partial\psi} \\ c\frac{\partial\beta}{\partial\psi} & 1 + c\beta \end{bmatrix}$$

where $\beta = \frac{1-\Phi^2}{1+\Phi^2}$, $\Phi = \tan\left(\frac{N}{2}\psi\right)$, $\psi = \theta + \arctan\frac{\partial\phi/\partial y}{\partial\phi/\partial x}$, θ is the orientation, and N is the symmetry.

```
>>> alpha = 0.015
>>> c = 0.02
>>> N = 6.
>>> theta = numerix.pi / 8.
>>> psi = theta + numerix.arctan2(phase.faceGrad[1],
...                               phase.faceGrad[0])
>>> Phi = numerix.tan(N * psi / 2)
>>> PhiSq = Phi**2
>>> beta = (1. - PhiSq) / (1. + PhiSq)
>>> DbetaDpsi = -N * 2 * Phi / (1 + PhiSq)
>>> Ddia = (1. + c * beta)
>>> Doff = c * DbetaDpsi
>>> I0 = Variable(value=((1, 0), (0, 1)))
>>> I1 = Variable(value=((0, -1), (1, 0)))
>>> D = alpha**2 * (1. + c * beta) * (Ddia * I0 + Doff * I1)
```

With these expressions defined, we can construct the phase field equation as

```
>>> tau = 3e-4
>>> kappa1 = 0.9
>>> kappa2 = 20.
>>> phaseEq = (TransientTerm(tau)
...             == DiffusionTerm(D)
...             + ImplicitSourceTerm((phase - 0.5 - kappa1 / numerix.pi * numerix.
...             ↪arctan(kappa2 * dT))
...             * (1 - phase)))
```

We seed a circular solidified region in the center

```
>>> radius = dx * 5.
>>> C = (nx * dx / 2, ny * dy / 2)
>>> x, y = mesh.cellCenters
>>> phase.setValue(1., where=((x - C[0])**2 + (y - C[1])**2) < radius**2)
```

and quench the entire simulation domain below the melting point

```
>>> dT.setValue(-0.5)
```

In a real solidification process, dendritic branching is induced by small thermal fluctuations along an otherwise smooth surface, but the granularity of the *Mesh* is enough “noise” in this case, so we don’t need to explicitly introduce randomness, the way we did in the Cahn-Hilliard problem.

FiPy’s viewers are utilitarian, striving to let the user see *something*, regardless of their operating system or installed packages, so you won’t be able to simultaneously view two fields “out of the box”, but, because all of Python is accessible and FiPy is object oriented, it is not hard to adapt one of the existing viewers to create a specialized display:

```
>>> if __name__ == "__main__":
...     try:
...         import pylab
...         class DendriteViewer(Matplotlib2DGridViewer):
```

(continues on next page)

(continued from previous page)

```

...     def __init__(self, phase, dT, title=None, limits={}, **kwlimits):
...         self.phase = phase
...         self.contour = None
...         Matplotlib2DGridViewer.__init__(self, vars=(dT,), title=title,
...                                         cmap=pylab.cm.hot,
...                                         limits=limits, **kwlimits)
...
...     def _plot(self):
...         Matplotlib2DGridViewer._plot(self)
...
...         if self.contour is not None:
...             for c in self.contour.collections:
...                 c.remove()
...
...         mesh = self.phase.mesh
...         shape = mesh.shape
...         x, y = mesh.cellCenters
...         z = self.phase.value
...         x, y, z = [a.reshape(shape, order='F') for a in (x, y, z)]
...
...         self.contour = self.axes.contour(x, y, z, (0.5,))
...
...     viewer = DendriteViewer(phase=phase, dT=dT,
...                             title=r"%s & %s" % (phase.name, dT.name),
...                             datamin=-0.1, datamax=0.05)
... except ImportError:
...     viewer = MultiViewer(viewers=(Viewer(vars=phase),
...                                       Viewer(vars=dT,
...                                             datamin=-0.5,
...                                             datamax=0.5)))

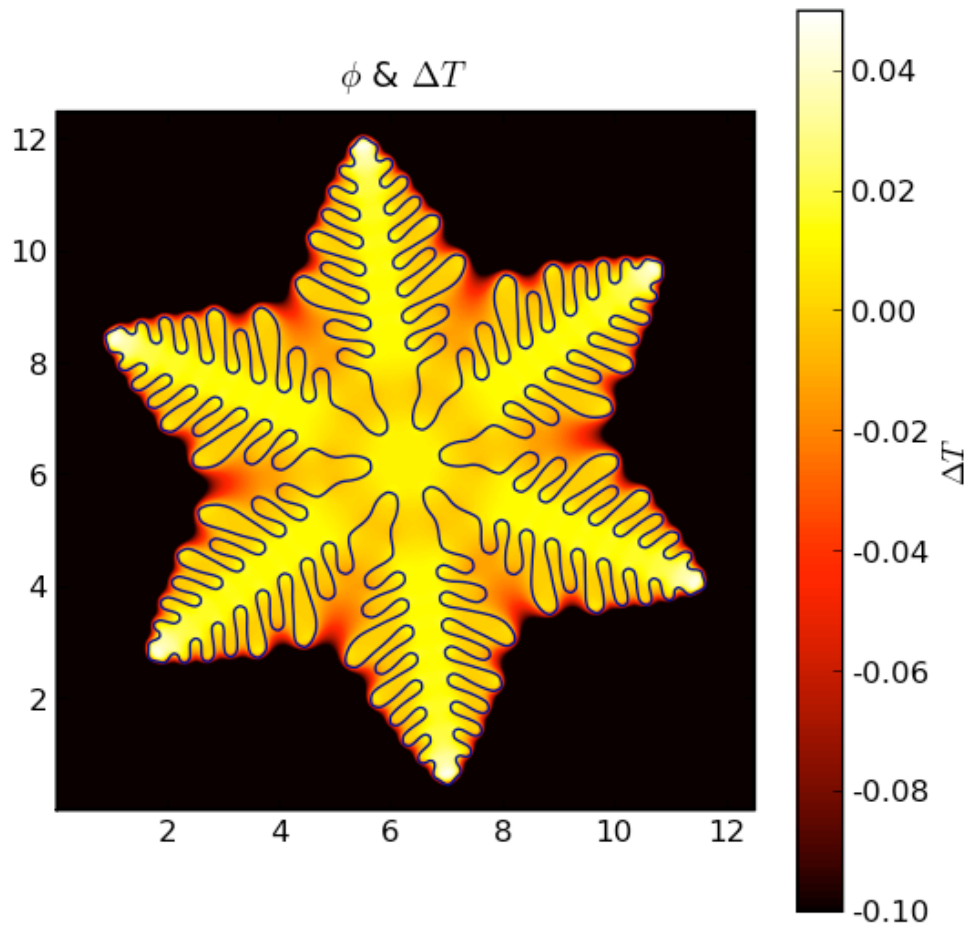
```

and iterate the solution in time, plotting as we go,

```

>>> if __name__ == '__main__':
...     steps = 10000
... else:
...     steps = 10
>>> from builtins import range
>>> for i in range(steps):
...     phase.updateOld()
...     dT.updateOld()
...     phaseEq.solve(phase, dt=dt)
...     heatEq.solve(dT, dt=dt)
...     if __name__ == "__main__" and (i % 10 == 0):
...         viewer.plot()

```



The non-uniform temperature results from the release of latent heat at the solidifying interface. The dendrite arms grow fastest where the temperature gradient is steepest.

We note that this FiPy simulation is written in about 50 lines of code (excluding the custom viewer), compared with over 800 lines of (fairly lucid) FORTRAN code used for the figures in [10].

21.5 examples.phase.impingement.mesh40x1

Solve for the impingement of two grains in one dimension.

In this example we solve a coupled phase and orientation equation on a one dimensional grid. This is another aspect of the model of Warren, Kobayashi, Lobkovsky and Carter [10]

```
>>> from fipy import CellVariable, ModularVariable, Grid1D, TransientTerm, \
↳ DiffusionTerm, ExplicitDiffusionTerm, ImplicitSourceTerm, GeneralSolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> nx = 40
>>> Lx = 2.5 * nx / 100.
>>> dx = Lx / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

This problem simulates the wet boundary that forms between grains of different orientations. The phase equation is given by

$$\tau_\phi \frac{\partial \phi}{\partial t} = \alpha^2 \nabla^2 \phi + \phi(1 - \phi)m_1(\phi, T) - 2s\phi|\nabla\theta| - \epsilon^2\phi|\nabla\theta|^2$$

where

$$m_1(\phi, T) = \phi - \frac{1}{2} - T\phi(1 - \phi)$$

and the orientation equation is given by

$$P(\epsilon|\nabla\theta)|\tau_\theta\phi^2\frac{\partial\theta}{\partial t} = \nabla \cdot \left[\phi^2 \left(\frac{s}{|\nabla\theta|} + \epsilon^2 \right) \nabla\theta \right]$$

where

$$P(w) = 1 - \exp(-\beta w) + \frac{\mu}{\epsilon} \exp(-\beta w)$$

The initial conditions for this problem are set such that $\phi = 1$ for $0 \leq x \leq L_x$ and

$$\theta = \begin{cases} 1 & \text{for } 0 \leq x < L_x/2, \\ 0 & \text{for } L_x/2 \leq x \leq L_x. \end{cases}$$

Here the phase and orientation equations are solved with an explicit and implicit technique respectively.

The parameters for these equations are

```
>>> timeStepDuration = 0.02
>>> phaseTransientCoeff = 0.1
>>> thetaSmallValue = 1e-6
>>> beta = 1e5
>>> mu = 1e3
>>> thetaTransientCoeff = 0.01
>>> gamma = 1e3
>>> epsilon = 0.008
>>> s = 0.01
>>> alpha = 0.015
```

The system is held isothermal at

```
>>> temperature = 1.
```

and is initially solid everywhere

```
>>> phase = CellVariable(
...     name='phase field',
...     mesh=mesh,
...     value=1.
... )
```

Because `theta` is an S^1 -valued variable (i.e. it maps to the circle) and thus intrinsically has 2π -periodicity, we must use `ModularVariable` instead of a `CellVariable`. A `ModularVariable` confines `theta` to $-\pi < \theta \leq \pi$ by adding or subtracting 2π where necessary and by defining a new subtraction operator between two angles.

```
>>> theta = ModularVariable(
...     name='theta',
...     mesh=mesh,
...     value=1.,
...     hasOld=1
... )
```

The left and right halves of the domain are given different orientations.

```
>>> theta.setValue(0., where=mesh.cellCenters[0] > Lx / 2.)
```

The phase equation is built in the following way.

```
>>> mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
```

The source term is linearized in the manner demonstrated in [examples.phase.simple](#) (Kobayashi, semi-implicit).

```
>>> thetaMag = theta.old.grad.mag
>>> implicitSource = mPhiVar * (phase - (mPhiVar < 0))
>>> implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
```

The phase equation is constructed.

```
>>> phaseEq = TransientTerm(phaseTransientCoeff) \
... == ExplicitDiffusionTerm(alpha**2) \
...     - ImplicitSourceTerm(implicitSource) \
...     + (mPhiVar > 0) * mPhiVar * phase
```

The `theta` equation is built in the following way. The details for this equation are fairly involved, see J. A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of `theta` on the circle.

```
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> phaseModSq = phaseMod * phaseMod
>>> expo = epsilon * beta * theta.grad.mag
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> pFunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
```

```
>>> phaseFace = phase.arithmeticFaceValue
>>> phaseSq = phaseFace * phaseFace
>>> gradMag = theta.faceGrad.mag
>>> eps = 1. / gamma / 10.
>>> gradMag += (gradMag < eps) * eps
>>> IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
>>> diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. `theta.faceGradNoMod` evaluates the gradient without modular arithmetic.

```
>>> thetaGradDiff = theta.faceGrad - theta.faceGradNoMod
>>> sourceCoeff = (diffusionCoeff * thetaGradDiff).divergence
```

Finally the `theta` equation can be constructed.

```
>>> thetaEq = TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...     DiffusionTerm(diffusionCoeff) \
...     + sourceCoeff
```

If the example is run interactively, we create viewers for the phase and orientation variables.

```
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase, datamin=0., datamax=1.)
...     thetaProductViewer = Viewer(vars=theta,
...                                 datamin=-numerix.pi, datamax=numerix.pi)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

we iterate the solution in time, plotting as we go if running interactively,

```
>>> steps = 10
>>> from builtins import range
>>> for i in range(steps):
...     theta.updateOld()
...     thetaEq.solve(theta, dt = timeStepDuration)
...     phaseEq.solve(phase, dt = timeStepDuration)
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared with test data. The test data was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `mesh40x1.gz` extracts the data and compares it with the `theta` variable.

```
>>> import os
>>> from future.utils import text_to_native_str
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + text_to_native_str('.gz
↳'))
>>> testData = CellVariable(mesh=mesh, value=testData)
>>> print(theta.allclose(testData))
1
```

21.6 examples.phase.impingement.mesh20x20

Solve for the impingement of four grains in two dimensions.

In the following examples, we solve the same set of equations as in `examples.phase.impingement.mesh40x1` with different initial conditions and a 2D mesh:

```
>>> from fipy.tools.parser import parse
```

```
>>> numberOfElements = parse('--numberOfElements', action = 'store',
...                           type = 'int', default = 400)
>>> numberOfSteps = parse('--numberOfSteps', action = 'store',
...                       type = 'int', default = 10)
```

```
>>> from fipy import CellVariable, ModularVariable, Grid2D, TransientTerm,
↳DiffusionTerm, ExplicitDiffusionTerm, ImplicitSourceTerm, GeneralSolver, Viewer
>>> from fipy.tools import numerix, dump
```

```
>>> steps = numberOfSteps
>>> N = int(numerix.sqrt(numberOfElements))
>>> L = 2.5 * N / 100.
```

(continues on next page)

(continued from previous page)

```
>>> dL = L / N
>>> mesh = Grid2D(dx=dL, dy=dL, nx=N, ny=N)
```

The initial conditions are given by $\phi = 1$ and

$$\theta = \begin{cases} \frac{2\pi}{3} & \text{for } x^2 - y^2 < L/2, \\ \frac{-2\pi}{3} & \text{for } (x - L)^2 - y^2 < L/2, \\ \frac{-2\pi}{3} + 0.3 & \text{for } x^2 - (y - L)^2 < L/2, \\ \frac{2\pi}{3} & \text{for } (x - L)^2 - (y - L)^2 < L/2. \end{cases}$$

This defines four solid regions with different orientations. Solidification occurs and then boundary wetting occurs where the orientation varies.

The parameters for this example are

```
>>> timeStepDuration = 0.02
>>> phaseTransientCoeff = 0.1
>>> thetaSmallValue = 1e-6
>>> beta = 1e5
>>> mu = 1e3
>>> thetaTransientCoeff = 0.01
>>> gamma = 1e3
>>> epsilon = 0.008
>>> s = 0.01
>>> alpha = 0.015
```

The system is held isothermal at

```
>>> temperature = 10.
```

and is initialized to liquid everywhere

```
>>> phase = CellVariable(name='phase field', mesh=mesh)
```

The orientation is initialized to a uniform value to denote the randomly oriented liquid phase

```
>>> theta = ModularVariable(
...     name='theta',
...     mesh=mesh,
...     value=-numerix.pi + 0.0001,
...     hasOld=1
... )
```

Four different solid circular domains are created at each corner of the domain with appropriate orientations

```
>>> x, y = mesh.cellCenters
>>> for a, b, thetaValue in ((0., 0., 2. * numerix.pi / 3.),
...                           (L, 0., -2. * numerix.pi / 3.),
...                           (0., L, -2. * numerix.pi / 3. + 0.3),
...                           (L, L, 2. * numerix.pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

The phase equation is built in the following way. The source term is linearized in the manner demonstrated in [examples.phase.simple](#) (Kobayashi, semi-implicit). Here we use a function to build the equation, so that it can be reused later.

```
>>> def buildPhaseEquation(phase, theta):
...
...     mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
...     thetaMag = theta.old.grad.mag
...     implicitSource = mPhiVar * (phase - (mPhiVar < 0))
...     implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
...
...     return TransientTerm(phaseTransientCoeff) == \
...         ExplicitDiffusionTerm(alpha**2) \
...         - ImplicitSourceTerm(implicitSource) \
...         + (mPhiVar > 0) * mPhiVar * phase
```

```
>>> phaseEq = buildPhaseEquation(phase, theta)
```

The theta equation is built in the following way. The details for this equation are fairly involved, see J. A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of theta on the circle. The source term requires the evaluation of the face gradient without the modular operators.

```
>>> def buildThetaEquation(phase, theta):
...
...     phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
...     phaseModSq = phaseMod * phaseMod
...     expo = epsilon * beta * theta.grad.mag
...     expo = (expo < 100.) * (expo - 100.) + 100.
...     pFunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
...
...     phaseFace = phase.arithmeticFaceValue
...     phaseSq = phaseFace * phaseFace
...     gradMag = theta.faceGrad.mag
...     eps = 1. / gamma / 10.
...     gradMag += (gradMag < eps) * eps
...     IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
...     diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
...
...     thetaGradDiff = theta.faceGrad - theta.faceGradNoMod
...     sourceCoeff = (diffusionCoeff * thetaGradDiff).divergence
...
...     return TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...         DiffusionTerm(diffusionCoeff) \
...         + sourceCoeff
```

```
>>> thetaEq = buildThetaEquation(phase, theta)
```

If the example is run interactively, we create viewers for the phase and orientation variables. Rather than viewing the raw orientation, which is not meaningful in the liquid phase, we weight the orientation by the phase

```
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase, datamin=0., datamax=1.)
...     thetaProd = -numerix.pi + phase * (theta + numerix.pi)
...     thetaProductViewer = Viewer(vars=thetaProd,
...                                   datamin=-numerix.pi, datamax=numerix.pi)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

The solution will be tested against data that was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `mesh20x20.gz` extracts the data and compares it with the *theta* variable.

```
>>> import os
>>> from future.utils import text_to_native_str
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + text_to_native_str('.gz
↳')) .flat
```

We step the solution in time, plotting as we go if running interactively,

```
>>> from builtins import range
>>> for i in range(steps):
...     theta.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration,
↳ solver=GeneralSolver(iterations=2000, tolerance=1e-15))
...     phaseEq.solve(phase, dt=timeStepDuration,
↳ solver=GeneralSolver(iterations=2000, tolerance=1e-15))
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared against Ryo Kobayashi's test data

```
>>> print(theta.allclose(testData, rtol=1e-7, atol=1e-7))
1
```

The following code shows how to restart a simulation from some saved data. First, reset the variables to their original values.

```
>>> phase.setValue(0)
>>> theta.setValue(-numerix.pi + 0.0001)
>>> x, y = mesh.cellCenters
>>> for a, b, thetaValue in ((0., 0., 2. * numerix.pi / 3.),
...                           (L, 0., -2. * numerix.pi / 3.),
...                           (0., L, -2. * numerix.pi / 3. + 0.3),
...                           (L, L, 2. * numerix.pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

Step through half the time steps.

```
>>> from builtins import range
>>> for i in range(steps // 2):
...     theta.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration,
↳ solver=GeneralSolver(iterations=2000, tolerance=1e-15))
...     phaseEq.solve(phase, dt=timeStepDuration,
↳ solver=GeneralSolver(iterations=2000, tolerance=1e-15))
```

We confirm that the solution has not yet converged to that given by Ryo Kobayashi's FORTRAN code:

```
>>> print(theta.allclose(testData))
0
```

We save the variables to disk.

```
>>> (f, filename) = dump.write({'phase' : phase, 'theta' : theta}, extension = '.gz')
```

and then recall them to test the data pickling mechanism

```
>>> data = dump.read(filename, f)
>>> newPhase = data['phase']
>>> newTheta = data['theta']
>>> newThetaEq = buildThetaEquation(newPhase, newTheta)
>>> newPhaseEq = buildPhaseEquation(newPhase, newTheta)
```

and finish the iterations,

```
>>> from builtins import range
>>> for i in range(steps // 2):
...     newTheta.updateOld()
...     newThetaEq.solve(newTheta, dt=timeStepDuration,
↳ solver=GeneralSolver(iterations=2000, tolerance=1e-15))
...     newPhaseEq.solve(newPhase, dt=timeStepDuration,
↳ solver=GeneralSolver(iterations=2000, tolerance=1e-15))
```

The solution is compared against Ryo Kobayashi’s test data

```
>>> print(newTheta.allclose(testData, rtol=1e-7))
1
```

21.7 examples.phase.polyxtal

Solve the dendritic growth of nuclei and subsequent grain impingement.

To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as “undercooling” or “supercooling”). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns. Warren, Kobayashi, Lobkovsky and Carter [10] have described a phase field model (“Allen-Cahn”, “non-conserved Ginsberg-Landau”, or “model A” of Hohenberg & Halperin) of such a system, including the effects of discrete crystalline orientations (anisotropy).

We start with a regular 2D Cartesian mesh

```
>>> from fipy import CellVariable, Variable, ModularVariable, Grid2D, TransientTerm,
↳ DiffusionTerm, ImplicitSourceTerm, MatplotlibViewer, Matplotlib2DGridViewer,
↳ MultiViewer
>>> from fipy.tools import numerix
>>> dx = dy = 0.025
>>> if __name__ == "__main__":
...     nx = ny = 200
... else:
...     nx = ny = 200
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

and we’ll take fixed timesteps

```
>>> dt = 5e-4
```

We consider the simultaneous evolution of a “phase field” variable ϕ (taken to be 0 in the liquid phase and 1 in the solid)

```
>>> phase = CellVariable(name=r'$\phi$', mesh=mesh, hasOld=True)
```

a dimensionless undercooling ΔT ($\Delta T = 0$ at the melting point)

```
>>> dT = CellVariable(name=r'\Delta T$', mesh=mesh, hasOld=True)
```

and an orientation $-\pi < \theta \leq \pi$

```
>>> theta = ModularVariable(name=r'\theta$', mesh=mesh, hasOld=True)
>>> theta.value = -numerix.pi + 0.0001
```

The `hasOld` flag causes the storage of the value of variable from the previous timestep. This is necessary for solving equations with non-linear coefficients or for coupling between PDEs.

The governing equation for the temperature field is the heat flux equation, with a source due to the latent heat of solidification

$$\frac{\partial \Delta T}{\partial t} = D_T \nabla^2 \Delta T + \frac{\partial \phi}{\partial t} + c(T_0 - T)$$

```
>>> DT = 2.25
>>> q = Variable(0.)
>>> T_0 = -0.1
>>> heatEq = (TransientTerm()
...           == DiffusionTerm(DT)
...           + (phase - phase.old) / dt
...           + q * T_0 - ImplicitSourceTerm(q))
```

The governing equation for the phase field is

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \phi + \phi(1 - \phi)m(\phi, \Delta T) - 2s\phi|\nabla \theta| - \epsilon^2 \phi |\nabla \theta|^2$$

where

$$m(\phi, \Delta T) = \phi - \frac{1}{2} - \frac{\kappa_1}{\pi} \arctan(\kappa_2 \Delta T)$$

represents a source of anisotropy. The coefficient D is an anisotropic diffusion tensor in two dimensions

$$D = \alpha^2 (1 + c\beta) \begin{bmatrix} 1 + c\beta & -c \frac{\partial \beta}{\partial \psi} \\ c \frac{\partial \beta}{\partial \psi} & 1 + c\beta \end{bmatrix}$$

where $\beta = \frac{1-\Phi^2}{1+\Phi^2}$, $\Phi = \tan\left(\frac{N}{2}\psi\right)$, $\psi = \theta + \arctan\left(\frac{\partial \phi / \partial y}{\partial \phi / \partial x}\right)$, θ is the orientation, and N is the symmetry.

```
>>> alpha = 0.015
>>> c = 0.02
>>> N = 4.
```

```
>>> psi = theta.arithmeticFaceValue + numerix.arctan2(phase.faceGrad[1],
...                                                    phase.faceGrad[0])
>>> Phi = numerix.tan(N * psi / 2)
>>> PhiSq = Phi**2
>>> beta = (1. - PhiSq) / (1. + PhiSq)
>>> DbetaDpsi = -N * 2 * Phi / (1 + PhiSq)
>>> Ddia = (1. + c * beta)
```

```
>>> Doff = c * DbetaDpsi
>>> I0 = Variable(value=((1, 0), (0, 1)))
>>> I1 = Variable(value=((0, -1), (1, 0)))
>>> D = alpha**2 * Ddia * (Ddia * I0 + Doff * I1)
```

With these expressions defined, we can construct the phase field equation as

```
>>> tau_phase = 3e-4
>>> kappa1 = 0.9
>>> kappa2 = 20.
>>> epsilon = 0.008
>>> s = 0.01
>>> thetaMag = theta.grad.mag
>>> phaseEq = (TransientTerm(tau_phase)
...           == DiffusionTerm(D)
...           + ImplicitSourceTerm((phase - 0.5 - kappa1 / numerix.pi * numerix.
->arctan(kappa2 * dT))
...                               * (1 - phase)
...                               - (2 * s + epsilon**2 * thetaMag) * thetaMag))
```

The governing equation for orientation is given by

$$P(\epsilon|\nabla\theta|)\tau_\theta\phi^2\frac{\partial\theta}{\partial t} = \nabla \cdot \left[\phi^2 \left(\frac{s}{|\nabla\theta|} + \epsilon^2 \right) \nabla\theta \right]$$

where

$$P(w) = 1 - \exp(-\beta w) + \frac{\mu}{\epsilon} \exp(-\beta w)$$

The theta equation is built in the following way. The details for this equation are fairly involved, see J. A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of theta on the circle.

```
>>> tau_theta = 3e-3
>>> mu = 1e3
>>> gamma = 1e3
>>> thetaSmallValue = 1e-6
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> beta_theta = 1e5
>>> expo = epsilon * beta_theta * theta.grad.mag
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> Pfunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
```

```
>>> gradMagTheta = theta.faceGrad.mag
>>> eps = 1. / gamma / 10.
>>> gradMagTheta += (gradMagTheta < eps) * eps
>>> IGamma = (gradMagTheta > 1. / gamma) * (1 / gradMagTheta - gamma) + gamma
>>> v_theta = phase.arithmeticFaceValue * (s * IGamma + epsilon**2)
>>> D_theta = phase.arithmeticFaceValue**2 * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. `theta.faceGradNoMod` evaluates the gradient without modular arithmetic.

```
>>> thetaEq = (TransientTerm(tau_theta * phaseMod**2 * Pfunc)
...           == DiffusionTerm(D_theta)
...           + (D_theta * (theta.faceGrad - theta.faceGradNoMod)).divergence)
```

We seed a circular solidified region in the center

```
>>> x, y = mesh.cellCenters
>>> numSeeds = 10
>>> numerix.random.seed(12345)
>>> for Cx, Cy, orientation in numerix.random.random([numSeeds, 3]):
```

(continues on next page)

(continued from previous page)

```

...     radius = dx * 5.
...     seed = ((x - Cx * nx * dx)**2 + (y - Cy * ny * dy)**2) < radius**2
...     phase[seed] = 1.
...     theta[seed] = numerix.pi * (2 * orientation - 1)

```

and quench the entire simulation domain below the melting point

```
>>> dT.setValue(-0.5)
```

In a real solidification process, dendritic branching is induced by small thermal fluctuations along an otherwise smooth surface, but the granularity of the *Mesh* is enough “noise” in this case, so we don’t need to explicitly introduce randomness, the way we did in the Cahn-Hilliard problem.

FiPy’s viewers are utilitarian, striving to let the user see *something*, regardless of their operating system or installed packages, so you the default color scheme of grain orientation won’t be very informative “out of the box”. Because all of Python is accessible and FiPy is object oriented, it is not hard to adapt one of the existing viewers to create a specialized display:

```

>>> from builtins import zip
>>> if __name__ == "__main__":
...     try:
...         class OrientationViewer(Matplotlib2DGridViewer):
...             def __init__(self, phase, orientation, title=None, limits={},
... **kwlimits):
...                 self.phase = phase
...                 Matplotlib2DGridViewer.__init__(self, vars=(orientation,),
... title=title,
... limits=limits, colorbar=None,
... **kwlimits)
...
...                 # make room for non-existent colorbar
...                 # stolen from matplotlib.colorbar.make_axes
...                 # https://github.com/matplotlib/matplotlib/blob
...                 # /ec1cd2567521c105a451ce15e06de10715f8b54d/lib
...                 # /matplotlib.colorbar.py#L838
...                 fraction = 0.15
...                 pb = self.axes.get_position(original=True).frozen()
...                 pad = 0.05
...                 x1 = 1.0-fraction
...                 pb1, pbx, pbcx = pb.splitx(x1-pad, x1)
...                 panchor = (1.0, 0.5)
...                 self.axes.set_position(pb1)
...                 self.axes.set_anchor(panchor)
...
...                 # make the gnomon
...                 fig = self.axes.get_figure()
...                 self.gnomon = fig.add_axes([0.85, 0.425, 0.15, 0.15], polar=True)
...                 self.gnomon.set_thetagrids([180, 270, 0, 90],
... [r"$\pm\pi$", r"$-\frac{\pi}{2}$", "$0$
... ", r"$+\frac{\pi}{2}$"],
... frac=1.3)
...                 self.gnomon.set_theta_zero_location("N")
...                 self.gnomon.set_theta_direction(-1)
...                 self.gnomon.set_rgrids([1.], [""])
...                 N = 100
...                 theta = numerix.arange(-numerix.pi, numerix.pi, 2 * numerix.pi /
... N)

```

(continues on next page)

(continued from previous page)

```

...         radii = numerix.ones((N,))
...         bars = self.gnomon.bar(theta, radii, width=2 * numerix.pi / N,
↳bottom=0.0)
...         colors = self._orientation_and_phase_to_rgb(orientation=numerix.
↳array([theta]), phase=1.)
...         for c, t, bar in zip(colors[0], theta, bars):
...             bar.set_facecolor(c)
...             bar.set_edgecolor(c)
...
...     def _reshape(self, var):
...         '''return values of var in an 2D array'''
...         return numerix.reshape(numerix.array(var),
...                                 var.mesh.shape[:::-1])[::-1]
...
...     @staticmethod
...     def _orientation_and_phase_to_rgb(orientation, phase):
...         from matplotlib import colors
...
...         hsv = numerix.empty(orientation.shape + (3,))
...         hsv[..., 0] = (orientation / numerix.pi + 1) / 2.
...         hsv[..., 1] = 1.
...         hsv[..., 2] = phase
...
...         return colors.hsv_to_rgb(hsv)
...
...     @property
...     def _data(self):
...         '''convert phase and orientation to rgb image array
...
...         orientation (-pi, pi) -> hue (0, 1)
...         phase (0, 1) -> value (0, 1)
...         '''
...         orientation = self._reshape(self.vars[0])
...         phase = self._reshape(self.phase)
...
...         return self._orientation_and_phase_to_rgb(orientation, phase)
...
...     def _plot(self):
...         self.image.set_data(self._data)
...
...     from matplotlib import pyplot
...     pyplot.ion()
...     w, h = pyplot.figaspect(1.)
...     fig = pyplot.figure(figsize=(2*w, h))
...     timer = fig.text(0.1, 0.9, "t = %.3f" % 0, fontsize=18)
...
...     viewer = MultiViewer(viewers=(MatplotlibViewer(vars=dT,
...                                                    cmap=pyplot.cm.hot,
...                                                    datamin=-0.5,
...                                                    datamax=0.5,
...                                                    axes=fig.add_subplot(121)),
...                                OrientationViewer(phase=phase,
...                                                    orientation=theta,
...                                                    title=theta.name,
...                                                    axes=fig.add_
↳subplot(122)))
...     except ImportError:

```

(continues on next page)

(continued from previous page)

```

...     viewer = MultiViewer(viewers=(Viewer(vars=dT,
...                                         datamin=-0.5,
...                                         datamax=0.5),
...                                     Viewer(vars=phase,
...                                         datamin=0.,
...                                         datamax=1.),
...                                     Viewer(vars=theta,
...                                         datamin=-numerix.pi,
...                                         datamax=numerix.pi)))
...     viewer.plot()

```

and iterate the solution in time, plotting as we go,

```

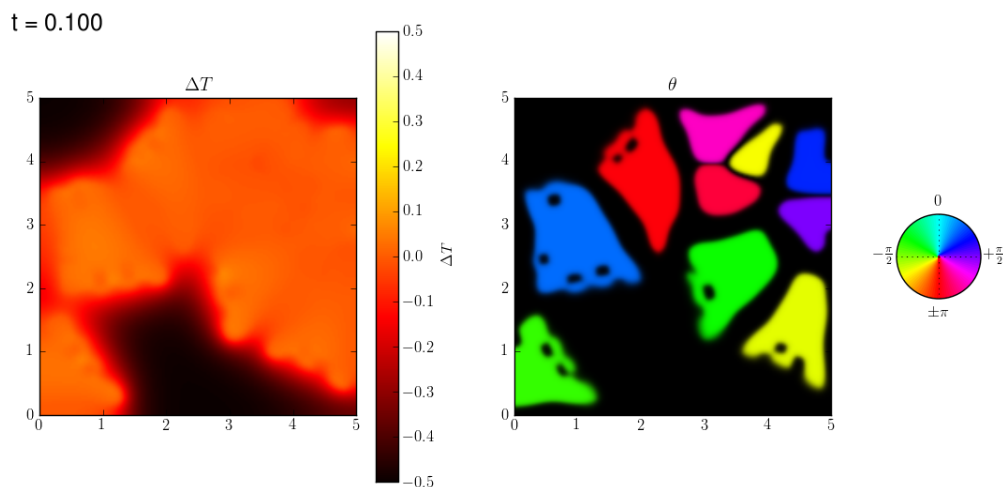
>>> if __name__ == "__main__":
...     total_time = 2.
...     else:
...         total_time = dt * 10
>>> elapsed = 0.
>>> save_interval = 0.002
>>> save_at = save_interval

```

```

>>> while elapsed < total_time:
...     if elapsed > 0.3:
...         q.value = 100
...     phase.updateOld()
...     dT.updateOld()
...     theta.updateOld()
...     thetaEq.solve(theta, dt=dt)
...     phaseEq.solve(phase, dt=dt)
...     heatEq.solve(dT, dt=dt)
...     elapsed += dt
...     if __name__ == "__main__" and elapsed >= save_at:
...         timer.set_text("t = %.3f" % elapsed)
...         viewer.plot()
...         save_at += save_interval

```



The non-uniform temperature results from the release of latent heat at the solidifying interface. The dendrite arms

grow fastest where the temperature gradient is steepest.

21.8 examples.phase.polyxtalCoupled

Simultaneously solve the dendritic growth of nuclei and subsequent grain impingement.

To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as “undercooling” or “supercooling”). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns. Warren, Kobayashi, Lobkovsky and Carter [10] have described a phase field model (“Allen-Cahn”, “non-conserved Ginsberg-Landau”, or “model A” of Hohenberg & Halperin) of such a system, including the effects of discrete crystalline orientations (anisotropy).

We start with a regular 2D Cartesian mesh

```
>>> from fipy import CellVariable, Variable, ModularVariable, Grid2D, TransientTerm, \
↳ DiffusionTerm, ImplicitSourceTerm, PowerLawConvectionTerm, MatplotlibViewer, \
↳ Matplotlib2DGridViewer, MultiViewer
>>> from fipy.tools import numerix
>>> dx = dy = 0.025
>>> if __name__ == "__main__":
...     nx = ny = 200
... else:
...     nx = ny = 20
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

and we'll take fixed timesteps

```
>>> dt = 5e-4
```

We consider the simultaneous evolution of a “phase field” variable ϕ (taken to be 0 in the liquid phase and 1 in the solid)

```
>>> phase = CellVariable(name=r'$\phi$', mesh=mesh, hasOld=True)
```

a dimensionless undercooling ΔT ($\Delta T = 0$ at the melting point)

```
>>> dT = CellVariable(name=r'$\Delta T$', mesh=mesh, hasOld=True)
```

and an orientation $-\pi < \theta \leq \pi$

```
>>> theta = ModularVariable(name=r'$\theta$', mesh=mesh, hasOld=True)
>>> theta.value = -numerix.pi + 0.0001
```

The `hasOld` flag causes the storage of the value of variable from the previous timestep. This is necessary for solving equations with non-linear coefficients or for coupling between PDEs.

The governing equation for the temperature field is the heat flux equation, with a source due to the latent heat of solidification

$$\frac{\partial \Delta T}{\partial t} = D_T \nabla^2 \Delta T + \frac{\partial \phi}{\partial t} + c(T_0 - T)$$

```
>>> DT = 2.25
>>> q = Variable(0.)
>>> T_0 = -0.1
```

(continues on next page)

(continued from previous page)

```
>>> heatEq = (TransientTerm(var=dT)
...           == DiffusionTerm(coeff=DT, var=dT)
...           + TransientTerm(var=phase)
...           + q * T_0 - ImplicitSourceTerm(coeff=q, var=dT))
```

The governing equation for the phase field is

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \phi + \phi(1 - \phi)m(\phi, \Delta T) - 2s\phi|\nabla \theta| - \epsilon^2 \phi |\nabla \theta|^2$$

where

$$m(\phi, \Delta T) = \phi - \frac{1}{2} - \frac{\kappa_1}{\pi} \arctan(\kappa_2 \Delta T)$$

represents a source of anisotropy. The coefficient D is an anisotropic diffusion tensor in two dimensions

$$D = \alpha^2 (1 + c\beta) \begin{bmatrix} 1 + c\beta & -c \frac{\partial \beta}{\partial \psi} \\ c \frac{\partial \beta}{\partial \psi} & 1 + c\beta \end{bmatrix}$$

where $\beta = \frac{1-\Phi^2}{1+\Phi^2}$, $\Phi = \tan\left(\frac{N}{2}\psi\right)$, $\psi = \theta + \arctan \frac{\partial \phi / \partial y}{\partial \phi / \partial x}$, θ is the orientation, and N is the symmetry.

```
>>> alpha = 0.015
>>> c = 0.02
>>> N = 4.
```

```
>>> psi = theta.arithmeticFaceValue + numerix.arctan2(phase.faceGrad[1],
...                                                    phase.faceGrad[0])
>>> Phi = numerix.tan(N * psi / 2)
>>> PhiSq = Phi**2
>>> beta = (1. - PhiSq) / (1. + PhiSq)
>>> DbetaDpsi = -N * 2 * Phi / (1 + PhiSq)
>>> Ddia = (1.+ c * beta)
```

```
>>> Doff = c * DbetaDpsi
>>> I0 = Variable(value=((1, 0), (0, 1)))
>>> I1 = Variable(value=((0, -1), (1, 0)))
>>> D = alpha**2 * Ddia * (Ddia * I0 + Doff * I1)
```

With these expressions defined, we can construct the phase field equation as

```
>>> tau_phase = 3e-4
>>> kappa1 = 0.9
>>> kappa2 = 20.
>>> epsilon = 0.008
>>> s = 0.01
>>> thetaMag = theta.grad.mag
>>> phaseEq = (TransientTerm(coeff=tau_phase, var=phase)
...           == DiffusionTerm(coeff=D, var=phase)
...           + ImplicitSourceTerm(coeff=((phase - 0.5 - kappa1 / numerix.pi *
↪numerix.arctan(kappa2 * dT))
...                                     * (1 - phase)
...                                     - (2 * s + epsilon**2 * thetaMag) *
↪thetaMag),
...           var=phase))
```

The governing equation for orientation is given by

$$P(\epsilon|\nabla\theta)\tau_\theta\phi^2\frac{\partial\theta}{\partial t} = \nabla \cdot \left[\phi^2 \left(\frac{s}{|\nabla\theta|} + \epsilon^2 \right) \nabla\theta \right]$$

where

$$P(w) = 1 - \exp(-\beta w) + \frac{\mu}{\epsilon} \exp(-\beta w)$$

The `theta` equation is built in the following way. The details for this equation are fairly involved, see J. A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of `theta` on the circle.

```
>>> tau_theta = 3e-3
>>> mu = 1e3
>>> gamma = 1e3
>>> thetaSmallValue = 1e-6
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> beta_theta = 1e5
>>> expo = epsilon * beta_theta * theta.grad.mag
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> Pfunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
```

```
>>> gradMagTheta = theta.faceGrad.mag
>>> eps = 1. / gamma / 10.
>>> gradMagTheta += (gradMagTheta < eps) * eps
>>> IGamma = (gradMagTheta > 1. / gamma) * (1 / gradMagTheta - gamma) + gamma
>>> v_theta = phase.arithmeticFaceValue * (s * IGamma + epsilon**2)
>>> D_theta = phase.arithmeticFaceValue**2 * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. `theta` [faceGradNoMod](#) evaluates the gradient without modular arithmetic.

```
>>> thetaEq = (TransientTerm(coeff=tau_theta * phaseMod**2 * Pfunc, var=theta)
...           == DiffusionTerm(coeff=D_theta, var=theta)
...           + PowerLawConvectionTerm(coeff=v_theta * (theta.faceGrad - theta.
↪faceGradNoMod), var=phase))
```

We seed a circular solidified region in the center

```
>>> x, y = mesh.cellCenters
>>> numSeeds = 10
>>> numerix.random.seed(12345)
>>> for Cx, Cy, orientation in numerix.random.random([numSeeds, 3]):
...     radius = dx * 5.
...     seed = ((x - Cx * nx * dx)**2 + (y - Cy * ny * dy)**2) < radius**2
...     phase[seed] = 1.
...     theta[seed] = numerix.pi * (2 * orientation - 1)
```

and quench the entire simulation domain below the melting point

```
>>> dT.setValue(-0.5)
```

In a real solidification process, dendritic branching is induced by small thermal fluctuations along an otherwise smooth surface, but the granularity of the `Mesh` is enough “noise” in this case, so we don’t need to explicitly introduce randomness, the way we did in the Cahn-Hilliard problem.

FiPy’s viewers are utilitarian, striving to let the user see *something*, regardless of their operating system or installed packages, so you the default color scheme of grain orientation won’t be very informative “out of the box”. Because

all of Python is accessible and FiPy is object oriented, it is not hard to adapt one of the existing viewers to create a specialized display:

```
>>> from builtins import zip
>>> if __name__ == "__main__":
...     try:
...         class OrientationViewer(Matplotlib2DGridViewer):
...             def __init__(self, phase, orientation, title=None, limits={},
↪ **kwlimits):
...                 self.phase = phase
...                 Matplotlib2DGridViewer.__init__(self, vars=(orientation,),
↪ title=title,
...                 limits=limits, colorbar=None,
↪ **kwlimits)
...
...                 # make room for non-existent colorbar
...                 # stolen from matplotlib.colorbar.make_axes
...                 # https://github.com/matplotlib/matplotlib/blob
...                 # /ec1cd2567521c105a451ce15e06de10715f8b54d/lib
...                 # /matplotlib.colorbar.py#L838
...                 fraction = 0.15
...                 pb = self.axes.get_position(original=True).frozen()
...                 pad = 0.05
...                 x1 = 1.0-fraction
...                 pb1, pbx, pbcx = pb.splitx(x1-pad, x1)
...                 panchor = (1.0, 0.5)
...                 self.axes.set_position(pb1)
...                 self.axes.set_anchor(panchor)
...
...                 # make the gnomon
...                 fig = self.axes.get_figure()
...                 self.gnomon = fig.add_axes([0.85, 0.425, 0.15, 0.15], polar=True)
...                 self.gnomon.set_thetagrids([180, 270, 0, 90],
...                 [r"$\pm\pi$", r"$-\frac{\pi}{2}$", "$0$
↪ ", r"$+\frac{\pi}{2}$"],
...                 frac=1.3)
...                 self.gnomon.set_theta_zero_location("N")
...                 self.gnomon.set_theta_direction(-1)
...                 self.gnomon.set_rgrids([1.], [""])
...                 N = 100
...                 theta = numerix.arange(-numerix.pi, numerix.pi, 2 * numerix.pi /
↪ N)
...                 radii = numerix.ones((N,))
...                 bars = self.gnomon.bar(theta, radii, width=2 * numerix.pi / N,
↪ bottom=0.0)
...                 colors = self._orientation_and_phase_to_rgb(orientation=numerix.
↪ array([theta]), phase=1.)
...                 for c, t, bar in zip(colors[0], theta, bars):
...                     bar.set_facecolor(c)
...                     bar.set_edgecolor(c)
...
...                 def _reshape(self, var):
...                     '''return values of var in an 2D array'''
...                     return numerix.reshape(numerix.array(var),
...                     var.mesh.shape[:-1])[:-1]
...
...                 @staticmethod
...                 def _orientation_and_phase_to_rgb(orientation, phase):
```

(continues on next page)

(continued from previous page)

```

...         from matplotlib import colors
...
...         hsv = numerix.empty(orientation.shape + (3,))
...         hsv[..., 0] = (orientation / numerix.pi + 1) / 2.
...         hsv[..., 1] = 1.
...         hsv[..., 2] = phase
...
...         return colors.hsv_to_rgb(hsv)
...
...     @property
...     def _data(self):
...         '''convert phase and orientation to rgb image array
...
...         orientation (-pi, pi) -> hue (0, 1)
...         phase (0, 1) -> value (0, 1)
...         '''
...         orientation = self._reshape(self.vars[0])
...         phase = self._reshape(self.phase)
...
...         return self._orientation_and_phase_to_rgb(orientation, phase)
...
...     def _plot(self):
...         self.image.set_data(self._data)
...
...     from matplotlib import pyplot
...     pyplot.ion()
...     w, h = pyplot.figaspect(1.)
...     fig = pyplot.figure(figsize=(2*w, h))
...     timer = fig.text(0.1, 0.9, "t = %.3f" % 0, fontsize=18)
...
...     viewer = MultiViewer(viewers=(MatplotlibViewer(vars=dT,
...                                                    cmap=pyplot.cm.hot,
...                                                    datamin=-0.5,
...                                                    datamax=0.5,
...                                                    axes=fig.add_subplot(121)),
...                                OrientationViewer(phase=phase,
...                                                    orientation=theta,
...                                                    title=theta.name,
...                                                    axes=fig.add_
...->subplot(122))))
...     except ImportError:
...         viewer = MultiViewer(viewers=(Viewer(vars=dT,
...                                              datamin=-0.5,
...                                              datamax=0.5),
...                                       Viewer(vars=phase,
...                                              datamin=0.,
...                                              datamax=1.),
...                                       Viewer(vars=theta,
...                                              datamin=-numerix.pi,
...                                              datamax=numerix.pi)))
...
...     viewer.plot()

```

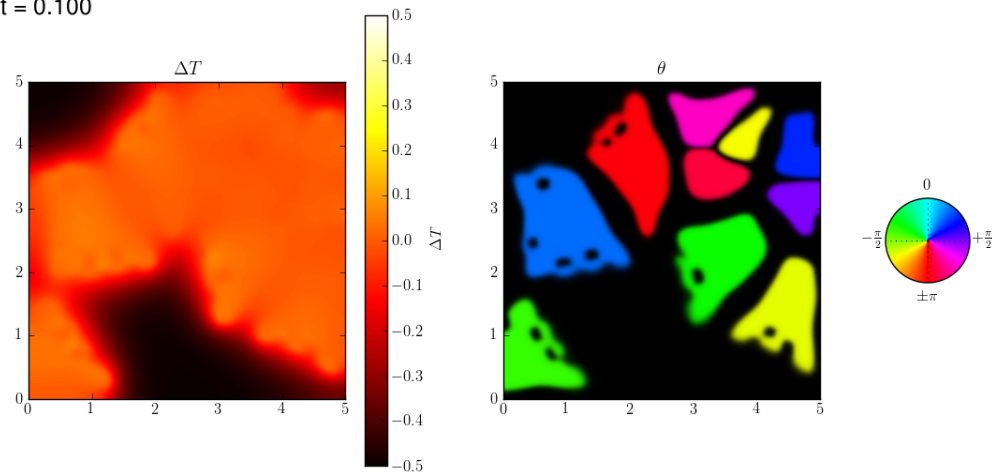
and iterate the solution in time, plotting as we go,

```
>>> eq = thetaEq & phaseEq & heatEq
```

```
>>> if __name__ == "__main__":
...     total_time = 2.
...     else:
...         total_time = dt * 10
>>> elapsed = 0.
>>> save_interval = 0.002
>>> save_at = save_interval
```

```
>>> while elapsed < total_time:
...     if elapsed > 0.3:
...         q.value = 100
...     phase.updateOld()
...     dT.updateOld()
...     theta.updateOld()
...     eq.solve(dt=dt)
...     elapsed += dt
...     if __name__ == "__main__" and elapsed >= save_at:
...         timer.set_text("t = %.3f" % elapsed)
...         viewer.plot()
...         save_at += save_interval
```

t = 0.100



The non-uniform temperature results from the release of latent heat at the solidifying interface. The dendrite arms grow fastest where the temperature gradient is steepest.

Chapter 22

Level Set Examples

<code>examples.levelSet.distanceFunction.mesh1D</code>	Create a level set variable in one dimension.
<code>examples.levelSet.distanceFunction.circle</code>	Solve the level set equation in two dimensions for a circle.
<code>examples.levelSet.advection.mesh1D</code>	Solve the distance function equation in one dimension and then advect it.
<code>examples.levelSet.advection.circle</code>	Solve a circular distance function equation and then advect it.

22.1 examples.levelSet.distanceFunction.mesh1D

Create a level set variable in one dimension.

The level set variable calculates its value over the domain to be the distance from the zero level set. This can be represented succinctly in the following equation with a boundary condition at the zero level set such that,

$$\frac{\partial \phi}{\partial x} = 1$$

with the boundary condition, $\phi = 0$ at $x = L/2$.

The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> from fipy import CellVariable, Grid1D, DistanceVariable, TransientTerm, \
↳ FirstOrderAdvectionTerm, AdvectionTerm, Viewer
>>> from fipy.tools import numerix, serialComm
```

```
>>> dx = 0.5
>>> nx = 10
```

Construct the mesh.

```
>>> mesh = Grid1D(dx=dx, nx=nx, communicator=serialComm)
```

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(name='level set variable',
...                         mesh=mesh,
...                         value=-1.,
...                         hasOld=1)
>>> x = mesh.cellCenters[0]
>>> var.setValue(1, where=x > dx * nx / 2)
```

Once the initial positive and negative regions have been initialized the `calcDistanceFunction()` method can be used to recalculate `var` as a distance function from the zero level set.

```
>>> var.calcDistanceFunction()
```

The problem can then be solved by executing the `solve()` method of the equation.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-5., datamax=5.)
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> print(numerix.allclose(var, x - dx * nx / 2))
1
```

22.2 examples.levelSet.distanceFunction.circle

Solve the level set equation in two dimensions for a circle.

The 2D level set equation can be written,

$$|\nabla\phi| = 1$$

and the boundary condition for a circle is given by, $\phi = 0$ at $(x - L/2)^2 + (y - L/2)^2 = (L/4)^2$.

The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> from fipy import CellVariable, Grid2D, DistanceVariable, TransientTerm,
↳ FirstOrderAdvectionTerm, AdvectionTerm, Viewer
>>> from fipy.tools import numerix, serialComm
```

```
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = nx * dx
>>> Ly = ny * dy
```

Construct the mesh.

```
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny, communicator=serialComm)
```

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(name='level set variable',
...                         mesh=mesh,
...                         value=-1.,
...                         hasOld=1)
```

```
>>> x, y = mesh.cellCenters
>>> var.setValue(1, where=(x - Lx / 2. )**2 + (y - Ly / 2. )**2 < (Lx / 4. )**2)
```

```
>>> var.calcDistanceFunction(order=1)
```

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-5., datamax=5.)
...     viewer.plot()
```

The result can be tested with the following commands.

```
>>> dY = dy / 2.
>>> dX = dx / 2.
>>> mm = min (dX, dY)
>>> m1 = dY * dX / numerix.sqrt(dY**2 + dX**2)
>>> def evalCell(phix, phiy, dx, dy):
...     aa = dy**2 + dx**2
...     bb = -2 * ( phix * dy**2 + phiy * dx**2)
...     cc = dy**2 * phix**2 + dx**2 * phiy**2 - dx**2 * dy**2
...     sqr = numerix.sqrt(bb**2 - 4. * aa * cc)
...     return ((-bb - sqr) / 2. / aa, (-bb + sqr) / 2. / aa)
>>> v1 = evalCell(-dY, -m1, dx, dy)[0]
>>> v2 = evalCell(-m1, -dX, dx, dy)[0]
>>> v3 = evalCell(m1, m1, dx, dy)[1]
>>> v4 = evalCell(v3, dY, dx, dy)[1]
>>> v5 = evalCell(dX, v3, dx, dy)[1]
>>> MASK = -1000.
>>> trialValues = CellVariable(mesh=mesh, value= \
...     numerix.array((
...     MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK,
...     MASK, MASK, MASK, MASK, -3*dY, -3*dY, -3*dY, MASK, MASK, MASK, MASK,
...     MASK, MASK, MASK, v1, -dY, -dY, -dY, v1, MASK, MASK, MASK,
...     MASK, MASK, v2, -m1, m1, dY, m1, -m1, v2, MASK, MASK,
...     MASK, -dX*3, -dX, m1, v3, v4, v3, m1, -dX, -dX*3, MASK,
...     MASK, -dX*3, -dX, dX, v5, MASK, v5, dX, -dX, -dX*3, MASK,
...     MASK, -dX*3, -dX, m1, v3, v4, v3, m1, -dX, -dX*3, MASK,
...     MASK, MASK, v2, -m1, m1, dY, m1, -m1, v2, MASK, MASK,
...     MASK, MASK, MASK, v1, -dY, -dY, -dY, v1, MASK, MASK, MASK,
...     MASK, MASK, MASK, MASK, -3*dY, -3*dY, -3*dY, MASK, MASK, MASK, MASK,
...     MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK, MASK), 'd'))
```

```
>>> var[numerix.array(trialValues == MASK)] = MASK
>>> print(numerix.allclose(var, trialValues))
True
```

22.3 examples.levelSet.advection.mesh1D

Solve the distance function equation in one dimension and then advect it.

This example first solves the distance function equation in one dimension:

$$|\nabla\phi| = 1$$

with $\phi = 0$ at $x = L/5$.

The variable is then advected with,

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0$$

The scheme used in the *FirstOrderAdvectionTerm* preserves the *var* as a distance function.

The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> from fipy import CellVariable, Grid1D, DistanceVariable, TransientTerm, \
↳ FirstOrderAdvectionTerm, AdvectionTerm, Viewer
>>> from fipy.tools import numerix, serialComm
```

```
>>> velocity = 1.
>>> dx = 1.
>>> nx = 10
>>> timeStepDuration = 1.
>>> steps = 2
>>> L = nx * dx
>>> interfacePosition = L / 5.
```

Construct the mesh.

```
>>> mesh = Grid1D(dx=dx, nx=nx, communicator=serialComm)
```

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(name='level set variable',
...                         mesh=mesh,
...                         value=-1.,
...                         hasOld=1)
>>> var.setValue(1., where=mesh.cellCenters[0] > interfacePosition)
>>> var.calcDistanceFunction()
```

The *advectionEquation* is constructed.

```
>>> advEqn = TransientTerm() + FirstOrderAdvectionTerm(velocity)
```

The problem can then be solved by executing a series of time steps.

```
>>> from builtins import range
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-10., datamax=10.)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following code:

```
>>> from builtins import range
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = mesh.cellCenters[0]
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = x - interfacePosition - timeStepDuration * steps * velocity
```

(continues on next page)

(continued from previous page)

```
>>> answer = numerix.where(x < distanceTravelled,
...                         x[0] - interfacePosition, answer)
>>> print(var.allclose(answer))
1
```

22.4 examples.levelSet.advection.circle

Solve a circular distance function equation and then advect it.

This example first imposes a circular distance function:

$$\phi(x, y) = \left[\left(x - \frac{L}{2} \right)^2 + \left(y - \frac{L}{2} \right)^2 \right]^{1/2} - \frac{L}{4}$$

The variable is advected with,

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0$$

The scheme used in the `FirstOrderAdvectionTerm` preserves the `var` as a distance function. The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> from fipy import CellVariable, Grid2D, DistanceVariable, TransientTerm,
↳ FirstOrderAdvectionTerm, AdvectionTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 1.
>>> N = 25
>>> velocity = 1.
>>> cfl = 0.1
>>> velocity = 1.
>>> distanceToTravel = L / 10.
>>> radius = L / 4.
>>> dL = L / N
>>> timeStepDuration = cfl * dL / velocity
>>> steps = int(distanceToTravel / dL / cfl)
```

Construct the mesh.

```
>>> mesh = Grid2D(dx=dL, dy=dL, nx=N, ny=N)
```

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(
...     name = 'level set variable',
...     mesh = mesh,
...     value = 1.,
...     hasOld = 1)
```

Initialize the *distanceVariable* to be a circular distance function.

```
>>> x, y = mesh.cellCenters
>>> initialArray = numerix.sqrt((x - L / 2.)**2 + (y - L / 2.)**2) - radius
>>> var.setValue(initialArray)
```

The advection equation is constructed.

```
>>> advEqn = TransientTerm() + FirstOrderAdvectionTerm(velocity)
```

The problem can then be solved by executing a series of time steps.

```
>>> from builtins import range
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-radius, datamax=radius)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following commands.

```
>>> from builtins import range
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = numerix.array(mesh.cellCenters[0])
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = initialArray - distanceTravelled
>>> answer = numerix.where(answer < 0., -1001., answer)
>>> solution = numerix.where(answer < 0., -1001., numerix.array(var))
>>> numerix.allclose(answer, solution, atol=4.7e-3)
1
```

If the advection equation is built with the *AdvectionTerm()* the result is more accurate,

```
>>> var.setValue(initialArray)
>>> advEqn = TransientTerm() + AdvectionTerm(velocity)
>>> from builtins import range
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> solution = numerix.where(answer < 0., -1001., numerix.array(var))
>>> numerix.allclose(answer, solution, atol=1.02e-3)
1
```

Chapter 23

Cahn-Hilliard Examples

<code>examples.cahnHilliard.mesh2DCoupled</code>	Solve the Cahn-Hilliard problem in two dimensions.
<code>examples.cahnHilliard.sphere</code>	Solves the Cahn-Hilliard problem on the surface of a sphere.

23.1 `examples.cahnHilliard.mesh2DCoupled`

Solve the Cahn-Hilliard problem in two dimensions.

Warning: This formulation has [serious performance problems](#) and is **not automatically tested**. Specifically, for non-trivial mesh sizes, [PySparse](#) requires enormous amounts of memory, [Trilinos](#) cannot solve the coupled form, and [PETSc](#) cannot solve the vector form.

The spinodal decomposition phenomenon is a spontaneous separation of an initially homogeneous mixture into two distinct regions of different properties (spin-up/spin-down, component A/component B). It is a “barrierless” phase separation process, such that under the right thermodynamic conditions, any fluctuation, no matter how small, will tend to grow. This is in contrast to nucleation, where a fluctuation must exceed some critical magnitude before it will survive and grow. Spinodal decomposition can be described by the “Cahn-Hilliard” equation (also known as “conserved Ginsberg-Landau” or “model B” of Hohenberg & Halperin)

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right).$$

where ϕ is a conserved order parameter, possibly representing alloy composition or spin. The double-well free energy function $f = (a^2/2)\phi^2(1 - \phi)^2$ penalizes states with intermediate values of ϕ between 0 and 1. The gradient energy term $\epsilon^2 \nabla^2 \phi$, on the other hand, penalizes sharp changes of ϕ . These two competing effects result in the segregation of ϕ into domains of 0 and 1, separated by abrupt, but smooth, transitions. The parameters a and ϵ determine the relative weighting of the two effects and D is a rate constant.

We can simulate this process in [FiPy](#) with a simple script:

```
>>> from fipy import CellVariable, Grid2D, GaussianNoiseVariable, DiffusionTerm, \
↳ TransientTerm, ImplicitSourceTerm, Viewer
>>> from fipy.tools import numerix
```

(Note that all of the functionality of NumPy is imported along with *FiPy*, although much is augmented for *FiPy*'s needs.)

```
>>> if __name__ == "__main__":
...     nx = ny = 20
... else:
...     nx = ny = 10
>>> mesh = Grid2D(nx=nx, ny=ny, dx=0.25, dy=0.25)
>>> phi = CellVariable(name=r"$\phi$", mesh=mesh)
>>> psi = CellVariable(name=r"$\psi$", mesh=mesh)
```

We start the problem with random fluctuations about $\phi = 1/2$

```
>>> noise = GaussianNoiseVariable(mesh=mesh,
...                               mean=0.5,
...                               variance=0.01).value
```

```
>>> phi[:] = noise
```

FiPy doesn't plot or output anything unless you tell it to:

```
>>> if __name__ == "__main__":
...     viewer = Viewer(vars=(phi,), datamin=0., datamax=1.)
```

We factor the Cahn-Hilliard equation into two 2nd-order PDEs and place them in canonical form for *FiPy* to solve them as a coupled set of equations.

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \psi$$

$$\psi = \left(\frac{\partial f}{\partial \phi} - \frac{\partial^2 f}{\partial \phi^2} \phi \right)_{\text{old}} + \frac{\partial^2 f}{\partial \phi^2} \phi - \epsilon^2 \nabla^2 \phi$$

The source term in ψ , $\frac{\partial f}{\partial \phi}$, is expressed in linearized form after Taylor expansion at $\phi = \phi_{\text{old}}$, for the same reasons discussed in [examples.phase.simple](#). We need to perform the partial derivatives

$$\frac{\partial f}{\partial \phi} = (a^2/2)2\phi(1-\phi)(1-2\phi)$$

$$\frac{\partial^2 f}{\partial \phi^2} = (a^2/2)2[1-6\phi(1-\phi)]$$

manually.

```
>>> D = a = epsilon = 1.
>>> dfdphi = a**2 * phi * (1 - phi) * (1 - 2 * phi)
>>> dfdphi_ = a**2 * (1 - phi) * (1 - 2 * phi)
>>> d2fdphi2 = a**2 * (1 - 6 * phi * (1 - phi))
>>> eq1 = (TransientTerm(var=phi) == DiffusionTerm(coeff=D, var=psi))
>>> eq2 = (ImplicitSourceTerm(coeff=1., var=psi)
...       == ImplicitSourceTerm(coeff=d2fdphi2, var=phi) - d2fdphi2 * phi + dfdphi
...       - DiffusionTerm(coeff=epsilon**2, var=phi))
>>> eq3 = (ImplicitSourceTerm(coeff=1., var=psi)
...       == ImplicitSourceTerm(coeff=dfdphi_, var=phi)
...       - DiffusionTerm(coeff=epsilon**2, var=phi))
```

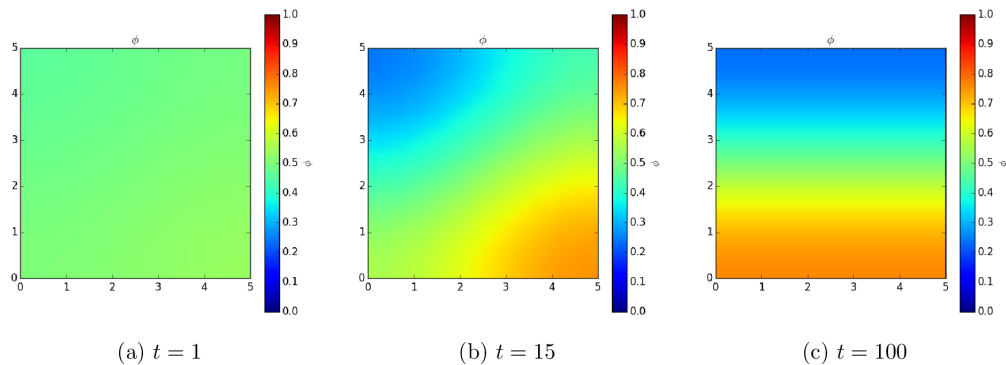
```
>>> eq = eq1 & eq2
```

Because the evolution of a spinodal microstructure slows with time, we use exponentially increasing time steps to keep the simulation “interesting”. The *FiPy* user always has direct control over the evolution of their problem.


```
>>> dexp = -5
>>> elapsed = 0.
>>> if __name__ == "__main__":
...     duration = 100.
... else:
...     duration = .5e-1
```

```
>>> while elapsed < duration:
...     dt = min(100, numerix.exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(dt=dt)
...     if __name__ == "__main__":
...         viewer.plot()
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Coupled equations. Press <return> to proceed...")
```



These equations can also be solved in *FiPy* using a vector equation. The variables ϕ and ψ are now stored in a single variable

```
>>> var = CellVariable(mesh=mesh, elementshape=(2,))
>>> var[0] = noise
```

```
>>> if __name__ == "__main__":
...     viewer = Viewer(name=r"$\phi$", vars=var[0,], datamin=0., datamax=1.)
```

```
>>> D = a = epsilon = 1.
>>> v0 = var[0]
>>> dfdphi = a**2 * v0 * (1 - v0) * (1 - 2 * v0)
>>> d2fdphi_ = a**2 * (1 - v0) * (1 - 2 * v0)
>>> d2fdphi2 = a**2 * (1 - 6 * v0 * (1 - v0))
```

The source terms have to be shaped correctly for a vector. The implicit source coefficient has to have a shape of (2, 2) while the explicit source has a shape (2,)

```
>>> source = (- d2fdphi2 * v0 + dfdphi) * (0, 1)
>>> impCoeff = d2fdphi2 * ((0, 0),
```

(continues on next page)

(continued from previous page)

```
...         (1., 0)) + ((0, 0),
...                     (0, -1.))
```

This is the same equation as the previous definition of *eq*, but now in a vector format.

```
>>> eq = (TransientTerm(((1., 0.),
...                      (0., 0.))) == DiffusionTerm([(0.,          D),
...                                                    (-epsilon**2, 0.)]))
...                                     + ImplicitSourceTerm(impCoeff) + source)
```

```
>>> dexp = -5
>>> elapsed = 0.
```

```
>>> while elapsed < duration:
...     dt = min(100, numerix.exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(var=var, dt=dt)
...     if __name__ == "__main__":
...         viewer.plot()
```

```
>>> print(numerix.allclose(var, (phi, psi)))
True
```

23.2 examples.cahnHilliard.sphere

Solves the Cahn-Hilliard problem on the surface of a sphere.

This phenomenon can occur on vesicles (http://www.youtube.com/watch?v=kDsFP67_ZSE).

```
>>> from fipy import CellVariable, Gmsh2DIn3DSpace, GaussianNoiseVariable, Viewer, _
...     TransientTerm, DiffusionTerm, DefaultSolver
>>> from fipy.tools import numerix
```

The only difference from `examples.cahnHilliard.mesh2D` is the declaration of mesh.

```
>>> mesh = Gmsh2DIn3DSpace('''
...     radius = 5.0;
...     cellSize = 0.3;
...
...     // create inner 1/8 shell
...     Point(1) = {0, 0, 0, cellSize};
...     Point(2) = {-radius, 0, 0, cellSize};
...     Point(3) = {0, radius, 0, cellSize};
...     Point(4) = {0, 0, radius, cellSize};
...     Circle(1) = {2, 1, 3};
...     Circle(2) = {4, 1, 2};
...     Circle(3) = {4, 1, 3};
...     Line Loop(1) = {1, -3, 2} ;
...     Ruled Surface(1) = {1};
...
...     // create remaining 7/8 inner shells
...     t1[] = Rotate {{0,0,1},{0,0,0},Pi/2} {Duplicata{Surface{1}}};
```

(continues on next page)

(continued from previous page)

```
...     t2[] = Rotate ({0,0,1},{0,0,0},Pi) {Duplicata{Surface{1}}};
...     t3[] = Rotate ({0,0,1},{0,0,0},Pi*3/2) {Duplicata{Surface{1}}};
...     t4[] = Rotate ({0,1,0},{0,0,0},-Pi/2) {Duplicata{Surface{1}}};
...     t5[] = Rotate ({0,0,1},{0,0,0},Pi/2) {Duplicata{Surface{t4[0]}}};
...     t6[] = Rotate ({0,0,1},{0,0,0},Pi) {Duplicata{Surface{t4[0]}}};
...     t7[] = Rotate ({0,0,1},{0,0,0},Pi*3/2) {Duplicata{Surface{t4[0]}}};
...
...     // create entire inner and outer shell
...     Surface Loop(100)={1,t1[0],t2[0],t3[0],t7[0],t4[0],t5[0],t6[0]};
...     '', overlap=2).extrude(extrudeFunc=lambda r: 1.1 * r)
>>> phi = CellVariable(name=r"$\phi$", mesh=mesh)
```

We start the problem with random fluctuations about $\phi = 1/2$

```
>>> phi.setValue(GaussianNoiseVariable(mesh=mesh,
...                                     mean=0.5,
...                                     variance=0.01))
```

FiPy doesn't plot or output anything unless you tell it to: If *MayaviClient* is available, we can customize the view with a subclass of *MayaviDaemon*.

```
>>> if __name__ == "__main__":
...     try:
...         viewer = MayaviClient(vars=phi,
...                               datamin=0., datamax=1.,
...                               daemon_file="examples/cahnHilliard/sphereDaemon.py")
...     except:
...         viewer = Viewer(vars=phi,
...                          datamin=0., datamax=1.,
...                          xmin=-2.5, xmax=2.5)
```

For *FiPy*, we need to perform the partial derivative $\partial f / \partial \phi$ manually and then put the equation in the canonical form by decomposing the spatial derivatives so that each *Term* is of a single, even order:

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D a^2 [1 - 6\phi(1 - \phi)] \nabla \phi - \nabla \cdot D \nabla \epsilon^2 \nabla^2 \phi.$$

FiPy would automatically interpolate $D * a^2 * (1 - 6 * \phi * (1 - \phi))$ onto the faces, where the diffusive flux is calculated, but we obtain somewhat more accurate results by performing a linear interpolation from ϕ at cell centers to PHI at face centers. Some problems benefit from non-linear interpolations, such as harmonic or geometric means, and *FiPy* makes it easy to obtain these, too.

```
>>> PHI = phi.arithmeticFaceValue
>>> D = a = epsilon = 1.
>>> eq = (TransientTerm()
...       == DiffusionTerm(coeff=D * a**2 * (1 - 6 * PHI * (1 - PHI)))
...       - DiffusionTerm(coeff=(D, epsilon**2)))
```

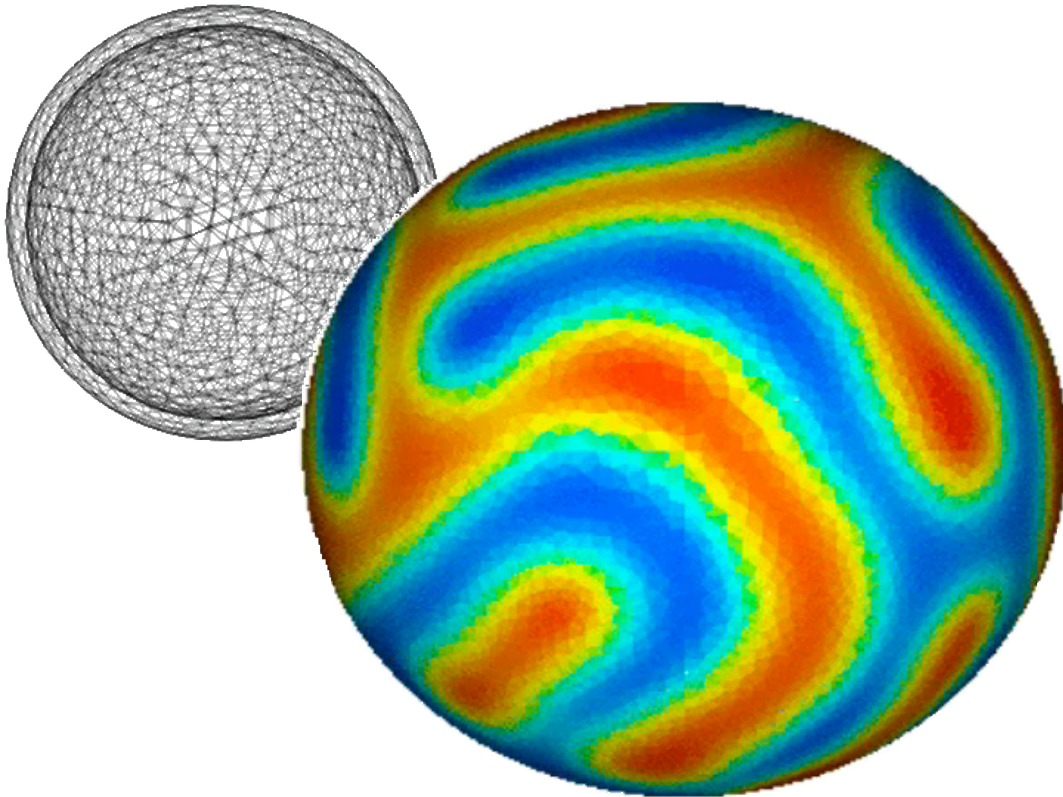
Because the evolution of a spinodal microstructure slows with time, we use exponentially increasing time steps to keep the simulation “interesting”. The *FiPy* user always has direct control over the evolution of their problem.

```
>>> dexp = -5
>>> elapsed = 0.
>>> if __name__ == "__main__":
...     duration = 1000.
... else:
```

(continues on next page)

(continued from previous page)

```
...     duration = 1e-2
>>> while elapsed < duration:
...     dt = min(100, numerix.exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(phi, dt=dt, solver=DefaultSolver(precon=None))
...     if __name__ == "__main__":
...         viewer.plot()
```



Fluid Flow Examples

examples.flow.stokesCavity
Solve the Navier-Stokes equation in the viscous limit.

24.1 examples.flow.stokesCavity

Solve the Navier-Stokes equation in the viscous limit.

Many thanks to Benny Malengier <bm@cage.ugent.be> for reworking this example and actually making it work correctly... see #209

This example is an implementation of a rudimentary Stokes solver on a collocated grid. It solves the Navier-Stokes equation in the viscous limit,

$$\nabla \cdot (\mu \nabla \vec{u}) = \nabla p$$

and the continuity equation,

$$\nabla \cdot \vec{u} = 0$$

where \vec{u} is the fluid velocity, p is the pressure and μ is the viscosity. The domain in this example is a square cavity of unit dimensions with a moving lid of unit speed. This example uses the SIMPLE algorithm with Rhie-Chow interpolation for collocated grids to solve the pressure-momentum coupling. Some of the details of the algorithm will be highlighted below but a good reference for this material is Ferziger and Peric [32] and Rossow [33]. The solution has a high degree of error close to the corners of the domain for the pressure but does a reasonable job of predicting the velocities away from the boundaries. A number of aspects of *FiPy* need to be improved to have a first class flow solver. These include, higher order spatial diffusion terms, proper wall boundary conditions, improved mass flux evaluation and extrapolation of cell values to the boundaries using gradients.

In the table below a comparison is made with the *Dolfyn* open source code on a 100 by 100 grid. The table shows the frequency of values that fall within the given error confidence bands. *Dolfyn* has the added features described above. When these features are switched off the results of *Dolfyn* and *FiPy* are identical.

% frequency of cells	x-velocity error (%)	y-velocity error (%)	pressure error (%)
90	< 0.1	< 0.1	< 5
5	0.1 to 0.6	0.1 to 0.3	5 to 11
4	0.6 to 7	0.3 to 4	11 to 35
1	7 to 96	4 to 80	35 to 179
0	> 96	> 80	> 179

To start, some parameters are declared.

```
>>> from fipy import CellVariable, FaceVariable, Grid2D, DiffusionTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 1.0
>>> N = 50
>>> dL = L / N
>>> viscosity = 1
>>> U = 1.
>>> #0.8 for pressure and 0.5 for velocity are typical relaxation values for SIMPLE
>>> pressureRelaxation = 0.8
>>> velocityRelaxation = 0.5
>>> if __name__ == '__main__':
...     sweeps = 300
... else:
...     sweeps = 5
```

Build the mesh.

```
>>> mesh = Grid2D(nx=N, ny=N, dx=dL, dy=dL)
```

Declare the variables.

```
>>> pressure = CellVariable(mesh=mesh, name='pressure')
>>> pressureCorrection = CellVariable(mesh=mesh)
>>> xVelocity = CellVariable(mesh=mesh, name='X velocity')
>>> yVelocity = CellVariable(mesh=mesh, name='Y velocity')
```

The velocity is required as a rank-1 *FaceVariable* for calculating the mass flux. This is required by the Rhie-Chow correction to avoid pressure/velocity decoupling.

```
>>> velocity = FaceVariable(mesh=mesh, rank=1)
```

Build the Stokes equations in the cell centers.

```
>>> xVelocityEq = DiffusionTerm(coeff=viscosity) - pressure.grad.dot([1., 0.])
>>> yVelocityEq = DiffusionTerm(coeff=viscosity) - pressure.grad.dot([0., 1.])
```

In this example the SIMPLE algorithm is used to couple the pressure and momentum equations. Let us assume we have solved the discretized momentum equations using a guessed pressure field p^* to obtain a velocity field \vec{u}^* . That is \vec{u}^* is found from

$$a_P \vec{u}_P^* = \sum_f a_A \vec{u}_A^* - V_P (\nabla p^*)_P$$

We would like to somehow correct these initial fields to satisfy both the discretized momentum and continuity equations. We now try to correct these initial fields with a correction such that $\vec{u} = \vec{u}^* + \vec{u}'$ and $p = p^* + p'$, where \vec{u} and

p now satisfy the momentum and continuity equations. Substituting the exact solution into the equations we obtain,

$$\nabla \cdot (\mu \nabla \vec{u}') = \nabla p'$$

and

$$\nabla \cdot \vec{u}^* + \nabla \cdot \vec{u}' = 0$$

We now use the discretized form of the equations to write the velocity correction in terms of the pressure correction. The discretized form of the above equation results in an equation for $p = p'$,

$$a_P \vec{u}'_P = \sum_f a_A \vec{u}'_A - V_P (\nabla p')_P$$

where notation from [Linear Equations](#) is used. The SIMPLE algorithm drops the second term in the above equation to leave,

$$\vec{u}'_P = - \frac{V_P (\nabla p')_P}{a_P}$$

By substituting the above expression into the continuity equations we obtain the pressure correction equation,

$$\nabla \frac{V_P}{a_P} \cdot \nabla p' = \nabla \cdot \vec{u}^*$$

In the discretized version of the above equation V_P/a_P is approximated at the face by $A_f d_{AP}/(a_P)_f$. In *FiPy* the pressure correction equation can be written as,

```
>>> ap = CellVariable(mesh=mesh, value=1.)
>>> coeff = 1./ ap.arithmeticFaceValue*mesh._faceAreas * mesh._cellDistances
>>> pressureCorrectionEq = DiffusionTerm(coeff=coeff) - velocity.divergence
```

Above would work good on a staggered grid, however, on a colocated grid as *FiPy* uses, the term `velocity.divergence` will cause oscillations in the pressure solution as velocity is a face variable. We can apply the Rhie-Chow correction terms for this. In this an intermediate velocity term \vec{u}^\diamond is considered which does not contain the pressure corrections:

$$\vec{u}_P^\diamond = \vec{u}_P^* + \frac{V_P}{a_P} (\nabla p^*)_P = \sum_f \frac{a_A}{a_P} \vec{u}_A^*$$

This velocity is interpolated at the edges, after which the pressure correction term is added again, but now considered at the edge:

$$\vec{u}_f = \frac{1}{2} (\vec{u}_L^\diamond + \vec{u}_R^\diamond) - \left(\frac{V}{a_P} \right)_{\text{avg L,R}} (\nabla p_f^*)$$

where $\left(\frac{V}{a_P} \right)_{\text{avg L,R}}$ is assumed a good approximation at the edge. Here L and R denote the two cells adjacent to the face. Expanding the not calculated terms we arrive at

$$\vec{u}_f = \frac{1}{2} (\vec{u}_L^* + \vec{u}_R^*) + \frac{1}{2} \left(\frac{V}{a_P} \right)_{\text{avg L,R}} (\nabla p_L^* + \nabla p_R^*) - \left(\frac{V}{a_P} \right)_{\text{avg L,R}} (\nabla p_f^*)$$

where we have replaced the coefficients of the cell pressure gradients by an averaged value over the edge. This formula has the consequence that the velocity on a face depends not only on the pressure of the adjacent cells, but also on the cells further away, which removes the unphysical pressure oscillations. We start by introducing needed terms

```
>>> from fipy.variables.faceGradVariable import _FaceGradVariable
>>> volume = CellVariable(mesh=mesh, value=mesh.cellVolumes, name='Volume')
>>> contrvolume=volume.arithmeticFaceValue
```

And set up the velocity with this formula in the SIMPLE loop. Now, set up the no-slip boundary conditions

```
>>> xVelocity.constrain(0., mesh.facesRight | mesh.facesLeft | mesh.facesBottom)
>>> xVelocity.constrain(U, mesh.facesTop)
>>> yVelocity.constrain(0., mesh.exteriorFaces)
>>> X, Y = mesh.faceCenters
>>> pressureCorrection.constrain(0., mesh.facesLeft & (Y < dL))
```

Set up the viewers,

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(pressure, xVelocity, yVelocity, velocity),
...                        xmin=0., xmax=1., ymin=0., ymax=1., colorbar=True)
```

Below, we iterate for a set number of sweeps. We use the `sweep()` method instead of `solve()` because we require the residual for output. We also use the `cacheMatrix()`, `matrix`, `cacheRHSvector()` and `RHSvector` because both the matrix and RHS vector are required by the SIMPLE algorithm. Additionally, the `sweep()` method is passed an underRelaxation factor to relax the solution. This argument cannot be passed to `solve()`.

```
>>> from builtins import range
>>> for sweep in range(sweeps):
...
...     ## solve the Stokes equations to get starred values
...     xVelocityEq.cacheMatrix()
...     xres = xVelocityEq.sweep(var=xVelocity,
...                               underRelaxation=velocityRelaxation)
...     xmat = xVelocityEq.matrix
...
...     yres = yVelocityEq.sweep(var=yVelocity,
...                               underRelaxation=velocityRelaxation)
...
...     ## update the ap coefficient from the matrix diagonal
...     ap[:] = -numerix.asarray(xmat.takeDiagonal())
...
...     ## update the face velocities based on starred values with the
...     ## Rhie-Chow correction.
...     ## cell pressure gradient
...     presgrad = pressure.grad
...     ## face pressure gradient
...     facepresgrad = _FaceGradVariable(pressure)
...
...     velocity[0] = xVelocity.arithmeticFaceValue \
...         + contrvolume / ap.arithmeticFaceValue * \
...             (presgrad[0].arithmeticFaceValue-facepresgrad[0])
...     velocity[1] = yVelocity.arithmeticFaceValue \
...         + contrvolume / ap.arithmeticFaceValue * \
...             (presgrad[1].arithmeticFaceValue-facepresgrad[1])
...     velocity[... , mesh.exteriorFaces.value] = 0.
...     velocity[0, mesh.facesTop.value] = U
...
...     ## solve the pressure correction equation
...     pressureCorrectionEq.cacheRHSvector()
...     ## left bottom point must remain at pressure 0, so no correction
```

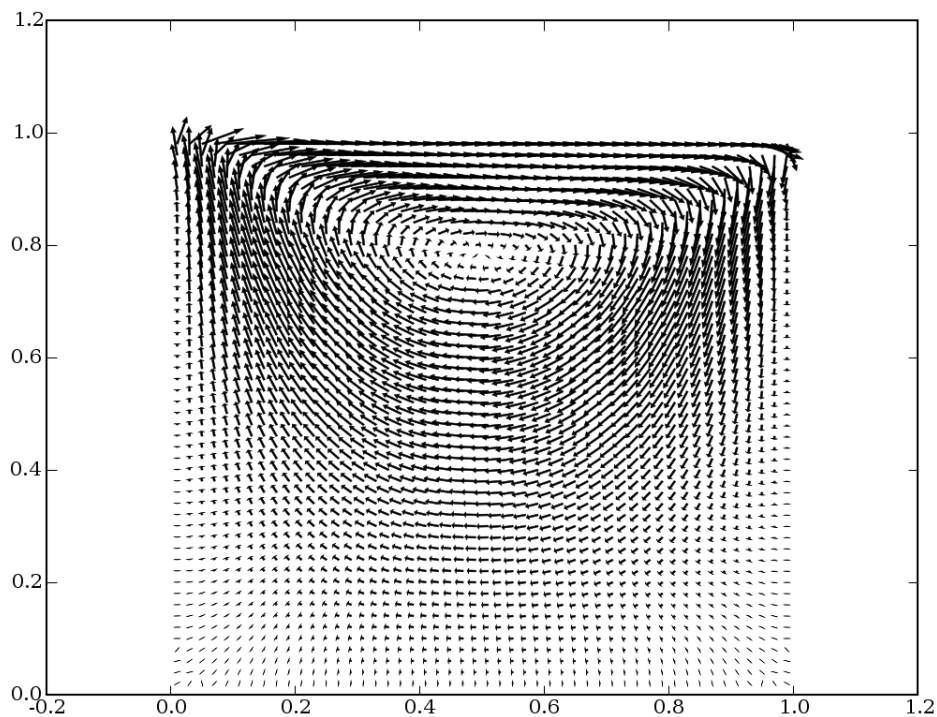
(continues on next page)

(continued from previous page)

```

...     pres = pressureCorrectionEq.sweep(var=pressureCorrection)
...     rhs = pressureCorrectionEq.RHSvector
...
...     ## update the pressure using the corrected value
...     pressure.setValue(pressure + pressureRelaxation * pressureCorrection )
...     ## update the velocity using the corrected pressure
...     xVelocity.setValue(xVelocity - pressureCorrection.grad[0] / \
...                         ap * mesh.cellVolumes)
...     yVelocity.setValue(yVelocity - pressureCorrection.grad[1] / \
...                         ap * mesh.cellVolumes)
...
...     if __name__ == '__main__':
...         if sweep%10 == 0:
...             print('sweep:', sweep, ', x residual:', xres, \
...                   ', y residual', yres, \
...                   ', p residual:', pres, \
...                   ', continuity:', max(abs(rhs)))
...
...         viewer.plot()

```



Test values in the last cell.

```

>>> print(numerix.allclose(pressure.globalValue[...,-1], 162.790867927))
1
>>> print(numerix.allclose(xVelocity.globalValue[...,-1], 0.265072740929))
1
>>> print(numerix.allclose(yVelocity.globalValue[...,-1], -0.150290488304))

```

(continues on next page)

(continued from previous page)

1

Chapter 25

Reactive Wetting Examples

`examples.reactiveWetting.
liquidVapor1D`

Solve a single-component, liquid-vapor, van der Waals system.

25.1 `examples.reactiveWetting.liquidVapor1D`

Solve a single-component, liquid-vapor, van der Waals system.

This example solves a single-component, liquid-vapor, van der Waals system as described by Wheeler *et al.* [7]. The free energy for this system takes the form,

$$f = -\frac{e\rho^2}{m^2} + \frac{RT}{m} \left(\ln \frac{\rho}{m - \bar{v}\rho} \right) \quad (25.1)$$

where ρ is the density. This free energy supports a two phase equilibrium with densities given by ρ^l and ρ^v in the liquid and vapor phases, respectively. The densities are determined by solving the following system of equations,

$$P(\rho^l) = P(\rho^v) \quad (25.2)$$

and

$$\mu(\rho^l) = \mu(\rho^v) \quad (25.3)$$

where μ is the chemical potential,

$$\mu = \frac{\partial f}{\partial \rho} \quad (25.4)$$

and P is the pressure,

$$P = \rho\mu - f \quad (25.5)$$

One choice of thermodynamic parameters that yields a relatively physical two phase system is

```
>>> molarWeight = 0.118
>>> ee = -0.455971
>>> gasConstant = 8.314
>>> temperature = 650.
>>> vbar = 1.3e-05
```

with equilibrium density values of

```
>>> liquidDensity = 7354.3402662299995
>>> vaporDensity = 82.855803327810008
```

The equilibrium densities are verified by substitution into Eqs. (25.2) and (25.3). Firstly, Eqs. (25.1), (25.4) and (25.5) are defined as python functions,

```
>>> from fipy import CellVariable, Grid1D, TransientTerm, VanLeerConvectionTerm, \
↳ DiffusionTerm, ImplicitSourceTerm, ConvectionTerm, CentralDifferenceConvectionTerm, \
↳ Viewer
>>> from fipy.tools import numerix
```

```
>>> def f(rho):
...     return ee * rho**2 / molarWeight**2 + gasConstant * temperature * rho / \
↳ molarWeight * \
...     numerix.log(rho / (molarWeight - vbar * rho))
```

```
>>> def mu(rho):
...     return 2 * ee * rho / molarWeight**2 + gasConstant * temperature / \
↳ molarWeight * \
...     (numerix.log(rho / (molarWeight - vbar * rho)) + molarWeight / \
↳ (molarWeight - vbar * rho))
```

```
>>> def P(rho):
...     return rho * mu(rho) - f(rho)
```

The equilibrium densities values are verified with

```
>>> print(numerix.allclose(mu(liquidDensity), mu(vaporDensity)))
True
```

and

```
>>> print(numerix.allclose(P(liquidDensity), P(vaporDensity)))
True
```

In order to derive governing equations, the free energy functional is defined.

$$F = \int \left[f + \frac{\epsilon T}{2} (\partial_j \rho)^2 \right] dV$$

Using standard dissipation laws, we write the governing equations for mass and momentum conservation,

$$\frac{\partial \rho}{\partial t} + \partial_j (\rho u_j) = 0 \quad (25.6)$$

and

$$\frac{\partial (\rho u_i)}{\partial t} + \partial_j (\rho u_i u_j) = \partial_j (\nu [\partial_j u_i + \partial_i u_j]) - \rho \partial_i \mu^{NC} \quad (25.7)$$

where the non-classical potential, μ^{NC} , is given by,

$$\mu^{NC} = \frac{\delta F}{\delta \rho} = \mu - \epsilon T \partial_j^2 \rho \quad (25.8)$$

As usual, to proceed, we define a mesh


```
>>> viscosity = 1e-3
>>> ConvectionTerm = CentralDifferenceConvectionTerm
>>> momentumEqn = TransientTerm(coeff=density, var=velocity) \
...     + ConvectionTerm(coeff=[[1]] * density.faceValue * velocity.
...     ↪faceValue, var=velocity) \
...     == DiffusionTerm(coeff=2 * viscosity, var=velocity) \
...     - ConvectionTerm(coeff=density.faceValue * [[1]], var=potentialNC) \
...     + ImplicitSourceTerm(coeff=density.grad[0], var=potentialNC)
```

The only required boundary condition eliminates flow in or out of the domain.

```
>>> velocity.constrain(0, mesh.exteriorFaces)
```

As previously stated, the μ^{NC} variable will be solved implicitly. To do this the Eq. (25.8) is linearized in ρ such that

$$\mu^{NC} = \mu^* + \left(\frac{\partial \mu}{\partial \rho} \right)^* (\rho - \rho^*) - \epsilon T \partial_j^2 \rho \quad (25.11)$$

The * superscript denotes the current held value. In *FiPy*, $\frac{\partial \mu}{\partial \rho}$ is written as,

```
>>> potentialDerivative = 2 * ee / molarWeight**2 + gasConstant * temperature *
...     ↪molarWeight / density / (molarWeight - vbar * density)**2
```

and μ^* is simply,

```
>>> potential = mu(density)
```

Eq. (25.11) can be scripted as

```
>>> potentialNCEqn = ImplicitSourceTerm(coeff=1, var=potentialNC) \
...     == potential \
...     + ImplicitSourceTerm(coeff=potentialDerivative, var=density) \
...     - potentialDerivative * density \
...     - DiffusionTerm(coeff=epsilon * temperature, var=density)
```

Due to a quirk in *FiPy*, the gradient of μ^{NC} needs to be constrained on the boundary. This is because *ConvectionTerm*'s will automatically assume a zero flux, which is not what we need in this case.

```
>>> potentialNC.faceGrad.constrain(value=[0], where=mesh.exteriorFaces)
```

All three equations are defined and are combined together with

```
>>> coupledEqn = massEqn & momentumEqn & potentialNCEqn
```

The system will be solved as a phase separation problem with an initial density close to the average density, but with some small amplitude noise. Under these circumstances, the final condition should be two separate phases of roughly equal volume. The initial condition for the density is defined by

```
>>> numerix.random.seed(2011)
>>> density[:] = (liquidDensity + vaporDensity) / 2 * \
...     (1 + 0.01 * (2 * numerix.random.random(mesh.numberOfCells) - 1))
```

Viewers are also defined.

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewers = Viewer(density), Viewer(velocity), Viewer(potentialNC)
```

(continues on next page)

(continued from previous page)

```

...     for viewer in viewers:
...         viewer.plot()
...     input('Arrange viewers, then press <return> to proceed...')
...     for viewer in viewers:
...         viewer.plot()

```

The following section defines the required control parameters. The *cfl* parameter limits the size of the time step so that $dt = cfl * dx / \max(\text{velocity})$.

```

>>> cfl = 0.1
>>> tolerance = 1e-1
>>> dt = 1e-14
>>> timestep = 0
>>> relaxation = 0.5
>>> if __name__ == '__main__':
...     totalSteps = 1e10
... else:
...     totalSteps = 10

```

In the following time stepping scheme a time step is recalculated if the residual increases between sweeps or the required tolerance is not attained within 20 sweeps. The major quirk in this scheme is the requirement of updating the `matrixDiagonal` using the entire coupled matrix. This could be achieved more elegantly by calling `cacheMatrix()` only on the necessary part of the equation. This currently doesn't work properly in *FiPy*.

```

>>> while timestep < totalSteps:
...
...     sweep = 0
...     dt *= 1.1
...     residual = 1.
...     initialResidual = None
...
...     density.updateOld()
...     velocity.updateOld()
...     matrixDiagonal.updateOld()
...
...     while residual > tolerance:
...
...         densityPrevious[:] = density
...         velocityPrevious[:] = velocity
...         previousResidual = residual
...
...         dt = min(dt, dx / max(abs(velocity)) * cfl)
...
...         coupledEqn.cacheMatrix()
...         residual = coupledEqn.sweep(dt=dt)
...
...         if initialResidual is None:
...             initialResidual = residual
...
...         residual = residual / initialResidual
...
...         if residual > previousResidual * 1.1 or sweep > 20:
...             density[:] = density.old
...             velocity[:] = velocity.old
...             matrixDiagonal[:] = matrixDiagonal.old
...             dt = dt / 10.

```

(continues on next page)

(continued from previous page)

```

...         if __name__ == '__main__':
...             print('Recalculate the time step')
...             timestep -= 1
...             break
...         else:
...             matrixDiagonal[:] = coupledEqn.matrix.takeDiagonal()[mesh.
↪numberOfCells:2 * mesh.numberOfCells]
...             density[:] = relaxation * density + (1 - relaxation) * densityPrevious
...             velocity[:] = relaxation * velocity + (1 - relaxation) *
↪velocityPrevious
...
...             sweep += 1
...
...         if __name__ == '__main__' and timestep % 10 == 0:
...             print('timestep: %e / %e, dt: %1.5e, free energy: %1.5e' % (timestep,
↪totalSteps, dt, freeEnergy))
...             for viewer in viewers:
...                 viewer.plot()
...
...         timestep += 1

```

```

>>> from fipy import input
>>> if __name__ == '__main__':
...     input('finished')

```

```

>>> print(freeEnergy < 1.5e9)
True

```


Chapter 26

Updating FiPy

examples.updating.update2_0to3_0	How to update scripts from version 2.0 to 3.0.
examples.updating.update1_0to2_0	How to update scripts from version 1.0 to 2.0.
examples.updating.update0_1to1_0	How to update scripts from version 0.1 to 1.0.

26.1 examples.updating.update2_0to3_0

How to update scripts from version 2.0 to 3.0.

FiPy 3.0 introduces several syntax changes from *FiPy* 2.0. We appreciate that this is very inconvenient for our users, but we hope you'll agree that the new syntax is easier to read and easier to use. We assure you that this is not something we do casually; it has been over two and a half years since our last incompatible change (when *FiPy* 2.0 superseded *FiPy* 1.0).

All examples included with version 3.0 have been updated to use the new syntax, but any scripts you have written for *FiPy* 2.0 will need to be updated. A complete listing of the changes needed to take the *FiPy* examples scripts from version 2.0 to version 3.0 can be found with

```
$ git diff version-2_1 version-3_0 examples/
```

but we summarize the necessary changes here. If these tips are not sufficient to make your scripts compatible with *FiPy* 3.0, please don't hesitate to ask for help on the [mailing list](#).

The following items **must** be changed in your scripts

- We have reconsidered the change in *FiPy* 2.0 that included all of the functions of the *numerix* module in the *fipy* namespace. You now must be more explicit when referring to any of these functions:

```
>>> from fipy import *
>>> y = numerix.exp(x)
```

```
>>> from fipy.tools.numerix import exp
>>> y = exp(x)
```

We generally use the first, but you may see us import specific functions if we feel it improves readability. You should feel free to use whichever form you find most comfortable.

Note: the old behavior can be obtained, at least for now, by setting the `FIPY_INCLUDE_NUMERIX_ALL` environment variable.

- If your equation contains a *TransientTerm*, then you must specify the timestep by passing a `dt=` argument when calling `solve()` or `sweep()`.

The remaining changes are not *required*, but they make scripts easier to read and we recommend them. *FiPy* may issue a `DeprecationWarning` for some cases, to indicate that we may not maintain the old syntax indefinitely.

- “getter” and “setter” methods have been replaced with properties, e.g., use

```
>>> x, y = mesh.cellCenters
```

instead of

```
>>> x, y = mesh.getCellCenters()
```

- Boundary conditions are better applied with the `constrain()` method than with the old *FixedValue* and *FixedFlux* classes. See *Boundary Conditions*.
- Individual *Mesh* classes should be imported directly from `fipy.meshes` and not `fipy.meshes.numMesh`.
- The *Gmsh* meshes now have simplified names: *Gmsh2D* instead of *GmshImporter2D*, *Gmsh3D* instead of *GmshImporter3D*, and *Gmsh2DIn3DSpace* instead of *GmshImporter2DIn3DSpace*.

26.2 examples.updating.update1_0to2_0

How to update scripts from version 1.0 to 2.0.

FiPy 2.0 introduces several syntax changes from *FiPy* 1.0. We appreciate that this is very inconvenient for our users, but we hope you’ll agree that the new syntax is easier to read and easier to use. We assure you that this is not something we do casually; it has been over three years since our last incompatible change (when *FiPy* 1.0 superceded *FiPy* 0.1).

All examples included with version 2.0 have been updated to use the new syntax, but any scripts you have written for *FiPy* 1.0 will need to be updated. A complete listing of the changes needed to take the *FiPy* examples scripts from version 1.0 to version 2.0 can be found with:

```
$ git diff version-1_2 version-2_0 examples/
```

but we summarize the necessary changes here. If these tips are not sufficient to make your scripts compatible with *FiPy* 2.0, please don’t hesitate to ask for help on the [mailing list](#).

The following items **must** be changed in your scripts

- The dimension axis of a *Variable* is now first, not last

```
>>> x = mesh.getCellCenters()[0]
```

instead of

```
>>> x = mesh.getCellCenters()[..., 0]
```

This seemingly arbitrary change simplifies a great many things in *FiPy*, but the one most noticeable to the user is that you can now write

```
>>> x, y = mesh.getCellCenters()
```

instead of

```
>>> x = mesh.getCellCenters()[..., 0]
>>> y = mesh.getCellCenters()[..., 1]
```

Unfortunately, we cannot reliably automate this conversion, but we find that searching for “...,” and “:,” finds almost everything. Please don’t blindly “search & replace all” as that is almost bound to create more problems than it’s worth.

Note: Any vector constants must be reoriented. For instance, in order to offset a *Mesh*, you must write

```
>>> mesh = Grid2D(...) + ((deltax,), (deltay,))
```

or

```
>>> mesh = Grid2D(...) + [[deltax], [deltay]]
```

instead of

```
>>> mesh = Grid2D(...) + (deltax, deltax)
```

- *VectorCellVariable* and *VectorFaceVariable* no longer exist. *CellVariable* and *FaceVariable* now both inherit from *MeshVariable*, which can have arbitrary rank. A field of scalars (default) will have rank=0, a field of vectors will have rank=1, etc. You should write

```
>>> vectorField = CellVariable(mesh=mesh, rank=1)
```

instead of

```
>>> vectorField = VectorCellVariable(mesh=mesh)
```

Note: Because vector fields are properly supported, use vector operations to manipulate them, such as

```
>>> phase.getFaceGrad().dot((( 0, 1),
...                          (-1, 0)))
```

instead of the hackish

```
>>> phase.getFaceGrad()._take((1, 0), axis=1) * (-1, 1)
```

- For internal reasons, *FiPy* now supports *CellVariable* and *FaceVariable* objects that contain integers, but it is not meaningful to solve a PDE for an integer field (*FiPy* should issue a warning if you try). As a result, when given, initial values must be specified as floating-point values:

```
>>> var = CellVariable(mesh=mesh, value=1.)
```

where they used to be quietly accepted as integers

```
>>> var = CellVariable(mesh=mesh, value=1)
```

If the `value` argument is not supplied, the *CellVariable* will contain floats, as before.

- The `faces` argument to `BoundaryCondition` now takes a mask, instead of a list of Face IDs. Now you write

```
>>> X, Y = mesh.getFaceCenters()
>>> FixedValue(faces=mesh.getExteriorFaces() & (X**2 < 1e-6), value=...)
```

instead of

```
>>> exteriorFaces = mesh.getExteriorFaces()
>>> X = exteriorFaces.getCenters()[..., 0]
>>> FixedValue(faces=exteriorFaces.where(X**2 < 1e-6), value=...)
```

With the old syntax, a different call to `getCenters()` had to be made for each set of Face objects. It was also extremely difficult to specify boundary conditions that depended both on position in space and on the current values of any other *Variable*.

```
>>> FixedValue(faces=(mesh.getExteriorFaces()
...               & ((X**2 < 1e-6)
...                  & (Y > 3.))
...             | (phi.getArithmeticFaceValue()
...                < sin(gamma.getArithmeticFaceValue()))), value=...)
```

although it probably could have been done with a rather convoluted (and slow!) `filter` function passed to where. There no longer are any `filter` methods used in *FiPy*. You now would write

```
>>> x, y = mesh.cellCenters
>>> initialArray[(x < dx) | (x > (Lx - dx)) | (y < dy) | (y > (Ly - dy))] = 1.
```

instead of the *much* slower

```
>>> def cellFilter(cell):
...     return ((cell.center[0] < dx)
...             or (cell.center[0] > (Lx - dx))
...             or (cell.center[1] < dy)
...             or (cell.center[1] > (Ly - dy)))
```

```
>>> positiveCells = mesh.getCells(filter=cellFilter)
>>> for cell in positiveCells:
...     initialArray[cell.ID] = 1.
```

Although they still exist, we find very little cause to ever call `getCells()` or `fipy.meshes.mesh.Mesh.getFaces()`.

- Some modules, such as `fipy.solvers`, have been significantly rearranged. For example, you need to change

```
>>> from fipy.solvers.linearPCGSolver import LinearPCGSolver
```

to either

```
>>> from fipy import LinearPCGSolver
```

or

```
>>> from fipy.solvers.pysparse.linearPCGSolver import LinearPCGSolver
```

- The `numerix.max()` and `numerix.min()` functions no longer exist. Either call `max()` and `min()` or the `max()` and `min()` methods of a *Variable*.

- The `Numeric` module has not been supported for a long time. Be sure to use

```
>>> from fipy import numerix
```

instead of

```
>>> import Numeric
```

The remaining changes are not *required*, but they make scripts easier to read and we recommend them. *FiPy* may issue a `DeprecationWarning` for some cases, to indicate that we may not maintain the old syntax indefinitely.

- All of the most commonly used classes and functions in *FiPy* are directly accessible in the `fipy` namespace. For brevity, our examples now start with

```
>>> from fipy import *
```

instead of the explicit

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.terms.powerLawConvectionTerm import PowerLawConvectionTerm
>>> from fipy.variables.cellVariable import CellVariable
```

imports that we used to use. Most of the explicit imports should continue to work, so you do not need to change them if you don't wish to, but we find our own scripts much easier to read without them.

All of the `numerix` module is now imported into the `fipy` namespace, so you can call `numerix` functions a number of different ways, including:

```
>>> from fipy import *
>>> y = exp(x)
```

or

```
>>> from fipy import numerix
>>> y = numerix.exp(x)
```

or

```
>>> from fipy.tools.numerix import exp
>>> y = exp(x)
```

We generally use the first, but you may see us use the others, and should feel free to use whichever form you find most comfortable.

Note: Internally, *FiPy* uses explicit imports, as is considered *best Python practice*, but we feel that clarity trumps orthodoxy when it comes to the examples.

- The function `fipy.viewers.make()` has been renamed to `fipy.viewers.Viewer()`. All of the limits can now be supplied as direct arguments, as well (although this is not required). The result is a more natural syntax:

```
>>> from fipy import Viewer
>>> viewer = Viewer(vars=(alpha, beta, gamma), datamin=0, datamax=1)
```

instead of

```
>>> from fipy import viewers
>>> viewer = viewers.make(vars=(alpha, beta, gamma),
...                       limits={'datamin': 0, 'datamax': 1})
```

With the old syntax, there was also a temptation to write

```
>>> from fipy.viewers import make
>>> viewer = make(vars=(alpha, beta, gamma))
```

which can be very hard to understand after the fact (make? make what?).

- A `ConvectionTerm` can now calculate its Péclet number automatically, so the `diffusionTerm` argument is no longer required

```
>>> eq = (TransientTerm()
...       == DiffusionTerm(coeff=diffCoeff)
...       + PowerLawConvectionTerm(coeff=convCoeff))
```

instead of

```
>>> diffTerm = DiffusionTerm(coeff=diffCoeff)
>>> eq = (TransientTerm()
...       == diffTerm
...       + PowerLawConvectionTerm(coeff=convCoeff, diffusionTerm=diffTerm))
```

- An `ImplicitSourceTerm` now “knows” how to partition itself onto the solution matrix, so you can write

```
>>> S0 = mXi * phase * (1 - phase) - phase * S1
>>> source = S0 + ImplicitSourceTerm(coeff=S1)
```

instead of

```
>>> S0 = mXi * phase * (1 - phase) - phase * S1 * (S1 < 0)
>>> source = S0 + ImplicitSourceTerm(coeff=S1 * (S1 < 0))
```

It is definitely still advantageous to hand-linearize your source terms, but it is no longer necessary to worry about putting the “wrong” sign on the diagonal of the matrix.

- To make clearer the distinction between iterations, timesteps, and sweeps (see FAQ *Iterations, timesteps, and sweeps? Oh, my!*) the `steps` argument to a `Solver` object has been renamed `iterations`.
- `ImplicitDiffusionTerm` has been renamed to `DiffusionTerm`.

26.3 examples.updating.update0_1to1_0

How to update scripts from version 0.1 to 1.0.

It seems unlikely that many users are still running *FiPy* 0.1, but for those that are, the syntax of *FiPy* scripts changed considerably between version 0.1 and version 1.0. We incremented the full version-number to stress that previous scripts are incompatible. We strongly believe that these changes are for the better, resulting in easier code to write and read as well as slightly improved efficiency, but we realize that this represents an inconvenience to our users that have already written scripts of their own. We will strive to avoid any such incompatible changes in the future.

Any scripts you have written for *FiPy* 0.1 should be updated in two steps, first to work with *FiPy* 1.0, and then with *FiPy* 2.0. As a tutorial for updating your scripts, we will walk through updating `examples/convection/exponential1D/input.py` from *FiPy* 0.1. If you attempt to run that script with *FiPy* 1.0, the script will fail and you will see the errors shown below:

This example solves the steady-state convection-diffusion equation given by:

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = (10, 0)$, or

```
>>> diffCoeff = 1.
>>> convCoeff = (10., 0.)
```

We define a 1D mesh

```
>>> L = 10.
>>> nx = 1000
>>> ny = 1
>>> from fipy.meshes.grid2D import Grid2D
>>> mesh = Grid2D(L / nx, L / ny, nx, ny)
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } x = 0, \\ 1 & \text{at } x = L, \end{cases}$$

or

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> from fipy.boundaryConditions.fixedFlux import FixedFlux
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight),
...     FixedFlux(mesh.getFacesTop(), 0.),
...     FixedFlux(mesh.getFacesBottom(), 0.)
... )
```

The solution variable is initialized to *valueLeft*:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(
...     name = "concentration",
...     mesh = mesh,
...     value = valueLeft)
```

The `SteadyConvectionDiffusionScEquation` object is used to create the equation. It needs to be passed a convection term instantiator as follows:

```
>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> from fipy.solvers import *
>>> from fipy.equations.stdyConvDiffScEquation import _
↳SteadyConvectionDiffusionScEquation
Traceback (most recent call last):
...
ImportError: No module named equations.stdyConvDiffScEquation
>>> eq = SteadyConvectionDiffusionScEquation(
...     var = var,
```

(continues on next page)

(continued from previous page)

```
...     diffusionCoeff = diffCoeff,
...     convectionCoeff = convCoeff,
...     solver = LinearLUSolver(tolerance = 1.e-15, steps = 2000),
...     convectionScheme = ExponentialConvectionTerm,
...     boundaryConditions = boundaryConditions
... )
Traceback (most recent call last):
...
NameError: name 'SteadyConvectionDiffusionScEquation' is not defined
```

More details of the benefits and drawbacks of each type of convection term can be found in the numerical section of the manual. Essentially the *ExponentialConvectionTerm* and *PowerLawConvectionTerm* will both handle most types of convection diffusion cases with the *PowerLawConvectionTerm* being more efficient.

We iterate to equilibrium

```
>>> from fipy.iterators.iterator import Iterator
>>> it = Iterator((eq,))
Traceback (most recent call last):
...
NameError: name 'eq' is not defined
>>> it.timestep()
Traceback (most recent call last):
...
NameError: name 'it' is not defined
```

and test the solution against the analytical result

$$\phi = \frac{1 - \exp(-u_x x / D)}{1 - \exp(-u_x L / D)}$$

or

```
>>> axis = 0
>>> x = mesh.getCellCenters()[:, axis]
>>> from fipy.tools import numerix
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> numerix.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
0
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
Traceback (most recent call last):
...
ImportError: No module named grid2DGistViewer
```

```
...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

We see that a number of errors are thrown:

- ImportError: No module named equations.stdyConvDiffScEquation

- `NameError: name 'SteadyConvectionDiffusionScEquation' is not defined`
- `NameError: name 'eq' is not defined`
- `NameError: name 'it' is not defined`
- `ImportError: No module named grid2DGistViewer`

As is usually the case with computer programming, many of these errors are caused by earlier errors. Let us update the script, section by section:

Although no error was generated by the use of `Grid2D`, *FiPy* 1.0 supports a true 1D mesh class, so we instantiate the mesh as

```
>>> L = 10.
>>> nx = 1000
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = L / nx, nx = nx)
```

The `Grid2D` class with $ny = 1$ still works perfectly well for 1D problems, but the `Grid1D` class is slightly more efficient, and it makes the code clearer when a 1D geometry is actually desired.

Because the mesh is now 1D, we must update the convection coefficient vector to be 1D as well

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,)
```

The *FixedValue* boundary conditions at the left and right are unchanged, but a *Grid1D* mesh does not even have top and bottom faces:

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight))
```

The creation of the solution variable is unchanged:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name = "concentration",
...                     mesh = mesh,
...                     value = valueLeft)
```

The biggest change between *FiPy* 0.1 and *FiPy* 1.0 is that `Equation` objects no longer exist at all. Instead, *Term* objects can be simply added, subtracted, and equated to assemble an equation. Where before the assembly of the equation occurred in the black-box of `SteadyConvectionDiffusionScEquation`, we now assemble it directly:

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> diffTerm = ImplicitDiffusionTerm(coeff = diffCoeff)
```

```
>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> eq = diffTerm + ExponentialConvectionTerm(coeff = convCoeff,
...                                           diffusionTerm = diffTerm)
```

One thing that `SteadyConvectionDiffusionScEquation` took care of automatically was that a `ConvectionTerm` must know about any *DiffusionTerm* in the equation in order to calculate a Péclet number. Now, the *DiffusionTerm* must be explicitly passed to the `ConvectionTerm` in the *diffusionTerm* parameter.

The `Iterator` class still exists, but it is no longer necessary. Instead, the solution to an implicit steady-state problem like this can simply be obtained by telling the equation to solve itself (with an appropriate *solver* if desired, although the default `LinearPCGSolver` is usually suitable):

```
>>> from fipy.solvers import *
>>> eq.solve(var = var,
...         solver = LinearLUSolver(tolerance = 1.e-15, steps = 2000),
...         boundaryConditions = boundaryConditions)
```

Note: In version 0.1, the `Equation` object had to be told about the *Variable*, *Solver*, and *BoundaryCondition* objects when it was created (and it, in turn, passed much of this information to the *Term* objects in order to create them). In version 1.0, the *Term* objects (and the equation assembled from them) are abstract. The *Variable*, *Solver*, and *BoundaryCondition* objects are only needed by the `solve()` method (and, in fact, the same equation could be used to solve different variables, with different solvers, subject to different boundary conditions, if desired).

The analytical solution is unchanged, and we can test as before

```
>>> numerix.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
1
```

or we can use the slightly simpler syntax

```
>>> print(var.allclose(analyticalArray, rtol = 1e-10, atol = 1e-10))
1
```

The `ImportError: No module named grid2DGistViewer` results because the `Viewer` classes have been moved and renamed. This error could be resolved by changing the *import* statement appropriately:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gistViewer.gist1DViewer import Gist1DViewer
...     viewer = Gist1DViewer(vars = var)
...     viewer.plot()
```

Instead, rather than instantiating a particular `Viewer` (which you can still do, if you desire), a generic “factory” method will return a `Viewer` appropriate for the supplied *Variable* object(s):

```
>>> if __name__ == '__main__':
...     import fipy.viewers
...     viewer = fipy.viewers.make(vars = var)
...     viewer.plot()
```

Please do not hesitate to contact us if this example does not help you convert your existing scripts to *FiPy* 1.0.

Part III

fipy Package Documentation

How to Read the Modules Documentation

Each chapter describes one of the main sub-packages of the `fipy` package. The sub-package `fipy.package` can be found in the directory `fipy/package/`. In a few cases, there will be packages within packages, *e.g.* `fipy.package.subpackage` located in `fipy/package/subpackage/`. These sub-sub-packages will not be given their own chapters; rather, their contents will be described in the chapter for their containing package.

27.1 package.subpackage package

27.1.1 Submodules

27.1.2 package.subpackage.base module

This module can be found in the file `package/subpackage/base.py`. You make it available to your script by either:

```
import package.subpackage.base
```

in which case you refer to it by its full name of `package.subpackage.base`, or:

```
from package.subpackage import base
```

in which case you can refer simply to `base`.

class `package.subpackage.base.Base`

Bases: `object`

With very few exceptions, the name of a class will be the capitalized form of the module it resides in. Depending on how you imported the module above, you will refer to either `package.subpackage.object.Object` or `object.Object`. Alternatively, you can use:

```
from package.subpackage.object import Object
```

and then refer simply to `Object`. For many classes, there is a shorthand notation:

```
from fipy import Object
```

Python is an object-oriented language and the FiPy framework is composed of objects or classes. Knowledge of object-oriented programming (OOP) is not necessary to use either Python or FiPy, but a few concepts are useful. OOP involves two main ideas:

encapsulation an object binds data with actions or “methods”. In most cases, you will not work with an object’s data directly; instead, you will set, retrieve, or manipulate the data using the object’s methods.

Methods are functions that are attached to objects and that have direct access to the data of those objects. Rather than passing the object data as an argument to a function:

```
fn(data, arg1, arg2, ...)
```

you instruct an object to invoke an appropriate method:

```
object.meth(arg1, arg2, ...)
```

If you are unfamiliar with object-oriented practices, there probably seems little advantage in this reordering. You will have to trust us that the latter is a much more powerful way to do things.

inheritance specialized objects are derived or inherited from more general objects. Common behaviors or data are defined in base objects and specific behaviors or data are either added or modified in derived objects. Objects that declare the existence of certain methods, without actually defining what those methods do, are called “abstract”. These objects exist to define the behavior of a family of objects, but rely on their descendants to actually provide that behavior.

Unlike many object-oriented languages, *Python* does not prevent the creation of abstract objects, but we will include a notice like

Attention: This class is abstract. Always create one of its subclasses.

for abstract classes which should be used for documentation but never actually created in a *FiPy* script.

```
__dict__ = mappingproxy({'__module__': 'package.subpackage.base', '__doc__': '\n Wit
```

```
__init__()
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'package.subpackage.base'
```

```
__weakref__
```

list of weak references to the object (if defined)

```
method1()
```

This is one thing that you can instruct any object that derives from *Base* to do, by calling `myObjectDerivedFromBase.method1()`

Parameters *self* (*object*) – This special argument refers to the object that is being created.

Attention: *self* is supplied automatically by the *Python* interpreter to all methods. You don’t need to (and should not) specify it yourself.

```
method2()
```

This is another thing that you can instruct any object that derives from *Base* to do.

27.1.3 package.subpackage.object module

class package.subpackage.object.**Object** (*arg1*, *arg2=None*, *arg3='string'*)

Bases: *package.subpackage.base.Base*

This method, like all those whose names begin and end with “__” are special. You won’t ever need to call these methods directly, but *Python* will invoke them for you under certain circumstances, which are described in the [Python Reference Manual: Special Method Names](#).

As an example, the `__init__()` method is invoked when you create an object, as in:

```
obj = Object(arg1=something, arg3=somethingElse, ...)
```

Parameters

- **arg1** – this argument is required. *Python* supports named arguments, so you must either list the value for *arg1* first:

```
obj = Object(val1, val2)
```

or you can specify the arguments in any order, as long as they are named:

```
obj = Object(arg2=val2, arg1=val1)
```

- **arg2** – this argument may be omitted, in which case it will be assigned a default value of `None`. If you do not use named arguments (and we recommend that you do), all required arguments must be specified before any optional arguments.
- **arg3** – this argument may be omitted, in which case it will be assigned a default value of `'string'`.

`__init__(arg1, arg2=None, arg3='string')`

This method, like all those whose names begin and end with “__” are special. You won’t ever need to call these methods directly, but *Python* will invoke them for you under certain circumstances, which are described in the [Python Reference Manual: Special Method Names](#).

As an example, the `__init__()` method is invoked when you create an object, as in:

```
obj = Object(arg1=something, arg3=somethingElse, ...)
```

Parameters

- **arg1** – this argument is required. *Python* supports named arguments, so you must either list the value for *arg1* first:

```
obj = Object(val1, val2)
```

or you can specify the arguments in any order, as long as they are named:

```
obj = Object(arg2=val2, arg1=val1)
```

- **arg2** – this argument may be omitted, in which case it will be assigned a default value of `None`. If you do not use named arguments (and we recommend that you do), all required arguments must be specified before any optional arguments.
- **arg3** – this argument may be omitted, in which case it will be assigned a default value of `'string'`.

```
__module__ = 'package.subpackage.object'
```

```
method2()
```

Object provides a new definition for the behavior of *method2()*, whereas the behavior of *method1()* is defined by *Base*.

27.1.4 Module contents

Each chapter describes one of the main sub-packages of the `fipy` package. The sub-package `fipy.package` can be found in the directory `fipy/package/`. In a few cases, there will be packages within packages, *e.g.* `fipy.package.subpackage` located in `fipy/package/subpackage/`. These sub-sub-packages will not be given their own chapters; rather, their contents will be described in the chapter for their containing package.

fipy.boundaryConditions package

28.1 Submodules

28.2 fipy.boundaryConditions.boundaryCondition module

class fipy.boundaryConditions.boundaryCondition.**BoundaryCondition** (*faces*, *value*)

Bases: `object`

Generic boundary condition base class.

Attention: This class is abstract. Always create one of its subclasses.

Parameters

- **faces** (*FaceVariable* of `bool`) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.

`__dict__` = `mappingproxy({'__module__': 'fipy.boundaryConditions.boundaryCondition', '...`

`__init__` (*faces*, *value*)

Parameters

- **faces** (*FaceVariable* of `bool`) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.

`__module__` = `'fipy.boundaryConditions.boundaryCondition'`

`__repr__` ()

Return repr(self).

`__weakref__`

list of weak references to the object (if defined)

28.3 fipy.boundaryConditions.constraint module

class fipy.boundaryConditions.constraint.Constraint (*value, where=None*)

Bases: object

Object to hold a *Variable* to *value* at *where*

see *constrain()*

__dict__ = mappingproxy({'__module__': 'fipy.boundaryConditions.constraint', '__init__':

__init__ (*value, where=None*)

Object to hold a *Variable* to *value* at *where*

see *constrain()*

__module__ = 'fipy.boundaryConditions.constraint'

__repr__ ()

Return repr(self).

__weakref__

list of weak references to the object (if defined)

28.4 fipy.boundaryConditions.fixedFlux module

class fipy.boundaryConditions.fixedFlux.FixedFlux (*faces, value*)

Bases: *fipy.boundaryConditions.boundaryCondition.BoundaryCondition*

The *FixedFlux* boundary condition adds a contribution, equivalent to a fixed flux (Neumann condition), to the equation's RHS vector. The contribution, given by *value*, is only added to entries corresponding to the specified *faces*, and is weighted by the face areas.

Parameters

- **faces** (*FaceVariable* of *bool*) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.

__init__ (*faces, value*)

Parameters

- **faces** (*FaceVariable* of *bool*) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.

__module__ = 'fipy.boundaryConditions.fixedFlux'

28.5 fipy.boundaryConditions.fixedValue module

class fipy.boundaryConditions.fixedValue.FixedValue (*faces, value*)

Bases: *fipy.boundaryConditions.boundaryCondition.BoundaryCondition*

The *FixedValue* boundary condition adds a contribution, equivalent to a fixed value (Dirichlet condition), to the equation's RHS vector and coefficient matrix. The contributions are given by $-\text{value} \times G_{\text{face}}$ for the RHS vector and G_{face} for the coefficient matrix. The parameter G_{face} represents the term's geometric coefficient, which depends on the type of term and the mesh geometry.

Contributions are only added to entries corresponding to the specified faces.

Parameters

- **faces** (*FaceVariable* of *bool*) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.

`__module__ = 'fipy.boundaryConditions.fixedValue'`

28.6 fipy.boundaryConditions.nthOrderBoundaryCondition module

`class fipy.boundaryConditions.nthOrderBoundaryCondition.NthOrderBoundaryCondition` (*faces, value, order*)

Bases: *fipy.boundaryConditions.boundaryCondition.BoundaryCondition*

This boundary condition is generally used in conjunction with a *ImplicitDiffusionTerm* that has multiple coefficients. It does not have any direct effect on the solution matrices, but its derivatives do.

Creates an *NthOrderBoundaryCondition*.

Parameters

- **faces** (*FaceVariable* of *bool*) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.
- **order** (*int*) – Order of the boundary condition. An *order* of 0 corresponds to a *FixedValue* and an *order* of 1 corresponds to a *FixedFlux*. Even and odd orders behave like *FixedValue* and *FixedFlux* objects, respectively, but apply to higher order terms.

`__init__` (*faces, value, order*)

Creates an *NthOrderBoundaryCondition*.

Parameters

- **faces** (*FaceVariable* of *bool*) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.
- **order** (*int*) – Order of the boundary condition. An *order* of 0 corresponds to a *FixedValue* and an *order* of 1 corresponds to a *FixedFlux*. Even and odd orders behave like *FixedValue* and *FixedFlux* objects, respectively, but apply to higher order terms.

`__module__ = 'fipy.boundaryConditions.nthOrderBoundaryCondition'`

28.7 fipy.boundaryConditions.test module

Test numeric implementation of the mesh

28.8 Module contents

class `fipy.boundaryConditions.Constraint` (*value*, *where=None*)

Bases: `object`

Object to hold a *Variable* to *value* at *where*

see `constrain()`

`__dict__` = `mappingproxy({'__module__': 'fipy.boundaryConditions.constraint', '__init__`

`__init__` (*value*, *where=None*)

Object to hold a *Variable* to *value* at *where*

see `constrain()`

`__module__` = `'fipy.boundaryConditions.constraint'`

`__repr__` ()

Return repr(self).

`__weakref__`

list of weak references to the object (if defined)

class `fipy.boundaryConditions.FixedFlux` (*faces*, *value*)

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

The *FixedFlux* boundary condition adds a contribution, equivalent to a fixed flux (Neumann condition), to the equation's RHS vector. The contribution, given by *value*, is only added to entries corresponding to the specified *faces*, and is weighted by the face areas.

Parameters

- **faces** (*FaceVariable* of `bool`) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.

`__init__` (*faces*, *value*)

Parameters

- **faces** (*FaceVariable* of `bool`) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.

`__module__` = `'fipy.boundaryConditions.fixedFlux'`

class `fipy.boundaryConditions.FixedValue` (*faces*, *value*)

Bases: `fipy.boundaryConditions.boundaryCondition.BoundaryCondition`

The *FixedValue* boundary condition adds a contribution, equivalent to a fixed value (Dirichlet condition), to the equation's RHS vector and coefficient matrix. The contributions are given by $-value \times G_{\text{face}}$ for the RHS vector and G_{face} for the coefficient matrix. The parameter G_{face} represents the term's geometric coefficient, which depends on the type of term and the mesh geometry.

Contributions are only added to entries corresponding to the specified faces.

Parameters

- **faces** (*FaceVariable* of `bool`) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.

`__module__` = `'fipy.boundaryConditions.fixedValue'`

class fipy.boundaryConditions.NthOrderBoundaryCondition (*faces, value, order*)

Bases: *fipy.boundaryConditions.boundaryCondition.BoundaryCondition*

This boundary condition is generally used in conjunction with a *ImplicitDiffusionTerm* that has multiple coefficients. It does not have any direct effect on the solution matrices, but its derivatives do.

Creates an *NthOrderBoundaryCondition*.

Parameters

- **faces** (*FaceVariable* of *bool*) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.
- **order** (*int*) – Order of the boundary condition. An *order* of 0 corresponds to a *FixedValue* and an *order* of 1 corresponds to a *FixedFlux*. Even and odd orders behave like *FixedValue* and *FixedFlux* objects, respectively, but apply to higher order terms.

__init__ (*faces, value, order*)

Creates an *NthOrderBoundaryCondition*.

Parameters

- **faces** (*FaceVariable* of *bool*) – Mask of faces where this condition applies.
- **value** (*float*) – Value to impose.
- **order** (*int*) – Order of the boundary condition. An *order* of 0 corresponds to a *FixedValue* and an *order* of 1 corresponds to a *FixedFlux*. Even and odd orders behave like *FixedValue* and *FixedFlux* objects, respectively, but apply to higher order terms.

__module__ = 'fipy.boundaryConditions.nthOrderBoundaryCondition'

fipy.matrices package

29.1 Submodules

29.2 fipy.matrices.offsetSparseMatrix module

`fipy.matrices.offsetSparseMatrix.OffsetSparseMatrix` (*SparseMatrix*, *numberOfVariables*, *numberOfEquations*)

Used in binary terms. *equationIndex* and *varIndex* need to be set statically before instantiation.

29.3 fipy.matrices.petscMatrix module

29.4 fipy.matrices.pysparseMatrix module

29.5 fipy.matrices.scipyMatrix module

29.6 fipy.matrices.sparseMatrix module

29.7 fipy.matrices.test module

29.8 fipy.matrices.trilinosMatrix module

29.9 Module contents

fipy.meshes package

30.1 Subpackages

30.1.1 fipy.meshes.builders package

Submodules

`fipy.meshes.builders.abstractGridBuilder` module

`fipy.meshes.builders.grid1DBuilder` module

`fipy.meshes.builders.grid2DBuilder` module

`fipy.meshes.builders.grid3DBuilder` module

`fipy.meshes.builders.periodicGrid1DBuilder` module

`fipy.meshes.builders.utilityClasses` module

Module contents

30.1.2 fipy.meshes.representations package

Submodules

`fipy.meshes.representations.abstractRepresentation` module

`fipy.meshes.representations.gridRepresentation` module

`fipy.meshes.representations.meshRepresentation` module

Module contents

30.1.3 fipy.meshes.topologies package

Submodules

`fipy.meshes.topologies.abstractTopology` module

`fipy.meshes.topologies.gridTopology` module

`fipy.meshes.topologies.meshTopology` module

Module contents

30.2 Submodules

30.3 `fipy.meshes.abstractMesh` module

```
class fipy.meshes.abstractMesh.AbstractMesh (communicator, _RepresentationClass=<class  
fipy.meshes.representations.abstractRepresentation._AbstractRepresentation  
_TopologyClass=<class  
fipy.meshes.topologies.abstractTopology._AbstractTopology>)
```

Bases: `object`

A class encapsulating all commonalities among meshes in FiPy.

property `VTKCellDataSet`

Returns a TVTK *DataSet* representing the cells of this mesh

property `VTKFaceDataSet`

Returns a TVTK *DataSet* representing the face centers of this mesh

__add__ (*other*)

Either translate a *Mesh* or concatenate two *Mesh* objects.

```
>>> from fipy.meshes import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print(baseMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5]]
```

If a vector is added to a *Mesh*, a translated *Mesh* is returned

```
>>> translatedMesh = baseMesh + ((5,), (10,))
>>> print(translatedMesh.cellCenters)
[[ 5.5  6.5  5.5  6.5]
 [10.5 10.5 11.5 11.5]]
```

If a *Mesh* is added to a *Mesh*, a concatenation of the two *Mesh* objects is returned

```
>>> addedMesh = baseMesh + (baseMesh + ((2,), (0,)))
>>> print(addedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  2.5  3.5  2.5  3.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5]]
```

The two *Mesh* objects need not be properly aligned in order to concatenate them but the resulting mesh may not have the intended connectivity

```
>>> addedMesh = baseMesh + (baseMesh + ((3,), (0,)))
>>> print(addedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  3.5  4.5  3.5  4.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5]]
```

```
>>> addedMesh = baseMesh + (baseMesh + ((2,), (2,)))
>>> print(addedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  2.5  3.5  2.5  3.5]
 [ 0.5  0.5  1.5  1.5  2.5  2.5  3.5  3.5]]
```

No provision is made to avoid or consolidate overlapping *Mesh* objects

```
>>> addedMesh = baseMesh + (baseMesh + ((1,), (0,)))
>>> print(addedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  1.5  2.5  1.5  2.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5]]
```

Different *Mesh* classes can be concatenated

```
>>> from fipy.meshes import Tri2D
>>> triMesh = Tri2D(dx = 1.0, dy = 1.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
>>> cellCenters = [[0.5, 1.5, 0.5, 1.5, 2.83333333, 3.83333333,
...                 2.5, 3.5, 2.16666667, 3.16666667, 2.5, 3.5],
...                [0.5, 0.5, 1.5, 1.5, 0.5, 0.5, 0.83333333, 0.83333333,
...                 0.5, 0.5, 0.16666667, 0.16666667]]
>>> print(numerix.allclose(triAddedMesh.cellCenters,
...                         cellCenters))
...
True
```

again, their faces need not align, but the mesh may not have the desired connectivity

```
>>> triMesh = Tri2D(dx = 1.0, dy = 2.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
>>> cellCenters = [[ 0.5, 1.5, 0.5, 1.5, 2.83333333, 3.83333333,
...                 2.5, 3.5, 2.16666667, 3.16666667, 2.5, 3.5],
...                [ 0.5, 0.5, 1.5, 1.5, 1., 1.,
...                 1.66666667, 1.66666667, 1., 1., 0.33333333, 0.33333333]]
>>> print(numerix.allclose(triAddedMesh.cellCenters,
...                         cellCenters))
...
True
```

Mesh concatenation is not limited to 2D meshes

```
>>> from fipy.meshes import Grid3D
>>> threeDBaseMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
...                          nx = 2, ny = 2, nz = 2)
>>> threeDSecondMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
...                             nx = 1, ny = 1, nz = 1)
>>> threeDAddedMesh = threeDBaseMesh + (threeDSecondMesh + ((2,), (0,), (0,)))
>>> print(threeDAddedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  0.5  1.5  0.5  1.5  2.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5  0.5]
 [ 0.5  0.5  0.5  0.5  1.5  1.5  1.5  1.5  0.5]]
```

but the different *Mesh* objects must, of course, have the same dimensionality.

```
>>> InvalidMesh = threeDBaseMesh + baseMesh
Traceback (most recent call last):
...
MeshAdditionError: Dimensions do not match
```

```
__dict__ = mappingproxy({'__module__': 'fipy.meshes.abstractMesh', '__doc__': '\n A
__div__ (other)
    Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> Ab-
    stractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__ ()

__init__ (communicator, _RepresentationClass=<class 'fipy.meshes.representations.abstractRepresentation._AbstractRepresenta
    _TopologyClass=<class 'fipy.meshes.topologies.abstractTopology._AbstractTopology'> )
    Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.meshes.abstractMesh'

__mul__ (other)

__radd__ (other)
    Either translate a Mesh or concatenate two Mesh objects.
```

```
>>> from fipy.meshes import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print(baseMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5]]
```

If a vector is added to a *Mesh*, a translated *Mesh* is returned

```
>>> translatedMesh = baseMesh + ((5,), (10,))
>>> print(translatedMesh.cellCenters)
[[ 5.5  6.5  5.5  6.5]
 [10.5 10.5 11.5 11.5]]
```

If a *Mesh* is added to a *Mesh*, a concatenation of the two *Mesh* objects is returned

```
>>> addedMesh = baseMesh + (baseMesh + ((2,), (0,)))
>>> print(addedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  2.5  3.5  2.5  3.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5]]
```

The two *Mesh* objects need not be properly aligned in order to concatenate them but the resulting mesh may not have the intended connectivity

```
>>> addedMesh = baseMesh + (baseMesh + ((3,), (0,)))
>>> print(addedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  3.5  4.5  3.5  4.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5]]
```

```
>>> addedMesh = baseMesh + (baseMesh + ((2,), (2,)))
>>> print(addedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  2.5  3.5  2.5  3.5]
 [ 0.5  0.5  1.5  1.5  2.5  2.5  3.5  3.5]]
```

No provision is made to avoid or consolidate overlapping *Mesh* objects

```
>>> addedMesh = baseMesh + (baseMesh + ((1,), (0,)))
>>> print(addedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  1.5  2.5  1.5  2.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5]]
```

Different *Mesh* classes can be concatenated

```
>>> from fipy.meshes import Tri2D
>>> triMesh = Tri2D(dx = 1.0, dy = 1.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
>>> cellCenters = [[0.5, 1.5, 0.5, 1.5, 2.83333333, 3.83333333,
...                 2.5, 3.5, 2.16666667, 3.16666667, 2.5, 3.5],
...                [0.5, 0.5, 1.5, 1.5, 0.5, 0.5, 0.83333333, 0.83333333,
...                 0.5, 0.5, 0.16666667, 0.16666667]]
>>> print( Numerix.allclose(triAddedMesh.cellCenters,
...                          cellCenters))
...
True
```

again, their faces need not align, but the mesh may not have the desired connectivity

```
>>> triMesh = Tri2D(dx = 1.0, dy = 2.0, nx = 2, ny = 1)
>>> triMesh = triMesh + ((2,), (0,))
>>> triAddedMesh = baseMesh + triMesh
>>> cellCenters = [[ 0.5, 1.5, 0.5, 1.5, 2.83333333, 3.83333333,
...                 2.5, 3.5, 2.16666667, 3.16666667, 2.5, 3.5],
...                [ 0.5, 0.5, 1.5, 1.5, 1., 1.,
...                 1.66666667, 1.66666667, 1., 1., 0.33333333, 0.33333333]]
>>> print( Numerix.allclose(triAddedMesh.cellCenters,
...                          cellCenters))
...
True
```

Mesh concatenation is not limited to 2D meshes

```
>>> from fipy.meshes import Grid3D
>>> threeDBaseMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
...                          nx = 2, ny = 2, nz = 2)
>>> threeDSecondMesh = Grid3D(dx = 1.0, dy = 1.0, dz = 1.0,
...                            nx = 1, ny = 1, nz = 1)
>>> threeDAddedMesh = threeDBaseMesh + (threeDSecondMesh + ((2,), (0,), (0,)))
>>> print(threeDAddedMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5  0.5  1.5  0.5  1.5  2.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5  0.5]
 [ 0.5  0.5  0.5  0.5  1.5  1.5  1.5  1.5  0.5]]
```

but the different *Mesh* objects must, of course, have the same dimensionality.

```
>>> InvalidMesh = threeDBaseMesh + baseMesh
Traceback (most recent call last):
...
MeshAdditionError: Dimensions do not match
```

```
__repr__ ()
    Return repr(self).

__rmul__ (other)

__setstate__ (state)
```

__sub__ (*other*)

Tests. `>>> from fipy import * >>> m = Grid1D() >>> print((m - ((1,))).cellCenters) [[-0.5]] >>> ((1,)) - m` Traceback (most recent call last): ... `TypeError: unsupported operand type(s) for -: 'tuple' and 'UniformGrid1D'`

__truediv__ (*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2.` Traceback (most recent call last): ... `NotImplementedError`

__weakref__

list of weak references to the object (if defined)

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

property cellDistanceVectors

property cellFaceIDs

Topology properties

property cellToFaceDistanceVectors

property cellVolumes

property extents

property exteriorFaces

property faceCenters

property facesBack

Return list of faces on back boundary of 3D *Mesh* with the z-axis running from front to back.

```
>>> from fipy import Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> print(numerix.allequal((6, 7, 8, 9, 10, 11),
...                        numerix.nonzero(mesh.facesBack) [0]))
True
>>> ignore = mesh.facesBack.value
```

property facesBottom

Return list of faces on bottom boundary of 2D or 3D *Mesh* with the y-axis running from bottom to top.

```
>>> from fipy import Grid2D, Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> print(numerix.allequal((12, 13, 14),
...                        numerix.nonzero(mesh.facesBottom) [0]))
True
>>> ignore = mesh.facesBottom.value
>>> x, y, z = mesh.faceCenters
>>> print(numerix.allequal((12, 13),
...                        numerix.nonzero(mesh.facesBottom & (x < 1)) [0]))
True
>>> ignore = mesh.facesBottom.value
```

property facesDown

Return list of faces on bottom boundary of 2D or 3D *Mesh* with the y-axis running from bottom to top.

```

>>> from fipy import Grid2D, Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> print(numerix.allequal((12, 13, 14),
...                         numerix.nonzero(mesh.facesBottom)[0]))
True
>>> ignore = mesh.facesBottom.value
>>> x, y, z = mesh.faceCenters
>>> print(numerix.allequal((12, 13),
...                         numerix.nonzero(mesh.facesBottom & (x < 1))[0]))
True
>>> ignore = mesh.facesBottom.value

```

property facesFront

Return list of faces on front boundary of 3D *Mesh* with the z-axis running from front to back.

```

>>> from fipy import Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> print(numerix.allequal((0, 1, 2, 3, 4, 5),
...                         numerix.nonzero(mesh.facesFront)[0]))
True
>>> ignore = mesh.facesFront.value

```

property facesLeft

Return face on left boundary of *Mesh* as list with the x-axis running from left to right.

```

>>> from fipy import Grid2D, Grid3D
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> print(numerix.allequal((21, 25),
...                         numerix.nonzero(mesh.facesLeft)[0]))
True
>>> ignore = mesh.facesLeft.value
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print(numerix.allequal((9, 13),
...                         numerix.nonzero(mesh.facesLeft)[0]))
True
>>> ignore = mesh.facesLeft.value

```

property facesRight

Return list of faces on right boundary of *Mesh* with the x-axis running from left to right.

```

>>> from fipy import Grid2D, Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> print(numerix.allequal((24, 28),
...                         numerix.nonzero(mesh.facesRight)[0]))
True
>>> ignore = mesh.facesRight.value
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print(numerix.allequal((12, 16),
...                         numerix.nonzero(mesh.facesRight)[0]))
True
>>> ignore = mesh.facesRight.value

```

property facesTop

Return list of faces on top boundary of 2D or 3D *Mesh* with the y-axis running from bottom to top.

```

>>> from fipy import Grid2D, Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)

```

(continues on next page)

(continued from previous page)

```

>>> print(numerix.allequal((18, 19, 20),
...                         numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print(numerix.allequal((6, 7, 8),
...                         numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value

```

property facesUp

Return list of faces on top boundary of 2D or 3D *Mesh* with the y-axis running from bottom to top.

```

>>> from fipy import Grid2D, Grid3D, numerix
>>> mesh = Grid3D(nx = 3, ny = 2, nz = 1, dx = 0.5, dy = 2., dz = 4.)
>>> print(numerix.allequal((18, 19, 20),
...                         numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print(numerix.allequal((6, 7, 8),
...                         numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value

```

getNearestCell (*point*)**property interiorFaceCellIDs****property interiorFaceIDs****property interiorFaces****property scale****property scaledCellDistances****property scaledCellToCellDistances****property scaledCellVolumes****property scaledFaceAreas****property scaledFaceToCellDistances****property x**

Equivalent to using `cellCenters[0]`.

```

>>> from fipy import *
>>> print(Grid1D(nx=2).x)
[ 0.5  1.5]

```

property y

Equivalent to using `cellCenters[1]`.

```

>>> from fipy import *
>>> print(Grid2D(nx=2, ny=2).y)
[ 0.5  0.5  1.5  1.5]
>>> print(Grid1D(nx=2).y)
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property zEquivalent to using `cellCenters[2]`.

```
>>> from fipy import *
>>> print(Grid3D(nx=2, ny=2, nz=2).z)
[ 0.5  0.5  0.5  0.5  1.5  1.5  1.5  1.5]
>>> print(Grid2D(nx=2, ny=2).z)
Traceback (most recent call last):
...
AttributeError: 1D and 2D meshes do not have a "z" attribute.
```

30.4 fipy.meshes.cylindricalGrid1D module

30.5 fipy.meshes.cylindricalGrid2D module

30.6 fipy.meshes.cylindricalNonUniformGrid1D module

1D Mesh

```
class fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D(dx=1.0,
                                                                              nx=None,
                                                                              ori-
                                                                              gin=0,
                                                                              over-
                                                                              lap=2,
                                                                              com-
                                                                              mu-
                                                                              ni-
                                                                              ca-
                                                                              tor=SerialPETScComm,
                                                                              *args,
                                                                              **kwargs)
```

Bases: `fipy.meshes.nonUniformGrid1D.NonUniformGrid1D`

Creates a 1D cylindrical grid mesh.

```
>>> mesh = CylindricalNonUniformGrid1D(nx = 3)
>>> print(mesh.cellCenters)
[[ 0.5  1.5  2.5]]
```

```
>>> mesh = CylindricalNonUniformGrid1D(dx = (1, 2, 3))
>>> print(mesh.cellCenters)
[[ 0.5  2.  4.5]]
```

```
>>> print(numerix.allclose(mesh.cellVolumes, (0.5, 4., 13.5)))
True
```

```
>>> mesh = CylindricalNonUniformGrid1D(nx = 2, dx = (1, 2, 3))
Traceback (most recent call last):
...
IndexError: nx != len(dx)
```

```
>>> mesh = CylindricalNonUniformGrid1D(nx=2, dx=(1., 2.)) + ((1.,),)
>>> print(mesh.cellCenters)
[[ 1.5  3. ]]
>>> print(numerix.allclose(mesh.cellVolumes, (1.5, 6)))
True
```

__init__(*dx=1.0, nx=None, origin=0, overlap=2, communicator=SerialPETScCommWrapper(), *args, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.meshes.cylindricalNonUniformGrid1D'

__mul__(*factor*)

Dilate a *Mesh* by *factor*.

```
>>> from fipy.meshes import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print(baseMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5]]
```

The *factor* can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.5  1.5  4.5  4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.  1.  3.  3. ]]
```

but the vector must have the same dimensionality as the *Mesh*

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

30.7 fipy.meshes.cylindricalNonUniformGrid2D module

2D rectangular Mesh

```
class fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D(dx=1.0,
                                                                           dy=1.0,
                                                                           nx=None,
                                                                           ny=None,
                                                                           ori-
                                                                           gin=0.0,
                                                                           0.0,
                                                                           over-
                                                                           lap=2,
                                                                           com-
                                                                           mu-
                                                                           ni-
                                                                           ca-
                                                                           tor=SerialPETScCommWrapper(),
                                                                           *args,
                                                                           **kwargs)
```

Bases: *fipy.meshes.nonUniformGrid2D.NonUniformGrid2D*

Creates a 2D cylindrical grid mesh with horizontal faces numbered first and then vertical faces.

```
__init__(dx=1.0, dy=1.0, nx=None, ny=None, origin=0.0, 0.0, overlap=2, commu-
          tor=SerialPETScCommWrapper(), *args, **kwargs)
    Initialize self. See help(type(self)) for accurate signature.
```

```
__module__ = 'fipy.meshes.cylindricalNonUniformGrid2D'
```

```
__mul__(factor)
    Dilate a Mesh by factor.
```

```
>>> from fipy.meshes import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print(baseMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5]]
```

The *factor* can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.5  1.5  4.5  4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.   1.   3.   3.]]
```

but the vector must have the same dimensionality as the *Mesh*

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

30.8 fipy.meshes.cylindricalUniformGrid1D module

1D Mesh

```
class fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D(dx=1.0,
                                                                    nx=1,
                                                                    ori-
                                                                    gin=0,
                                                                    over-
                                                                    lap=2,
                                                                    com-
                                                                    munica-
                                                                    tor=SerialPETScCommWrapper(
                                                                    *args,
                                                                    **kwargs)
```

Bases: *fipy.meshes.uniformGrid1D.UniformGrid1D*

Creates a 1D cylindrical grid mesh.

```
>>> mesh = CylindricalUniformGrid1D(nx = 3)
>>> print(mesh.cellCenters)
[[ 0.5  1.5  2.5]]
```

```
__init__(dx=1.0, nx=1, origin=0, overlap=2, communicator=SerialPETScCommWrapper(), *args,
          **kwargs)
    Initialize self. See help(type(self)) for accurate signature.
__module__ = 'fipy.meshes.cylindricalUniformGrid1D'
property cellVolumes
```

30.9 fipy.meshes.cylindricalUniformGrid2D module

2D cylindrical rectangular Mesh with constant spacing in x and constant spacing in y

```
class fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D(dx=1.0,
                                                                    dy=1.0,
                                                                    nx=1,
                                                                    ny=1,
                                                                    ori-
                                                                    gin=0,
                                                                    0, over-
                                                                    lap=2,
                                                                    com-
                                                                    munica-
                                                                    tor=SerialPETScCommWrapper(
                                                                    *args,
                                                                    **kwargs)
```

Bases: *fipy.meshes.uniformGrid2D.UniformGrid2D*

Creates a 2D cylindrical grid in the radial and axial directions, appropriate for axial symmetry.

```
__init__(dx=1.0, dy=1.0, nx=1, ny=1, origin=0, 0, overlap=2, commu-
        tor=SerialPETScCommWrapper(), *args, **kwargs)
    Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.meshes.cylindricalUniformGrid2D'

property cellVolumes
```

30.10 fipy.meshes.factoryMeshes module

```
fipy.meshes.factoryMeshes.Grid3D(dx=1.0, dy=1.0, dz=1.0, nx=None, ny=None, nz=None,
                                Lx=None, Ly=None, Lz=None, overlap=2, commu-
                                tor=SerialPETScCommWrapper())
```

Factory function to select between *UniformGrid3D* and *NonUniformGrid3D*. If $L\{x,y,z\}$ is specified, the length of the domain is always $L\{x,y,z\}$ regardless of $d\{x,y,z\}$, unless $d\{x,y,z\}$ is a list of spacings, in which case $L\{x,y,z\}$ will be the sum of $d\{x,y,z\}$ and $n\{x,y,z\}$ will be the count of $d\{x,y,z\}$.

Parameters

- **dx** (*float*) – Grid spacing in the horizontal direction
- **dy** (*float*) – Grid spacing in the vertical direction
- **dz** (*float*) – Grid spacing in the depth direction
- **nx** (*int*) – Number of cells in the horizontal direction
- **ny** (*int*) – Number of cells in the vertical direction
- **nz** (*int*) – Number of cells in the depth direction
- **Lx** (*float*) – Domain length in the horizontal direction
- **Ly** (*float*) – Domain length in the vertical direction
- **Lz** (*float*) – Domain length in the depth direction
- **overlap** (*int*) – Number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, *fipy.tools.serialComm* or *fipy.tools.parallelComm*. Select *~fipy.tools.serialComm* to create a serial mesh when running in parallel; mostly used for test purposes.

```
fipy.meshes.factoryMeshes.Grid2D(dx=1.0, dy=1.0, nx=None, ny=None, Lx=None, Ly=None,
                                overlap=2, communicator=SerialPETScCommWrapper())
```

Factory function to select between *UniformGrid2D* and *NonUniformGrid2D*. If $L\{x,y\}$ is specified, the length of the domain is always $L\{x,y\}$ regardless of $d\{x,y\}$, unless $d\{x,y\}$ is a list of spacings, in which case $L\{x,y\}$ will be the sum of $d\{x,y\}$ and $n\{x,y\}$ will be the count of $d\{x,y\}$.

```
>>> print(Grid2D(Lx=3., nx=2).dx)
1.5
```

Parameters

- **dx** (*float*) – Grid spacing in the horizontal direction
- **dy** (*float*) – Grid spacing in the vertical direction
- **nx** (*int*) – Number of cells in the horizontal direction

- **ny** (*int*) – Number of cells in the vertical direction
- **Lx** (*float*) – Domain length in the horizontal direction
- **Ly** (*float*) – Domain length in the vertical direction
- **overlap** (*int*) – Number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, *fipy.tools.serialComm* or *fipy.tools.parallelComm*. Select *~fipy.tools.serialComm* to create a serial mesh when running in parallel; mostly used for test purposes.

```
fipy.meshes.factoryMeshes.Grid1D(dx=1.0, nx=None, Lx=None, overlap=2, communicator=SerialPETScCommWrapper())
```

Factory function to select between *UniformGrid1D* and *NonUniformGrid1D*. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*, unless *dx* is a list of spacings, in which case *Lx* will be the sum of *dx* and *nx* will be the count of *dx*.

Parameters

- **dx** (*float*) – Grid spacing in the horizontal direction
- **nx** (*int*) – Number of cells in the horizontal direction
- **Lx** (*float*) – Domain length in the horizontal direction
- **overlap** (*int*) – Number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, *fipy.tools.serialComm* or *fipy.tools.parallelComm*. Select *~fipy.tools.serialComm* to create a serial mesh when running in parallel; mostly used for test purposes.

```
fipy.meshes.factoryMeshes.CylindricalGrid2D(dr=None, dz=None, nr=None, nz=None, Lr=None, Lz=None, dx=1.0, dy=1.0, nx=None, ny=None, Lx=None, Ly=None, origin=0, 0, overlap=2, communicator=SerialPETScCommWrapper())
```

Factory function to select between *CylindricalUniformGrid2D* and *CylindricalNonUniformGrid2D*. If *Lr* is specified the length of the domain is always *Lr* regardless of *dr*, unless *dr* is a list of spacings, in which case *Lr* will be the sum of *dr*.

Parameters

- **dr** (*float*) – Grid spacing in the radial direction. Alternative: *dx*.
- **dz** (*float*) – grid spacing in the vertical direction. Alternative: *dy*.
- **nr** (*int*) – Number of cells in the radial direction. Alternative: *nx*.
- **nz** (*int*) – Number of cells in the vertical direction. Alternative: *ny*.
- **Lr** (*float*) – Domain length in the radial direction. Alternative: *Lx*.
- **Lz** (*float*) – Domain length in the vertical direction. Alternative: *Ly*.
- **overlap** (*int*) – the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, *fipy.tools.serialComm* or *fipy.tools.parallelComm*. Select *~fipy.tools.serialComm* to create a serial mesh when running in parallel; mostly used for test purposes.

```
fipy.meshes.factoryMeshes.CylindricalGrid1D(dr=None, nr=None, Lr=None,
                                             dx=1.0, nx=None, Lx=None, ori-
                                             gin=0, overlap=2, communica-
                                             tor=SerialPETScCommWrapper())
```

Factory function to select between *CylindricalUniformGrid1D* and *CylindricalNonUniformGrid1D*. If *Lr* is specified the length of the domain is always *Lr* regardless of *dr*, unless *dr* is a list of spacings, in which case *Lr* will be the sum of *dr*.

Parameters

- **dr** (*float*) – Grid spacing in the radial direction. Alternative: *dx*.
- **nr** (*int*) – Number of cells in the radial direction. Alternative: *nx*.
- **Lr** (*float*) – Domain length in the radial direction. Alternative: *Lx*.
- **overlap** (*int*) – the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, *fipy.tools.serialComm* or *fipy.tools.parallelComm*. Select *~fipy.tools.serialComm* to create a serial mesh when running in parallel; mostly used for test purposes.

30.11 fipy.meshes.gmshMesh module

```
fipy.meshes.gmshMesh.openMSHFile(name, dimensions=None, coordDimensions=None, commu-
                                   nicator=SerialPETScCommWrapper(), overlap=1, mode='r',
                                   background=None)
```

Open a Gmsh *MSH* file

Parameters

- **filename** (*str*) – Gmsh output file
- **dimensions** (*int*) – Dimension of mesh
- **coordDimensions** (*int*) – Dimension of shapes
- **overlap** (*int*) – The number of overlapping cells for parallel simulations. Generally 1 is adequate. Higher order equations or discretizations require more. If *overlap* is greater than one, communication reverts to serial, as Gmsh only provides one layer of ghost cells.
- **mode** (*str*) – Beginning with *r* for reading and *w* for writing. The file will be created if it doesn't exist when opened for writing; it will be truncated when opened for writing. Add a *b* to the mode for binary files.
- **background** (*CellVariable*) – Specifies the desired characteristic lengths of the mesh cells

```
fipy.meshes.gmshMesh.openPOSFile(name, communicator=SerialPETScCommWrapper(),
                                   mode='w')
```

Open a Gmsh *POS* post-processing file

```
class fipy.meshes.gmshMesh.Gmsh2D(arg, coordDimensions=2, communica-
                                   tor=SerialPETScCommWrapper(), overlap=1, back-
                                   ground=None)
```

Bases: *fipy.meshes.mesh2D.Mesh2D*

Construct a 2D Mesh using Gmsh

```
>>> radius = 5.
>>> side = 4.
>>> squaredCircle = Gmsh2D('''
... // A mesh consisting of a square inside a circle inside a circle
...
... // define the basic dimensions of the mesh
...
... cellSize = 1;
... radius = %(radius)g;
... side = %(side)g;
...
... // define the compass points of the inner circle
...
... Point(1) = {0, 0, 0, cellSize};
... Point(2) = {-radius, 0, 0, cellSize};
... Point(3) = {0, radius, 0, cellSize};
... Point(4) = {radius, 0, 0, cellSize};
... Point(5) = {0, -radius, 0, cellSize};
...
... // define the compass points of the outer circle
...
... Point(6) = {-2*radius, 0, 0, cellSize};
... Point(7) = {0, 2*radius, 0, cellSize};
... Point(8) = {2*radius, 0, 0, cellSize};
... Point(9) = {0, -2*radius, 0, cellSize};
...
... // define the corners of the square
...
... Point(10) = {side/2, side/2, 0, cellSize/2};
... Point(11) = {-side/2, side/2, 0, cellSize/2};
... Point(12) = {-side/2, -side/2, 0, cellSize/2};
... Point(13) = {side/2, -side/2, 0, cellSize/2};
...
... // define the inner circle
...
... Circle(1) = {2, 1, 3};
... Circle(2) = {3, 1, 4};
... Circle(3) = {4, 1, 5};
... Circle(4) = {5, 1, 2};
...
... // define the outer circle
...
... Circle(5) = {6, 1, 7};
... Circle(6) = {7, 1, 8};
... Circle(7) = {8, 1, 9};
... Circle(8) = {9, 1, 6};
...
... // define the square
...
... Line(9) = {10, 13};
... Line(10) = {13, 12};
... Line(11) = {12, 11};
... Line(12) = {11, 10};
...
... // define the three boundaries
...
... Line Loop(1) = {1, 2, 3, 4};
```

(continues on next page)

(continued from previous page)

```

... Line Loop(2) = {5, 6, 7, 8};
... Line Loop(3) = {9, 10, 11, 12};
...
... // define the three domains
...
... Plane Surface(1) = {2, 1};
... Plane Surface(2) = {1, 3};
... Plane Surface(3) = {3};
...
... // label the three domains
...
... // attention: if you use any "Physical" labels, you *must* label
... // all elements that correspond to FiPy Cells (Physical Surface in 2D
... // and Physical Volume in 3D) or Gmsh will not include them and FiPy
... // will not be able to include them in the Mesh.
...
... // note: if you do not use any labels, all Cells will be included.
...
... Physical Surface("Outer") = {1};
... Physical Surface("Middle") = {2};
... Physical Surface("Inner") = {3};
...
... // label the "north-west" part of the exterior boundary
...
... // note: you only need to label the Face elements
... // (Physical Line in 2D and Physical Surface in 3D) that correspond
... // to boundaries you are interested in. FiPy does not need them to
... // construct the Mesh.
...
... Physical Line("NW") = {5};
... ''' % locals()

```

It can be easier to specify certain domains and boundaries within Gmsh than it is to define the same domains and boundaries with FiPy expressions.

Here we compare obtaining the same Cells and Faces using FiPy's parametric descriptions and Gmsh's labels.

```
>>> x, y = squaredCircle.cellCenters
```

```
>>> middle = ((x**2 + y**2 <= radius**2)
...           & ~((x > -side/2) & (x < side/2)
...           & (y > -side/2) & (y < side/2)))
```

```
>>> print((middle == squaredCircle.physicalCells["Middle"]).all())
True
```

```
>>> X, Y = squaredCircle.faceCenters
```

```
>>> NW = ((X**2 + Y**2 > (1.99*radius)**2)
...       & (X**2 + Y**2 < (2.01*radius)**2)
...       & (X <= 0) & (Y >= 0))
```

```
>>> print((NW == squaredCircle.physicalFaces["NW"]).all())
True
```

It is possible to direct Gmsh to give the mesh different densities in different locations

```

>>> geo = '''
... // A mesh consisting of a square
...
... // define the corners of the square
...
... Point(1) = {1, 1, 0, 1};
... Point(2) = {0, 1, 0, 1};
... Point(3) = {0, 0, 0, 1};
... Point(4) = {1, 0, 0, 1};
...
... // define the square
...
... Line(1) = {1, 2};
... Line(2) = {2, 3};
... Line(3) = {3, 4};
... Line(4) = {4, 1};
...
... // define the boundary
...
... Line Loop(1) = {1, 2, 3, 4};
...
... // define the domain
...
... Plane Surface(1) = {1};
... '''

```

```

>>> from fipy import CellVariable, numerix

```

```

>>> error = []
>>> bkg = None
>>> from builtins import range
>>> for refine in range(4):
...     square = Gmsh2D(geo, background=bkg)
...     x, y = square.cellCenters
...     bkg = CellVariable(mesh=square, value=abs(x / 4) + 0.01)
...     error.append(((2 * numerix.sqrt(square.cellVolumes) / bkg - 1)**2) *
... cellVolumeAverage)

```

Check that the mesh is (semi)monotonically approaching the desired density (the first step may increase, depending on the number of partitions)

```

>>> print(numerix.greater(error[:-1], error[1:]).all())
True

```

and that the final density is close enough to the desired density

```

>>> print(error[-1] < 0.02)
True

```

The initial mesh doesn't have to be from Gmsh

```

>>> from fipy import Tri2D

```

```

>>> trisquare = Tri2D(nx=1, ny=1)
>>> x, y = trisquare.cellCenters

```

(continues on next page)

(continued from previous page)

```
>>> bkg = CellVariable(mesh=trisquare, value=abs(x / 4) + 0.01)
>>> std1 = (numerix.sqrt(2 * trisquare.cellVolumes) / bkg).std()
```

```
>>> square = Gmsh2D(geo, background=bkg)
>>> x, y = square.cellCenters
>>> bkg = CellVariable(mesh=square, value=abs(x / 4) + 0.01)
>>> std2 = (numerix.sqrt(2 * square.cellVolumes) / bkg).std()
```

```
>>> print(std1 > std2)
True
```

Parameters

- **arg** (*str*) – (i) the path to an *MSH* file, (ii) a path to a Gmsh geometry (*.geo*) file, or (iii) a Gmsh geometry script
- **coordDimensions** (*int*) – Dimension of shapes
- **overlap** (*int*) – The number of overlapping cells for parallel simulations. Generally 1 is adequate. Higher order equations or discretizations require more. If *overlap* is greater than one, communication reverts to serial, as Gmsh only provides one layer of ghost cells.
- **background** (*CellVariable*) – Specifies the desired characteristic lengths of the mesh cells

```
__init__(arg, coordDimensions=2, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.gmshMesh'
```

```
__setstate__(state)
```

```
class fipy.meshes.gmshMesh.Gmsh2DIn3DSpace(arg, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Bases: *fipy.meshes.gmshMesh.Gmsh2D*

Create a topologically 2D Mesh in 3D coordinates using Gmsh

Parameters

- **arg** (*str*) – (i) the path to an *MSH* file, (ii) a path to a Gmsh geometry (*.geo*) file, or (iii) a Gmsh geometry script
- **coordDimensions** (*int*) – Dimension of shapes
- **overlap** (*int*) – The number of overlapping cells for parallel simulations. Generally 1 is adequate. Higher order equations or discretizations require more. If *overlap* is greater than one, communication reverts to serial, as Gmsh only provides one layer of ghost cells.
- **background** (*CellVariable*) – Specifies the desired characteristic lengths of the mesh cells

```
__init__(arg, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.gmshMesh'
```

```
class fipy.meshes.gmshMesh.Gmsh3D(arg, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Bases: `fipy.meshes.mesh.Mesh`

Create a 3D Mesh using Gmsh

Parameters

- **arg** (*str*) – (i) the path to an *MSH* file, (ii) a path to a Gmsh geometry (*.geo*) file, or (iii) a Gmsh geometry script
- **overlap** (*int*) – The number of overlapping cells for parallel simulations. Generally 1 is adequate. Higher order equations or discretizations require more. If *overlap* is greater than one, communication reverts to serial, as Gmsh only provides one layer of ghost cells.
- **background** (*CellVariable*) – Specifies the desired characteristic lengths of the mesh cells

```
__init__(arg, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.gmshMesh'
```

```
__setstate__(state)
```

```
class fipy.meshes.gmshMesh.GmshGrid2D(dx=1.0, dy=1.0, nx=1, ny=None, coordDimensions=2, communicator=SerialPETScCommWrapper(), overlap=1)
```

Bases: `fipy.meshes.gmshMesh.Gmsh2D`

Should serve as a drop-in replacement for *Grid2D*

```
__init__(dx=1.0, dy=1.0, nx=1, ny=None, coordDimensions=2, communicator=SerialPETScCommWrapper(), overlap=1)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.gmshMesh'
```

```
class fipy.meshes.gmshMesh.GmshGrid3D(dx=1.0, dy=1.0, dz=1.0, nx=1, ny=None, nz=None, communicator=SerialPETScCommWrapper(), overlap=1)
```

Bases: `fipy.meshes.gmshMesh.Gmsh3D`

Should serve as a drop-in replacement for *Grid3D*

```
__init__(dx=1.0, dy=1.0, dz=1.0, nx=1, ny=None, nz=None, communicator=SerialPETScCommWrapper(), overlap=1)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.gmshMesh'
```

30.12 fipy.meshes.grid1D module

30.13 fipy.meshes.grid2D module

30.14 fipy.meshes.grid3D module

30.15 fipy.meshes.mesh module

exception `fipy.meshes.mesh.MeshAdditionError`

Bases: `Exception`

`__module__` = `'fipy.meshes.mesh'`

`__weakref__`

list of weak references to the object (if defined)

class `fipy.meshes.mesh.Mesh` (`vertexCoords`, `faceVertexIDs`, `cellFaceIDs`, `communicator=SerialPETScCommWrapper()`, `_RepresentationClass=<class 'fipy.meshes.representations.meshRepresentation._MeshRepresentation'>`, `_TopologyClass=<class 'fipy.meshes.topologies.meshTopology._MeshTopology'>`)

Bases: `fipy.meshes.abstractMesh.AbstractMesh`

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

`__init__` (`vertexCoords`, `faceVertexIDs`, `cellFaceIDs`, `communicator=SerialPETScCommWrapper()`, `_RepresentationClass=<class 'fipy.meshes.representations.meshRepresentation._MeshRepresentation'>`, `_TopologyClass=<class 'fipy.meshes.topologies.meshTopology._MeshTopology'>`)

Initialize self. See help(type(self)) for accurate signature.

`__module__` = `'fipy.meshes.mesh'`

`__mul__` (`factor`)

Dilate a *Mesh* by *factor*.

```
>>> from fipy.meshes import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print(baseMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5]]
```

The *factor* can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.5  1.5  4.5  4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print(dilatedMesh.cellCenters)
```

(continues on next page)

(continued from previous page)

```
[[ 1.5  4.5  1.5  4.5]
 [ 1.   1.   3.   3. ]]
```

but the vector must have the same dimensionality as the *Mesh*

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

`__rmul__` (*factor*)

Dilate a *Mesh* by *factor*.

```
>>> from fipy.meshes import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print(baseMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5]]
```

The *factor* can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.5  1.5  4.5  4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.   1.   3.   3. ]]
```

but the vector must have the same dimensionality as the *Mesh*

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

30.16 fipy.meshes.mesh1D module

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

```
class fipy.meshes.mesh1D.Mesh1D(vertexCoords, faceVertexIDs, cellFaceIDs, communi-
                                cator=SerialPETScCommWrapper(), _Representation-
                                Class=<class 'fipy.meshes.representations.meshRepresentation._MeshRepresentation'>
                                _TopologyClass=<class 'fipy.meshes.topologies.meshTopology._Mesh1DTopology'>)
```

Bases: *fipy.meshes.mesh.Mesh*

```
__init__(vertexCoords, faceVertexIDs, cellFaceIDs, communi-
        ator=SerialPETScCommWrapper(), _RepresentationClass=<class
        'fipy.meshes.representations.meshRepresentation._MeshRepresentation'>, _Topology-
        Class=<class 'fipy.meshes.topologies.meshTopology._Mesh1DTopology'>)
    Initialize self. See help(type(self)) for accurate signature.
```

```
__module__ = 'fipy.meshes.mesh1D'
```

```
__mul__(factor)
    Dilate a Mesh by factor.
```

```
>>> from fipy.meshes import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print(baseMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5]]
```

The *factor* can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.5  1.5  4.5  4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.   1.   3.   3. ]]
```

but the vector must have the same dimensionality as the *Mesh*

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

30.17 fipy.meshes.mesh2D module

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

```
class fipy.meshes.mesh2D.Mesh2D(vertexCoords, faceVertexIDs, cellFaceIDs, communi-
                                ator=SerialPETScCommWrapper(), _Representation-
                                Class=<class 'fipy.meshes.representations.meshRepresentation._MeshRepresentation'>,
                                _TopologyClass=<class 'fipy.meshes.topologies.meshTopology._Mesh2DTopology'>)
```

Bases: `fipy.meshes.mesh.Mesh`

```
__init__(vertexCoords, faceVertexIDs, cellFaceIDs, communi-
        ator=SerialPETScCommWrapper(), _RepresentationClass=<class
        'fipy.meshes.representations.meshRepresentation._MeshRepresentation'>, _Topology-
        Class=<class 'fipy.meshes.topologies.meshTopology._Mesh2DTopology'>)
    Initialize self. See help(type(self)) for accurate signature.
```

`__module__ = 'fipy.meshes.mesh2D'`

`__mul__(factor)`

Dilate a *Mesh* by *factor*.

```
>>> from fipy.meshes import Grid2D
>>> baseMesh = Grid2D(dx = 1.0, dy = 1.0, nx = 2, ny = 2)
>>> print(baseMesh.cellCenters)
[[ 0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5]]
```

The *factor* can be a scalar

```
>>> dilatedMesh = baseMesh * 3
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.5  1.5  4.5  4.5]]
```

or a vector

```
>>> dilatedMesh = baseMesh * ((3,), (2,))
>>> print(dilatedMesh.cellCenters)
[[ 1.5  4.5  1.5  4.5]
 [ 1.   1.   3.   3. ]]
```

but the vector must have the same dimensionality as the *Mesh*

```
>>> dilatedMesh = baseMesh * ((3,), (2,), (1,))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

extrude (*extrudeFunc*=<function *Mesh2D*.<lambda>>, *layers*=1)

This function returns a new 3D mesh. The 2D mesh is extruded using the *extrudeFunc* and the number of layers.

```
>>> from fipy.meshes.nonUniformGrid2D import NonUniformGrid2D
>>> print(NonUniformGrid2D(nx=2, ny=2).extrude(layers=2).cellCenters)
[[ 0.5  1.5  0.5  1.5  0.5  1.5  0.5  1.5]
 [ 0.5  0.5  1.5  1.5  0.5  0.5  1.5  1.5]
 [ 0.5  0.5  0.5  0.5  1.5  1.5  1.5  1.5]]
```

```
>>> from fipy.meshes.tri2D import Tri2D
>>> print(Tri2D().extrude(layers=2).cellCenters.allclose([[ 0.83333333, 0.5,
↪ 0.16666667, 0.5, 0.83333333, 0.5,
... 0.16666667, 0.5
↪ ],
... [ 0.5, 0.
↪ 83333333, 0.5, 0.16666667, 0.5, 0.83333333,
... 0.5, 0.
↪ 16666667],
... [ 0.5, 0.5,
↪ 0.5, 0.5, 1.5, 1.5, 1.5,
... 1.5
↪ ]]))
True
```

Parameters

- **extrudeFunc** (*function*) – Takes the vertex coordinates and returns the displaced values
- **layers** (*int*) – Number of layers in the extruded mesh (number of times *extrudeFunc* will be called)

30.18 fipy.meshes.nonUniformGrid1D module

1D Mesh

```
class fipy.meshes.nonUniformGrid1D.NonUniformGrid1D(dx=1.0, nx=None, overlap=2,
                                                    communicator=SerialPETScCommWrapper(),
                                                    _BuilderClass=<class
'fipy.meshes.builders.grid1DBuilder._NonuniformGrid1DBuilder'>,
                                                    _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid1DRepresentation'>,
                                                    _TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid1DTopology'>
```

Bases: *fipy.meshes.mesh1D.Mesh1D*

Creates a 1D grid mesh.

```
>>> mesh = NonUniformGrid1D(nx = 3)
>>> print(mesh.cellCenters)
[[ 0.5  1.5  2.5]]
```

```
>>> mesh = NonUniformGrid1D(dx = (1, 2, 3))
>>> print(mesh.cellCenters)
[[ 0.5  2.  4.5]]
```

```
>>> mesh = NonUniformGrid1D(nx = 2, dx = (1, 2, 3))
Traceback (most recent call last):
...
IndexError: nx != len(dx)
```

```
__init__(dx=1.0, nx=None, overlap=2, communicator=SerialPETScCommWrapper(), _Builder-
Class=<class 'fipy.meshes.builders.grid1DBuilder._NonuniformGrid1DBuilder'>, _Repre-
sentationClass=<class 'fipy.meshes.representations.gridRepresentation._Grid1DRepresentation'>,
_TopologyClass=<class 'fipy.meshes.topologies.gridTopology._Grid1DTopology'>)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.meshes.nonUniformGrid1D'
```

30.19 fipy.meshes.nonUniformGrid2D module

2D rectangular Mesh

```
class fipy.meshes.nonUniformGrid2D.NonUniformGrid2D(dx=1.0,          dy=1.0,
                                                    nx=None,          ny=None,
                                                    overlap=2,        communi-
                                                    tor=SerialPETScCommWrapper(),
                                                    _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid2D
_TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid2DTopology'>
```

Bases: *fipy.meshes.mesh2D.Mesh2D*

Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces.

```
__init__(dx=1.0,      dy=1.0,      nx=None,      ny=None,      overlap=2,      communi-
        cator=SerialPETScCommWrapper(),          _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>,      _Topology-
        Class=<class 'fipy.meshes.topologies.gridTopology._Grid2DTopology'> )
    Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.meshes.nonUniformGrid2D'
```

30.20 fipy.meshes.nonUniformGrid3D module

```
class fipy.meshes.nonUniformGrid3D.NonUniformGrid3D(dx=1.0,      dy=1.0,      dz=1.0,
                                                    nx=None, ny=None, nz=None,
                                                    overlap=2,        communi-
                                                    tor=SerialPETScCommWrapper(),
                                                    _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid3D
_TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid3DTopology'>
```

Bases: *fipy.meshes.mesh.Mesh*

3D rectangular-prism Mesh

X axis runs from left to right. Y axis runs from bottom to top. Z axis runs from front to back.

Numbering System:

Vertices: Numbered in the usual way. X coordinate changes most quickly, then Y, then Z.

Cells: Same numbering system as vertices.

Faces: XY faces numbered first, then XZ faces, then YZ faces. Within each subcategory, it is numbered in the usual way.

```
__init__(dx=1.0,      dy=1.0,      dz=1.0,      nx=None,      ny=None,      nz=None,      overlap=2,      com-
        municator=SerialPETScCommWrapper(),          _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid3DRepresentation'>,      _Topology-
        Class=<class 'fipy.meshes.topologies.gridTopology._Grid3DTopology'> )
    Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.meshes.nonUniformGrid3D'
```

30.21 fipy.meshes.periodicGrid1D module

Periodic 1D Mesh

class fipy.meshes.periodicGrid1D.**PeriodicGrid1D** (*dx=1.0, nx=None, overlap=2, *args, **kwargs*)

Bases: *fipy.meshes.nonUniformGrid1D.NonUniformGrid1D*

Creates a Periodic grid mesh.

```
>>> mesh = PeriodicGrid1D(dx = (1, 2, 3))
```

```
>>> print (numerix.allclose (numerix.nonzero (mesh.exteriorFaces) [0],
...                               [3]))
True
```

```
>>> print (numerix.allclose (mesh.faceCellIDs.filled (-999),
...                               [[2, 0, 1, 2],
...                               [0, 1, 2, -999]]))
True
```

```
>>> print (numerix.allclose (mesh._cellDistances,
...                               [ 2., 1.5, 2.5, 1.5]))
True
```

```
>>> print (numerix.allclose (mesh._cellToCellDistances,
...                               [[ 2., 1.5, 2.5],
...                               [ 1.5, 2.5, 2. ]]))
True
```

```
>>> print (numerix.allclose (mesh.faceNormals,
...                               [[ 1., 1., 1., 1.]])
True
```

```
>>> print (numerix.allclose (mesh._cellVertexIDs,
...                               [[1, 2, 2],
...                               [0, 1, 0]]))
True
```

__init__ (*dx=1.0, nx=None, overlap=2, *args, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.meshes.periodicGrid1D'

property cellCenters

Defined outside of a geometry class since we need the *CellVariable* version of *cellCenters*; that is, the *cellCenters* defined in *fipy.meshes.mesh* and not in any geometry (since a *CellVariable* requires a reference to a mesh).

30.22 fipy.meshes.periodicGrid2D module

2D periodic rectangular Mesh

```
class fipy.meshes.periodicGrid2D.PeriodicGrid2D (dx=1.0,      dy=1.0,      nx=None,
                                                ny=None,      overlap=2,  communica-
                                                tor=SerialPETScCommWrapper(),
                                                *args, **kwargs)
```

Bases: fipy.meshes.periodicGrid2D._BasePeriodicGrid2D

Creates a periodic 2D grid mesh with horizontal faces numbered first and then vertical faces. Vertices and cells are numbered in the usual way.

```
>>> from fipy import numerix
```

```
>>> mesh = PeriodicGrid2D(dx = 1., dy = 0.5, nx = 2, ny = 2)
```

```
>>> print(numerix.allclose(numerix.nonzero(mesh.exteriorFaces)[0],
...                        [ 4, 5, 8, 11]))
True
```

```
>>> print(numerix.allclose(mesh.faceCellIDs.filled(-1),
...                        [[2, 3, 0, 1, 2, 3, 1, 0, 1, 3, 2, 3],
...                         [0, 1, 2, 3, -1, -1, 0, 1, -1, 2, 3, -1]]))
True
```

```
>>> print(numerix.allclose(mesh._cellDistances,
...                        [ 0.5, 0.5, 0.5, 0.5, 0.25, 0.25, 1., 1., 0.5, 1., 1.,
...                          ↪0.5]))
True
```

```
>>> print(numerix.allclose(mesh.cellFaceIDs,
...                        [[0, 1, 2, 3],
...                         [7, 6, 10, 9],
...                         [2, 3, 0, 1],
...                         [6, 7, 9, 10]]))
True
```

```
>>> print(numerix.allclose(mesh._cellToCellDistances,
...                        [[ 0.5, 0.5, 0.5, 0.5],
...                         [ 1., 1., 1., 1. ],
...                         [ 0.5, 0.5, 0.5, 0.5],
...                         [ 1., 1., 1., 1. ]]))
True
```

```
>>> normals = [[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
...            [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]]
```

```
>>> print(numerix.allclose(mesh.faceNormals, normals))
True
```

```
>>> print(numerix.allclose(mesh._cellVertexIDs,
...                        [[4, 5, 7, 8],
...                         [3, 4, 6, 7],
```

(continues on next page)

(continued from previous page)

```

...             [1, 2, 4, 5],
...             [0, 1, 3, 4]])
True

```

```
__module__ = 'fipy.meshes.periodicGrid2D'
```

```

class fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight(dx=1.0,      dy=1.0,
                                                         nx=None,    ny=None,
                                                         overlap=2,  communica-
                                                         tor=SerialPETScCommWrapper(),
                                                         *args, **kwargs)

```

Bases: fipy.meshes.periodicGrid2D._BasePeriodicGrid2D

```
__module__ = 'fipy.meshes.periodicGrid2D'
```

```

class fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom(dx=1.0,      dy=1.0,
                                                         nx=None,    ny=None,
                                                         overlap=2,  communica-
                                                         tor=SerialPETScCommWrapper(),
                                                         *args, **kwargs)

```

Bases: fipy.meshes.periodicGrid2D._BasePeriodicGrid2D

```
__module__ = 'fipy.meshes.periodicGrid2D'
```

30.23 fipy.meshes.periodicGrid3D module

3D periodic rectangular Mesh

```

class fipy.meshes.periodicGrid3D.PeriodicGrid3D(dx=1.0,      dy=1.0,      dz=1.0,
                                                  nx=None,    ny=None,    nz=None,
                                                  overlap=2,  communica-
                                                  tor=SerialPETScCommWrapper(),
                                                  *args, **kwargs)

```

Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D

Creates a periodic 3D grid mesh with horizontal faces numbered first and then vertical faces. Vertices and cells are numbered in the usual way.

```
>>> from fipy import numerix
```

```

>>> mesh = PeriodicGrid3D(dx=1., dy=0.5, dz=2., nx=2, ny=2, nz=1)
>>> print(numerix.allclose(numerix.nonzero(mesh.exteriorFaces)[0],
...                        [4, 5, 6, 7, 12, 13, 16, 19]))
...
True

```

```

>>> print(numerix.allclose(mesh.faceCellIDs.filled(-1),
...                        [[0, 1, 2, 3, 0, 1, 2, 3, 2, 3,
...                        0, 1, 2, 3, 1, 0, 1, 3, 2, 3],
...                        [0, 1, 2, 3, -1, -1, -1, -1, 0, 1,
...                        2, 3, -1, -1, 0, 1, -1, 2, 3, -1]]))
...
True

```

```

>>> print(numerix.allclose(mesh._cellDistances,
...                        [2., 2., 2., 2., 1., 1., 1., 1., 0.5, 0.5,
...

```

(continues on next page)

(continued from previous page)

```
...             0.5, 0.5, 0.25, 0.25, 1., 1., 0.5, 1., 1., 0.5]))
True
```

```
>>> print (numerix.allclose(mesh.cellFaceIDs,
...             [[14, 15, 17, 18],
...             [15, 14, 18, 17],
...             [8, 9, 10, 11],
...             [10, 11, 8, 9],
...             [0, 1, 2, 3],
...             [0, 1, 2, 3]]))
True
```

```
>>> print (numerix.allclose(mesh._cellToCellDistances,
...             [[1., 1., 1., 1.],
...             [1., 1., 1., 1.],
...             [0.5, 0.5, 0.5, 0.5],
...             [0.5, 0.5, 0.5, 0.5],
...             [2., 2., 2., 2.],
...             [2., 2., 2., 2.]])
True
```

```
>>> normals = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
...            [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
...            [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

```
>>> print (numerix.allclose(mesh.faceNormals, normals))
True
```

```
>>> print (numerix.allclose(mesh._cellVertexIDs,
...             [[13, 14, 16, 17],
...             [12, 13, 15, 16],
...             [10, 11, 13, 14],
...             [9, 10, 12, 13],
...             [4, 5, 7, 8],
...             [3, 4, 6, 7],
...             [1, 2, 4, 5],
...             [0, 1, 3, 4]]))
True
```

```
__module__ = 'fipy.meshes.periodicGrid3D'
```

```
class fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight (dx=1.0,      dy=1.0,
                                                         dz=1.0,      nx=None,
                                                         ny=None,      nz=None,
                                                         overlap=2,  communicator=SerialPETScCommWrapper(),
                                                         *args, **kwargs)
```

```
Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
```

```
__module__ = 'fipy.meshes.periodicGrid3D'
```

```

class fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom(dx=1.0,      dy=1.0,
                                                         dz=1.0,      nx=None,
                                                         ny=None,      nz=None,
                                                         overlap=2,  communica-
                                                         tor=SerialPETScCommWrapper(),
                                                         *args, **kwargs)

    Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
    __module__ = 'fipy.meshes.periodicGrid3D'

class fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack(dx=1.0,      dy=1.0,
                                                         dz=1.0,      nx=None,
                                                         ny=None,      nz=None,
                                                         overlap=2,  communica-
                                                         tor=SerialPETScCommWrapper(),
                                                         *args, **kwargs)

    Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
    __module__ = 'fipy.meshes.periodicGrid3D'

class fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom(dx=1.0,
                                                                    dy=1.0,
                                                                    dz=1.0,
                                                                    nx=None,
                                                                    ny=None,
                                                                    nz=None,
                                                                    overlap=2,
                                                                    commu-
                                                                    nica-
                                                                    tor=SerialPETScCommWrapper(),
                                                                    *args,
                                                                    **kwargs)

    Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
    __module__ = 'fipy.meshes.periodicGrid3D'

class fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack(dx=1.0,
                                                                    dy=1.0,
                                                                    dz=1.0,
                                                                    nx=None,
                                                                    ny=None,
                                                                    nz=None,
                                                                    overlap=2,
                                                                    commu-
                                                                    nica-
                                                                    tor=SerialPETScCommWrapper(),
                                                                    *args,
                                                                    **kwargs)

    Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
    __module__ = 'fipy.meshes.periodicGrid3D'

```

```
class fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack(dx=1.0,
                                                                    dy=1.0,
                                                                    dz=1.0,
                                                                    nx=None,
                                                                    ny=None,
                                                                    nz=None,
                                                                    overlap=2,
                                                                    commu-
                                                                    nica-
                                                                    tor=SerialPETScCommWrapper(),
                                                                    *args,
                                                                    **kwargs)

Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D

__module__ = 'fipy.meshes.periodicGrid3D'
```

30.24 fipy.meshes.skewedGrid2D module

```
class fipy.meshes.skewedGrid2D.SkewedGrid2D(dx=1.0, dy=1.0, nx=None, ny=1, rand=0,
                                              *args, **kwargs)
```

Bases: *fipy.meshes.mesh2D.Mesh2D*

Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces. The points are skewed by a random amount (between *rand* and *-rand*) in the X and Y directions.

Note: This *Mesh* only operates in serial

```
__init__(dx=1.0, dy=1.0, nx=None, ny=1, rand=0, *args, **kwargs)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.skewedGrid2D'
```

property physicalShape

Return physical dimensions of *Grid2D*.

property shape

30.25 fipy.meshes.test module

Test implementation of the mesh

30.26 fipy.meshes.tri2D module

```
class fipy.meshes.tri2D.Tri2D(dx=1.0, dy=1.0, nx=1, ny=1, _RepresentationClass=<class
                                                                    'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>,
                                                                    _TopologyClass=<class 'fipy.meshes.topologies.meshTopology._Mesh2DTopology'>)
```

Bases: *fipy.meshes.mesh2D.Mesh2D*

This class creates a mesh made out of triangles. It does this by starting with a standard Cartesian mesh (*Grid2D*) and dividing each cell in that mesh (hereafter referred to as a “box”) into four equal parts with the dividing lines being the diagonals.

Creates a 2D triangular mesh with horizontal faces numbered first then vertical faces, then diagonal faces. Vertices are numbered starting with the vertices at the corners of boxes and then the vertices at the centers of boxes. Cells on the right of boxes are numbered first, then cells on the top of boxes, then cells on the left of boxes, then cells on the bottom of boxes. Within each of the “sub-categories” in the above, the vertices, cells and faces are numbered in the usual way.

Parameters

- **dx, dy** (*float*) – The X and Y dimensions of each “box”. If $dx \neq dy$, the line segments connecting the cell centers will not be orthogonal to the faces.
- **nx, ny** (*int*) – The number of boxes in the X direction and the Y direction. The total number of boxes will be equal to $nx * ny$, and the total number of cells will be equal to $4 * nx * ny$.

```
__init__(dx=1.0, dy=1.0, nx=1, ny=1, _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>, _Topology-
Class=<class 'fipy.meshes.topologies.meshTopology._Mesh2DTopology'>)
```

Creates a 2D triangular mesh with horizontal faces numbered first then vertical faces, then diagonal faces. Vertices are numbered starting with the vertices at the corners of boxes and then the vertices at the centers of boxes. Cells on the right of boxes are numbered first, then cells on the top of boxes, then cells on the left of boxes, then cells on the bottom of boxes. Within each of the “sub-categories” in the above, the vertices, cells and faces are numbered in the usual way.

Parameters

- **dx, dy** (*float*) – The X and Y dimensions of each “box”. If $dx \neq dy$, the line segments connecting the cell centers will not be orthogonal to the faces.
- **nx, ny** (*int*) – The number of boxes in the X direction and the Y direction. The total number of boxes will be equal to $nx * ny$, and the total number of cells will be equal to $4 * nx * ny$.

```
__module__ = 'fipy.meshes.tri2D'
```

property physicalShape

Return physical dimensions of *Grid2D*.

property shape

30.27 fipy.meshes.uniformGrid module

```
class fipy.meshes.uniformGrid.UniformGrid(communicator, _RepresentationClass=<class
'fipy.meshes.representations.abstractRepresentation._AbstractRepresenta
_TopologyClass=<class
'fipy.meshes.topologies.abstractTopology._AbstractTopology'>)
```

Bases: *fipy.meshes.abstractMesh.AbstractMesh*

Wrapped scaled geometry properties

```
__module__ = 'fipy.meshes.uniformGrid'
```

30.28 fipy.meshes.uniformGrid1D module

1D Mesh

```
class fipy.meshes.uniformGrid1D.UniformGrid1D(dx=1.0, nx=1, origin=(0,
), overlap=2, communicator=SerialPETScCommWrapper(),
_RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid1DRepresent
_TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid1DTopology'>)
```

Bases: *fipy.meshes.uniformGrid.UniformGrid*

Creates a 1D grid mesh.

```
>>> mesh = UniformGrid1D(nx = 3)
>>> print(mesh.cellCenters)
[[ 0.5  1.5  2.5]]
```

```
__init__(dx=1.0, nx=1, origin=(0, ), overlap=2, communica-
tor=SerialPETScCommWrapper(), _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid1DRepresentation'>, _Topology-
Class=<class 'fipy.meshes.topologies.gridTopology._Grid1DTopology'>)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.uniformGrid1D'
```

```
__mul__(factor)
```

property exteriorFaces

Geometry set and calc

property faceCellIDs

property faceNormals

property vertexCoords

30.29 fipy.meshes.uniformGrid2D module

2D rectangular Mesh with constant spacing in x and constant spacing in y

```
class fipy.meshes.uniformGrid2D.UniformGrid2D(dx=1.0, dy=1.0, nx=1, ny=1, ori-
gin=((0, ), (0, )), overlap=2, communi-
cator=SerialPETScCommWrapper(),
_RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid2DRepresent
_TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid2DTopology'>)
```

Bases: *fipy.meshes.uniformGrid.UniformGrid*

Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces.

```
__init__(dx=1.0, dy=1.0, nx=1, ny=1, origin=((0, ), (0, )), overlap=2, com-
municator=SerialPETScCommWrapper(), _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>, _Topology-
Class=<class 'fipy.meshes.topologies.gridTopology._Grid2DTopology'>)
```

Initialize self. See help(type(self)) for accurate signature.

```

__module__ = 'fipy.meshes.uniformGrid2D'
__mul__ (factor)
property faceCellIDs
property faceNormals
property faceVertexIDs
property vertexCoords

```

30.30 fipy.meshes.uniformGrid3D module

```

class fipy.meshes.uniformGrid3D.UniformGrid3D(dx=1.0, dy=1.0, dz=1.0, nx=1,
                                              ny=1, nz=1, origin=[[0], [0], [0]],
                                              overlap=2, communicator=SerialPETScCommWrapper(),
                                              _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid3DRepresentation'>,
                                              _TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid3DTopology'> )

```

Bases: *fipy.meshes.uniformGrid.UniformGrid*

3D rectangular-prism Mesh with uniform grid spacing in each dimension.

X axis runs from left to right. Y axis runs from bottom to top. Z axis runs from front to back.

Numbering System:

Vertices: Numbered in the usual way. X coordinate changes most quickly, then Y, then Z.

*** arrays are arranged Z, Y, X because in numerix, the final index is the one that changes the most quickly ***

Cells: Same numbering system as vertices.

Faces: XY faces numbered first, then XZ faces, then YZ faces. Within each subcategory, it is numbered in the usual way.

```

__init__(dx=1.0, dy=1.0, dz=1.0, nx=1, ny=1, nz=1, origin=[[0], [0], [0]],
         overlap=2, communicator=SerialPETScCommWrapper(), _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid3DRepresentation'>, _Topology-
         Class=<class 'fipy.meshes.topologies.gridTopology._Grid3DTopology'> )
    Initialize self. See help(type(self)) for accurate signature.

```

```

__module__ = 'fipy.meshes.uniformGrid3D'
__mul__ (factor)
property faceCellIDs
property faceNormals
property faceVertexIDs
property vertexCoords

```

30.31 Module contents

`fipy.meshes.Grid3D(dx=1.0, dy=1.0, dz=1.0, nx=None, ny=None, nz=None, Lx=None, Ly=None, Lz=None, overlap=2, communicator=SerialPETScCommWrapper())`

Factory function to select between *UniformGrid3D* and *NonUniformGrid3D*. If $L\{x,y,z\}$ is specified, the length of the domain is always $L\{x,y,z\}$ regardless of $d\{x,y,z\}$, unless $d\{x,y,z\}$ is a list of spacings, in which case $L\{x,y,z\}$ will be the sum of $d\{x,y,z\}$ and $n\{x,y,z\}$ will be the count of $d\{x,y,z\}$.

Parameters

- **dx** (*float*) – Grid spacing in the horizontal direction
- **dy** (*float*) – Grid spacing in the vertical direction
- **dz** (*float*) – Grid spacing in the depth direction
- **nx** (*int*) – Number of cells in the horizontal direction
- **ny** (*int*) – Number of cells in the vertical direction
- **nz** (*int*) – Number of cells in the depth direction
- **Lx** (*float*) – Domain length in the horizontal direction
- **Ly** (*float*) – Domain length in the vertical direction
- **Lz** (*float*) – Domain length in the depth direction
- **overlap** (*int*) – Number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, `fipy.tools.serialComm` or `fipy.tools.parallelComm`. Select `~fipy.tools.serialComm` to create a serial mesh when running in parallel; mostly used for test purposes.

`fipy.meshes.Grid2D(dx=1.0, dy=1.0, nx=None, ny=None, Lx=None, Ly=None, overlap=2, communicator=SerialPETScCommWrapper())`

Factory function to select between *UniformGrid2D* and *NonUniformGrid2D*. If $L\{x,y\}$ is specified, the length of the domain is always $L\{x,y\}$ regardless of $d\{x,y\}$, unless $d\{x,y\}$ is a list of spacings, in which case $L\{x,y\}$ will be the sum of $d\{x,y\}$ and $n\{x,y\}$ will be the count of $d\{x,y\}$.

```
>>> print(Grid2D(Lx=3., nx=2).dx)
1.5
```

Parameters

- **dx** (*float*) – Grid spacing in the horizontal direction
- **dy** (*float*) – Grid spacing in the vertical direction
- **nx** (*int*) – Number of cells in the horizontal direction
- **ny** (*int*) – Number of cells in the vertical direction
- **Lx** (*float*) – Domain length in the horizontal direction
- **Ly** (*float*) – Domain length in the vertical direction
- **overlap** (*int*) – Number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, `fipy.tools.serialComm` or `fipy.tools.parallelComm`. Select `~fipy.tools.serialComm` to create a serial mesh when running in parallel; mostly used for test purposes.

```
fipy.meshes.Grid1D(dx=1.0, nx=None, Lx=None, overlap=2, communicator=SerialPETScCommWrapper())
```

Factory function to select between *UniformGrid1D* and *NonUniformGrid1D*. If *Lx* is specified the length of the domain is always *Lx* regardless of *dx*, unless *dx* is a list of spacings, in which case *Lx* will be the sum of *dx* and *nx* will be the count of *dx*.

Parameters

- **dx** (*float*) – Grid spacing in the horizontal direction
- **nx** (*int*) – Number of cells in the horizontal direction
- **Lx** (*float*) – Domain length in the horizontal direction
- **overlap** (*int*) – Number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, *fipy.tools.serialComm* or *fipy.tools.parallelComm*. Select *~fipy.tools.serialComm* to create a serial mesh when running in parallel; mostly used for test purposes.

```
fipy.meshes.CylindricalGrid2D(dr=None, dz=None, nr=None, nz=None, Lr=None, Lz=None, dx=1.0, dy=1.0, nx=None, ny=None, Lx=None, Ly=None, origin=0, 0, overlap=2, communicator=SerialPETScCommWrapper())
```

Factory function to select between *CylindricalUniformGrid2D* and *CylindricalNonUniformGrid2D*. If *Lr* is specified the length of the domain is always *Lr* regardless of *dr*, unless *dr* is a list of spacings, in which case *Lr* will be the sum of *dr*.

Parameters

- **dr** (*float*) – Grid spacing in the radial direction. Alternative: *dx*.
- **dz** (*float*) – grid spacing in the vertical direction. Alternative: *dy*.
- **nr** (*int*) – Number of cells in the radial direction. Alternative: *nx*.
- **nz** (*int*) – Number of cells in the vertical direction. Alternative: *ny*.
- **Lr** (*float*) – Domain length in the radial direction. Alternative: *Lx*.
- **Lz** (*float*) – Domain length in the vertical direction. Alternative: *Ly*.
- **overlap** (*int*) – the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, *fipy.tools.serialComm* or *fipy.tools.parallelComm*. Select *~fipy.tools.serialComm* to create a serial mesh when running in parallel; mostly used for test purposes.

```
fipy.meshes.CylindricalGrid1D(dr=None, nr=None, Lr=None, dx=1.0, nx=None, Lx=None, origin=0, overlap=2, communicator=SerialPETScCommWrapper())
```

Factory function to select between *CylindricalUniformGrid1D* and *CylindricalNonUniformGrid1D*. If *Lr* is specified the length of the domain is always *Lr* regardless of *dr*, unless *dr* is a list of spacings, in which case *Lr* will be the sum of *dr*.

Parameters

- **dr** (*float*) – Grid spacing in the radial direction. Alternative: *dx*.
- **nr** (*int*) – Number of cells in the radial direction. Alternative: *nx*.
- **Lr** (*float*) – Domain length in the radial direction. Alternative: *Lx*.

- **overlap** (*int*) – the number of overlapping cells for parallel simulations. Generally 2 is adequate. Higher order equations or discretizations require more.
- **communicator** (*CommWrapper*) – Generally, *fipy.tools.serialComm* or *fipy.tools.parallelComm*. Select *~fipy.tools.serialComm* to create a serial mesh when running in parallel; mostly used for test purposes.

class *fipy.meshes.PeriodicGrid1D* (*dx=1.0, nx=None, overlap=2, *args, **kwargs*)

Bases: *fipy.meshes.nonUniformGrid1D.NonUniformGrid1D*

Creates a Periodic grid mesh.

```
>>> mesh = PeriodicGrid1D(dx = (1, 2, 3))
```

```
>>> print (numerix.allclose(numerix.nonzero(mesh.exteriorFaces)[0],
...                          [3]))
True
```

```
>>> print (numerix.allclose(mesh.faceCellIDs.filled(-999),
...                          [[2, 0, 1, 2],
...                           [0, 1, 2, -999]]))
True
```

```
>>> print (numerix.allclose(mesh._cellDistances,
...                          [ 2., 1.5, 2.5, 1.5]))
True
```

```
>>> print (numerix.allclose(mesh._cellToCellDistances,
...                          [[ 2., 1.5, 2.5],
...                           [ 1.5, 2.5, 2. ]]))
True
```

```
>>> print (numerix.allclose(mesh.faceNormals,
...                          [[ 1., 1., 1., 1.]])
True
```

```
>>> print (numerix.allclose(mesh._cellVertexIDs,
...                          [[1, 2, 2],
...                           [0, 1, 0]]))
True
```

__init__ (*dx=1.0, nx=None, overlap=2, *args, **kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = **'fipy.meshes.periodicGrid1D'**

property cellCenters

Defined outside of a geometry class since we need the *CellVariable* version of *cellCenters*; that is, the *cellCenters* defined in *fipy.meshes.mesh* and not in any geometry (since a *CellVariable* requires a reference to a mesh).

class *fipy.meshes.PeriodicGrid2D* (*dx=1.0, dy=1.0, nx=None, ny=None, overlap=2, communicator=SerialPETScCommWrapper(), *args, **kwargs*)

Bases: *fipy.meshes.periodicGrid2D._BasePeriodicGrid2D*

Creates a periodic 2D grid mesh with horizontal faces numbered first and then vertical faces. Vertices and cells are numbered in the usual way.

```
>>> from fipy import numerix
```

```
>>> mesh = PeriodicGrid2D(dx = 1., dy = 0.5, nx = 2, ny = 2)
```

```
>>> print(numerix.allclose(numerix.nonzero(mesh.exteriorFaces)[0],
...                        [ 4, 5, 8, 11]))
True
```

```
>>> print(numerix.allclose(mesh.faceCellIDs.filled(-1),
...                        [[2, 3, 0, 1, 2, 3, 1, 0, 1, 3, 2, 3],
...                        [0, 1, 2, 3, -1, -1, 0, 1, -1, 2, 3, -1]]))
True
```

```
>>> print(numerix.allclose(mesh._cellDistances,
...                        [ 0.5, 0.5, 0.5, 0.5, 0.25, 0.25, 1., 1., 0.5, 1., 1.,
↪0.5]))
True
```

```
>>> print(numerix.allclose(mesh.cellFaceIDs,
...                        [[0, 1, 2, 3],
...                        [7, 6, 10, 9],
...                        [2, 3, 0, 1],
...                        [6, 7, 9, 10]]))
True
```

```
>>> print(numerix.allclose(mesh._cellToCellDistances,
...                        [[ 0.5, 0.5, 0.5, 0.5],
...                        [ 1., 1., 1., 1. ],
...                        [ 0.5, 0.5, 0.5, 0.5],
...                        [ 1., 1., 1., 1. ]]))
True
```

```
>>> normals = [[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
...            [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0]]
```

```
>>> print(numerix.allclose(mesh.faceNormals, normals))
True
```

```
>>> print(numerix.allclose(mesh._cellVertexIDs,
...                        [[4, 5, 7, 8],
...                        [3, 4, 6, 7],
...                        [1, 2, 4, 5],
...                        [0, 1, 3, 4]]))
True
```

```
__module__ = 'fipy.meshes.periodicGrid2D'
```

```
class fipy.meshes.PeriodicGrid2DLeftRight(dx=1.0,          dy=1.0,          nx=None,
                                          ny=None,         overlap=2,         communica-
                                          tor=SerialPETScCommWrapper(),      *args,
                                          **kwargs)
```

```
Bases: fipy.meshes.periodicGrid2D._BasePeriodicGrid2D
```

```
__module__ = 'fipy.meshes.periodicGrid2D'
```

```
class fipy.meshes.PeriodicGrid2DTopBottom(dx=1.0,          dy=1.0,          nx=None,
                                          ny=None,          overlap=2,        communica-
                                          tor=SerialPETScCommWrapper(),    *args,
                                          **kwargs)
```

Bases: fipy.meshes.periodicGrid2D._BasePeriodicGrid2D

```
__module__ = 'fipy.meshes.periodicGrid2D'
```

```
class fipy.meshes.PeriodicGrid3D(dx=1.0, dy=1.0, dz=1.0, nx=None, ny=None, nz=None, over-
                                lap=2, communicator=SerialPETScCommWrapper(), *args,
                                **kwargs)
```

Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D

Creates a periodic 3D grid mesh with horizontal faces numbered first and then vertical faces. Vertices and cells are numbered in the usual way.

```
>>> from fipy import numerix
```

```
>>> mesh = PeriodicGrid3D(dx=1., dy=0.5, dz=2., nx=2, ny=2, nz=1)
>>> print(numerix.allclose(numerix.nonzero(mesh.exteriorFaces)[0],
...                        [4, 5, 6, 7, 12, 13, 16, 19]))
True
```

```
>>> print(numerix.allclose(mesh.faceCellIDs.filled(-1),
...                        [[0, 1, 2, 3, 0, 1, 2, 3, 2, 3,
...                          0, 1, 2, 3, 1, 0, 1, 3, 2, 3],
...                         [0, 1, 2, 3, -1, -1, -1, -1, 0, 1,
...                          2, 3, -1, -1, 0, 1, -1, 2, 3, -1]]))
True
```

```
>>> print(numerix.allclose(mesh._cellDistances,
...                        [2., 2., 2., 2., 1., 1., 1., 1., 0.5, 0.5,
...                         0.5, 0.5, 0.25, 0.25, 1., 1., 0.5, 1., 1., 0.5]))
True
```

```
>>> print(numerix.allclose(mesh.cellFaceIDs,
...                        [[14, 15, 17, 18],
...                         [15, 14, 18, 17],
...                         [8, 9, 10, 11],
...                         [10, 11, 8, 9],
...                         [0, 1, 2, 3],
...                         [0, 1, 2, 3]]))
True
```

```
>>> print(numerix.allclose(mesh._cellToCellDistances,
...                        [[1., 1., 1., 1.],
...                         [1., 1., 1., 1.],
...                         [0.5, 0.5, 0.5, 0.5],
...                         [0.5, 0.5, 0.5, 0.5],
...                         [2., 2., 2., 2.],
...                         [2., 2., 2., 2.]])
True
```

```
>>> normals = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
...            [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
...            [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```



```
>>> print (numerix.allclose(mesh.faceNormals, normals))
True
```

```
>>> print (numerix.allclose(mesh._cellVertexIDs,
...                          [[13, 14, 16, 17],
...                          [12, 13, 15, 16],
...                          [10, 11, 13, 14],
...                          [9, 10, 12, 13],
...                          [4, 5, 7, 8],
...                          [3, 4, 6, 7],
...                          [1, 2, 4, 5],
...                          [0, 1, 3, 4]]))
True
```

```
__module__ = 'fipy.meshes.periodicGrid3D'
```

```
class fipy.meshes.PeriodicGrid3DLeftRight(dx=1.0, dy=1.0, dz=1.0, nx=None,
                                          ny=None, nz=None, overlap=2, communi-
                                          cator=SerialPETScCommWrapper(), *args,
                                          **kwargs)
```

```
Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
```

```
__module__ = 'fipy.meshes.periodicGrid3D'
```

```
class fipy.meshes.PeriodicGrid3DTopBottom(dx=1.0, dy=1.0, dz=1.0, nx=None,
                                          ny=None, nz=None, overlap=2, communi-
                                          cator=SerialPETScCommWrapper(), *args,
                                          **kwargs)
```

```
Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
```

```
__module__ = 'fipy.meshes.periodicGrid3D'
```

```
class fipy.meshes.PeriodicGrid3DFrontBack(dx=1.0, dy=1.0, dz=1.0, nx=None,
                                          ny=None, nz=None, overlap=2, communi-
                                          cator=SerialPETScCommWrapper(), *args,
                                          **kwargs)
```

```
Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
```

```
__module__ = 'fipy.meshes.periodicGrid3D'
```

```
class fipy.meshes.PeriodicGrid3DLeftRightTopBottom(dx=1.0, dy=1.0, dz=1.0,
                                                    nx=None, ny=None, nz=None,
                                                    overlap=2, communi-
                                                    cator=SerialPETScCommWrapper(),
                                                    *args, **kwargs)
```

```
Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
```

```
__module__ = 'fipy.meshes.periodicGrid3D'
```

```
class fipy.meshes.PeriodicGrid3DLeftRightFrontBack(dx=1.0, dy=1.0, dz=1.0,
                                                    nx=None, ny=None, nz=None,
                                                    overlap=2, communi-
                                                    cator=SerialPETScCommWrapper(),
                                                    *args, **kwargs)
```

```
Bases: fipy.meshes.periodicGrid3D._BasePeriodicGrid3D
```

```
__module__ = 'fipy.meshes.periodicGrid3D'
```

```
class fipy.meshes.PeriodicGrid3DTopBottomFrontBack(dx=1.0, dy=1.0, dz=1.0,
                                                    nx=None, ny=None, nz=None,
                                                    overlap=2, communicator=SerialPETScCommWrapper(),
                                                    *args, **kwargs)
```

Bases: `fipy.meshes.periodicGrid3D._BasePeriodicGrid3D`

```
__module__ = 'fipy.meshes.periodicGrid3D'
```

```
class fipy.meshes.SkewedGrid2D(dx=1.0, dy=1.0, nx=None, ny=1, rand=0, *args, **kwargs)
```

Bases: `fipy.meshes.mesh2D.Mesh2D`

Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces. The points are skewed by a random amount (between *rand* and *-rand*) in the X and Y directions.

Note: This *Mesh* only operates in serial

```
__init__(dx=1.0, dy=1.0, nx=None, ny=1, rand=0, *args, **kwargs)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.skewedGrid2D'
```

property physicalShape

Return physical dimensions of *Grid2D*.

property shape

```
class fipy.meshes.Tri2D(dx=1.0, dy=1.0, nx=1, ny=1, _RepresentationClass=<class
    'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>,
    _TopologyClass=<class 'fipy.meshes.topologies.meshTopology._Mesh2DTopology'>)
```

Bases: `fipy.meshes.mesh2D.Mesh2D`

This class creates a mesh made out of triangles. It does this by starting with a standard Cartesian mesh (*Grid2D*) and dividing each cell in that mesh (hereafter referred to as a “box”) into four equal parts with the dividing lines being the diagonals.

Creates a 2D triangular mesh with horizontal faces numbered first then vertical faces, then diagonal faces. Vertices are numbered starting with the vertices at the corners of boxes and then the vertices at the centers of boxes. Cells on the right of boxes are numbered first, then cells on the top of boxes, then cells on the left of boxes, then cells on the bottom of boxes. Within each of the “sub-categories” in the above, the vertices, cells and faces are numbered in the usual way.

Parameters

- **dx, dy** (*float*) – The X and Y dimensions of each “box”. If $dx \neq dy$, the line segments connecting the cell centers will not be orthogonal to the faces.
- **nx, ny** (*int*) – The number of boxes in the X direction and the Y direction. The total number of boxes will be equal to $nx * ny$, and the total number of cells will be equal to $4 * nx * ny$.

```
__init__(dx=1.0, dy=1.0, nx=1, ny=1, _RepresentationClass=<class
    'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>,
    _TopologyClass=<class 'fipy.meshes.topologies.meshTopology._Mesh2DTopology'>)
```

Creates a 2D triangular mesh with horizontal faces numbered first then vertical faces, then diagonal faces. Vertices are numbered starting with the vertices at the corners of boxes and then the vertices at the centers of boxes. Cells on the right of boxes are numbered first, then cells on the top of boxes, then cells on the left of boxes, then cells on the bottom of boxes. Within each of the “sub-categories” in the above, the vertices, cells and faces are numbered in the usual way.

Parameters

- **dx, dy** (*float*) – The X and Y dimensions of each “box”. If $dx \neq dy$, the line segments connecting the cell centers will not be orthogonal to the faces.
- **nx, ny** (*int*) – The number of boxes in the X direction and the Y direction. The total number of boxes will be equal to $nx * ny$, and the total number of cells will be equal to $4 * nx * ny$.

`__module__ = 'fipy.meshes.tri2D'`

property physicalShape

Return physical dimensions of *Grid2D*.

property shape

`fipy.meshes.openMSHFile` (*name*, *dimensions=None*, *coordDimensions=None*, *communicator=SerialPETScCommWrapper()*, *overlap=1*, *mode='r'*, *background=None*)

Open a Gmsh *MSH* file

Parameters

- **filename** (*str*) – Gmsh output file
- **dimensions** (*int*) – Dimension of mesh
- **coordDimensions** (*int*) – Dimension of shapes
- **overlap** (*int*) – The number of overlapping cells for parallel simulations. Generally 1 is adequate. Higher order equations or discretizations require more. If *overlap* is greater than one, communication reverts to serial, as Gmsh only provides one layer of ghost cells.
- **mode** (*str*) – Beginning with *r* for reading and *w* for writing. The file will be created if it doesn't exist when opened for writing; it will be truncated when opened for writing. Add a *b* to the mode for binary files.
- **background** (*CellVariable*) – Specifies the desired characteristic lengths of the mesh cells

`fipy.meshes.openPOSFile` (*name*, *communicator=SerialPETScCommWrapper()*, *mode='w'*)

Open a Gmsh *POS* post-processing file

class `fipy.meshes.Gmsh2D` (*arg*, *coordDimensions=2*, *communicator=SerialPETScCommWrapper()*, *overlap=1*, *background=None*)

Bases: `fipy.meshes.mesh2D.Mesh2D`

Construct a 2D Mesh using Gmsh

```
>>> radius = 5.
>>> side = 4.
>>> squaredCircle = Gmsh2D('''
... // A mesh consisting of a square inside a circle inside a circle
...
... // define the basic dimensions of the mesh
...
... cellSize = 1;
... radius = %(radius)g;
... side = %(side)g;
...
... // define the compass points of the inner circle
...
... Point(1) = {0, 0, 0, cellSize};
```

(continues on next page)

(continued from previous page)

```

... Point(2) = {-radius, 0, 0, cellSize};
... Point(3) = {0, radius, 0, cellSize};
... Point(4) = {radius, 0, 0, cellSize};
... Point(5) = {0, -radius, 0, cellSize};
...
... // define the compass points of the outer circle
...
... Point(6) = {-2*radius, 0, 0, cellSize};
... Point(7) = {0, 2*radius, 0, cellSize};
... Point(8) = {2*radius, 0, 0, cellSize};
... Point(9) = {0, -2*radius, 0, cellSize};
...
... // define the corners of the square
...
... Point(10) = {side/2, side/2, 0, cellSize/2};
... Point(11) = {-side/2, side/2, 0, cellSize/2};
... Point(12) = {-side/2, -side/2, 0, cellSize/2};
... Point(13) = {side/2, -side/2, 0, cellSize/2};
...
... // define the inner circle
...
... Circle(1) = {2, 1, 3};
... Circle(2) = {3, 1, 4};
... Circle(3) = {4, 1, 5};
... Circle(4) = {5, 1, 2};
...
... // define the outer circle
...
... Circle(5) = {6, 1, 7};
... Circle(6) = {7, 1, 8};
... Circle(7) = {8, 1, 9};
... Circle(8) = {9, 1, 6};
...
... // define the square
...
... Line(9) = {10, 13};
... Line(10) = {13, 12};
... Line(11) = {12, 11};
... Line(12) = {11, 10};
...
... // define the three boundaries
...
... Line Loop(1) = {1, 2, 3, 4};
... Line Loop(2) = {5, 6, 7, 8};
... Line Loop(3) = {9, 10, 11, 12};
...
... // define the three domains
...
... Plane Surface(1) = {2, 1};
... Plane Surface(2) = {1, 3};
... Plane Surface(3) = {3};
...
... // label the three domains
...
... // attention: if you use any "Physical" labels, you *must* label
... // all elements that correspond to FiPy Cells (Physical Surface in 2D
... // and Physical Volume in 3D) or Gmsh will not include them and FiPy

```

(continues on next page)

(continued from previous page)

```

... // will not be able to include them in the Mesh.
...
... // note: if you do not use any labels, all Cells will be included.
...
... Physical Surface("Outer") = {1};
... Physical Surface("Middle") = {2};
... Physical Surface("Inner") = {3};
...
... // label the "north-west" part of the exterior boundary
...
... // note: you only need to label the Face elements
... // (Physical Line in 2D and Physical Surface in 3D) that correspond
... // to boundaries you are interested in. FiPy does not need them to
... // construct the Mesh.
...
... Physical Line("NW") = {5};
... ''' % locals()

```

It can be easier to specify certain domains and boundaries within Gmsh than it is to define the same domains and boundaries with FiPy expressions.

Here we compare obtaining the same Cells and Faces using FiPy's parametric descriptions and Gmsh's labels.

```
>>> x, y = squaredCircle.cellCenters
```

```

>>> middle = ((x**2 + y**2 <= radius**2)
...           & ~(x > -side/2) & (x < side/2)
...           & (y > -side/2) & (y < side/2)))

```

```

>>> print((middle == squaredCircle.physicalCells["Middle"]).all())
True

```

```
>>> X, Y = squaredCircle.faceCenters
```

```

>>> NW = ((X**2 + Y**2 > (1.99*radius)**2)
...       & (X**2 + Y**2 < (2.01*radius)**2)
...       & (X <= 0) & (Y >= 0))

```

```

>>> print((NW == squaredCircle.physicalFaces["NW"]).all())
True

```

It is possible to direct Gmsh to give the mesh different densities in different locations

```

>>> geo = '''
... // A mesh consisting of a square
...
... // define the corners of the square
...
... Point(1) = {1, 1, 0, 1};
... Point(2) = {0, 1, 0, 1};
... Point(3) = {0, 0, 0, 1};
... Point(4) = {1, 0, 0, 1};
...
... // define the square
...
...

```

(continues on next page)

(continued from previous page)

```

... Line(1) = {1, 2};
... Line(2) = {2, 3};
... Line(3) = {3, 4};
... Line(4) = {4, 1};
...
... // define the boundary
...
... Line Loop(1) = {1, 2, 3, 4};
...
... // define the domain
...
... Plane Surface(1) = {1};
... '''

```

```
>>> from fipy import CellVariable, numerix
```

```

>>> error = []
>>> bkg = None
>>> from builtins import range
>>> for refine in range(4):
...     square = Gmsh2D(geo, background=bkg)
...     x, y = square.cellCenters
...     bkg = CellVariable(mesh=square, value=abs(x / 4) + 0.01)
...     error.append(((2 * numerix.sqrt(square.cellVolumes) / bkg - 1)**2) *
    ↪ cellVolumeAverage)

```

Check that the mesh is (semi)monotonically approaching the desired density (the first step may increase, depending on the number of partitions)

```

>>> print(numerix.greater(error[:-1], error[1:]).all())
True

```

and that the final density is close enough to the desired density

```

>>> print(error[-1] < 0.02)
True

```

The initial mesh doesn't have to be from Gmsh

```
>>> from fipy import Tri2D
```

```

>>> trisquare = Tri2D(nx=1, ny=1)
>>> x, y = trisquare.cellCenters
>>> bkg = CellVariable(mesh=trisquare, value=abs(x / 4) + 0.01)
>>> std1 = (numerix.sqrt(2 * trisquare.cellVolumes) / bkg).std()

```

```

>>> square = Gmsh2D(geo, background=bkg)
>>> x, y = square.cellCenters
>>> bkg = CellVariable(mesh=square, value=abs(x / 4) + 0.01)
>>> std2 = (numerix.sqrt(2 * square.cellVolumes) / bkg).std()

```

```

>>> print(std1 > std2)
True

```

Parameters

- **arg** (*str*) – (i) the path to an *MSH* file, (ii) a path to a Gmsh geometry (*.geo*) file, or (iii) a Gmsh geometry script
- **coordDimensions** (*int*) – Dimension of shapes
- **overlap** (*int*) – The number of overlapping cells for parallel simulations. Generally 1 is adequate. Higher order equations or discretizations require more. If *overlap* is greater than one, communication reverts to serial, as Gmsh only provides one layer of ghost cells.
- **background** (*CellVariable*) – Specifies the desired characteristic lengths of the mesh cells

```
__init__(arg, coordDimensions=2, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.gmshMesh'
```

```
__setstate__(state)
```

```
class fipy.meshes.Gmsh2DIn3DSpace(arg, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Bases: *fipy.meshes.gmshMesh.Gmsh2D*

Create a topologically 2D Mesh in 3D coordinates using Gmsh

Parameters

- **arg** (*str*) – (i) the path to an *MSH* file, (ii) a path to a Gmsh geometry (*.geo*) file, or (iii) a Gmsh geometry script
- **coordDimensions** (*int*) – Dimension of shapes
- **overlap** (*int*) – The number of overlapping cells for parallel simulations. Generally 1 is adequate. Higher order equations or discretizations require more. If *overlap* is greater than one, communication reverts to serial, as Gmsh only provides one layer of ghost cells.
- **background** (*CellVariable*) – Specifies the desired characteristic lengths of the mesh cells

```
__init__(arg, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.gmshMesh'
```

```
class fipy.meshes.Gmsh3D(arg, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Bases: *fipy.meshes.mesh.Mesh*

Create a 3D Mesh using Gmsh

Parameters

- **arg** (*str*) – (i) the path to an *MSH* file, (ii) a path to a Gmsh geometry (*.geo*) file, or (iii) a Gmsh geometry script
- **overlap** (*int*) – The number of overlapping cells for parallel simulations. Generally 1 is adequate. Higher order equations or discretizations require more. If *overlap* is greater than one, communication reverts to serial, as Gmsh only provides one layer of ghost cells.
- **background** (*CellVariable*) – Specifies the desired characteristic lengths of the mesh cells

```
__init__(arg, communicator=SerialPETScCommWrapper(), overlap=1, background=None)
```

Initialize self. See help(type(self)) for accurate signature.

```
__module__ = 'fipy.meshes.gmshMesh'
__setstate__(state)
class fipy.meshes.GmshGrid2D(dx=1.0, dy=1.0, nx=1, ny=None, coordDimensions=2, communica-
                             tor=SerialPETScCommWrapper(), overlap=1)
Bases: fipy.meshes.gmshMesh.Gmsh2D
Should serve as a drop-in replacement for Grid2D
__init__(dx=1.0, dy=1.0, nx=1, ny=None, coordDimensions=2, communica-
         tor=SerialPETScCommWrapper(), overlap=1)
    Initialize self. See help(type(self)) for accurate signature.
__module__ = 'fipy.meshes.gmshMesh'
class fipy.meshes.GmshGrid3D(dx=1.0, dy=1.0, dz=1.0, nx=1, ny=None, nz=None, communica-
                             tor=SerialPETScCommWrapper(), overlap=1)
Bases: fipy.meshes.gmshMesh.Gmsh3D
Should serve as a drop-in replacement for Grid3D
__init__(dx=1.0, dy=1.0, dz=1.0, nx=1, ny=None, nz=None, communica-
         tor=SerialPETScCommWrapper(), overlap=1)
    Initialize self. See help(type(self)) for accurate signature.
__module__ = 'fipy.meshes.gmshMesh'
```


Chapter 31

fipy.solvers package

31.1 Subpackages

31.1.1 fipy.solvers.petsc package

Subpackages

fipy.solvers.petsc.comms package

Submodules

fipy.solvers.petsc.comms.parallelPETScCommWrapper module

```
class fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper
    Bases: fipy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper
    MPI Communicator wrapper
    Encapsulates capabilities needed for PETSc.
    Barrier ()
    MaxAll (vec)
    MinAll (vec)
    property Nproc
    __init__ ()
        Initialize self. See help(type(self)) for accurate signature.
    __module__ = 'fipy.solvers.petsc.comms.parallelPETScCommWrapper'
    all (a, axis=None)
    allclose (a, b, rtol=1e-05, atol=1e-08)
    allequal (a, b)
    allgather (sendobj=None)
```

any (*a*, *axis=None*)
bcast (*obj=None*, *root=0*)
property **procID**
sum (*a*, *axis=None*)

fipy.solvers.petsc.comms.petscCommWrapper module

class `fipy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper` (*petsc4py_comm=<petsc4py.PETSc.Comm object>*)
Bases: `fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper`
MPI Communicator wrapper
Encapsulates capabilities needed for PETSc. Some capabilities are not parallel.
Norm2 (*vec*)
__init__ (*petsc4py_comm=<petsc4py.PETSc.Comm object>*)
Initialize self. See help(type(self)) for accurate signature.
__module__ = `'fipy.solvers.petsc.comms.petscCommWrapper'`
property **mpi4py_comm**

fipy.solvers.petsc.comms.serialPETScCommWrapper module

class `fipy.solvers.petsc.comms.serialPETScCommWrapper.SerialPETScCommWrapper`
Bases: `fipy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper`
property **Nproc**
__init__ ()
Initialize self. See help(type(self)) for accurate signature.
__module__ = `'fipy.solvers.petsc.comms.serialPETScCommWrapper'`
property **procID**

Module contents

Submodules

fipy.solvers.petsc.dummySolver module

class `fipy.solvers.petsc.dummySolver.DummySolver` (**args, **kwargs*)
Bases: `fipy.solvers.petsc.petscSolver.PETScSolver`
Solver that doesn't do anything.
PETSc is intolerant of having zeros on the diagonal
Create a *Solver* object.
Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.

- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.petsc.dummySolver'
```

fipy.solvers.petsc.linearBicgSolver module

```
class fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver (tolerance=1e-10,
                                                            iterations=1000,
                                                            precon=None)
```

Bases: *fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver*

The *LinearBicgSolver* is an interface to the biconjugate gradient solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearBicgSolver'
```

```
solver = 'bicg'
```

fipy.solvers.petsc.linearCGSSolver module

```
class fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver (tolerance=1e-10,      it-
                                                            erations=1000,      pre-
                                                            con=None)
```

Bases: *fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver*

The *LinearCGSSolver* is an interface to the conjugate gradient squared solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearCGSSolver'
```

```
solver = 'cgs'
```

fipy.solvers.petsc.linearGMRESSolver module

```
class fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver (tolerance=1e-10,
                                                                iterations=1000,
                                                                precon=None)
```

Bases: *fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver*

The *LinearGMRESSolver* is an interface to the GMRES solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.

- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearGMRESSolver'  
solver = 'gmres'
```

fipy.solvers.petsc.linearLUSolver module

```
class fipy.solvers.petsc.linearLUSolver.LinearLUSolver (tolerance=1e-10,      itera-  
                                                         tions=10, precon='lu')  
Bases: fipy.solvers.petsc.petscSolver.PETScSolver
```

The *LinearLUSolver* is an interface to the LU preconditioner in PETSc. A direct solve is performed.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Ignored.

```
__init__ (tolerance=1e-10, iterations=10, precon='lu')
```

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Ignored.

```
__module__ = 'fipy.solvers.petsc.linearLUSolver'
```

fipy.solvers.petsc.linearPCGSolver module

```
class fipy.solvers.petsc.linearPCGSolver.LinearPCGSolver (tolerance=1e-10,      it-  
                                                         erations=1000,      pre-  
                                                         con=None)  
Bases: fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver
```

The *LinearPCGSolver* is an interface to the cg solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearPCGSolver'  
solver = 'cg'
```

fipy.solvers.petsc.petscKrylovSolver module

```
class fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver (tolerance=1e-10,
                                                             iterations=1000,
                                                             precon=None)
```

Bases: *fipy.solvers.petsc.petscSolver.PETScSolver*

Attention: This class is abstract, always create one of its subclasses. It provides the code to call all Krylov solvers from the PETSc package.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__init__ (tolerance=1e-10, iterations=1000, precon=None)
```

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.petscKrylovSolver'
```

fipy.solvers.petsc.petscSolver module

```
class fipy.solvers.petsc.petscSolver.PETScSolver (*args, **kwargs)
```

Bases: *fipy.solvers.solver.Solver*

Attention: This class is abstract. Always create one of its subclasses.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__init__ (*args, **kwargs)
```

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.petsc.petscSolver'
```

Module contents

`fiPy.solvers.petsc.DefaultSolver`
alias of `fiPy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver`

class `fiPy.solvers.petsc.DummySolver` (*args, **kwargs)
Bases: `fiPy.solvers.petsc.petscSolver.PETScSolver`

Solver that doesn't do anything.

PETSc is intolerant of having zeros on the diagonal

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

`__module__` = `'fiPy.solvers.petsc.dummySolver'`

`fiPy.solvers.petsc.DefaultAsymmetricSolver`
alias of `fiPy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver`

`fiPy.solvers.petsc.GeneralSolver`
alias of `fiPy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver`

class `fiPy.solvers.petsc.LinearLUSolver` (*tolerance=1e-10, iterations=10, precon='lu'*)
Bases: `fiPy.solvers.petsc.petscSolver.PETScSolver`

The *LinearLUSolver* is an interface to the LU preconditioner in PETSc. A direct solve is performed.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Ignored.

`__init__` (*tolerance=1e-10, iterations=10, precon='lu'*)

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Ignored.

`__module__` = `'fiPy.solvers.petsc.linearLUSolver'`

class `fiPy.solvers.petsc.LinearPCGSolver` (*tolerance=1e-10, iterations=1000, precon=None*)
Bases: `fiPy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver`

The *LinearPCGSolver* is an interface to the cg solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearPCGSolver'
```

```
solver = 'cg'
```

```
class fipy.solvers.petsc.LinearGMRESSolver (tolerance=1e-10, iterations=1000, pre-
                                         con=None)
```

Bases: *fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver*

The *LinearGMRESSolver* is an interface to the GMRES solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearGMRESSolver'
```

```
solver = 'gmres'
```

```
class fipy.solvers.petsc.LinearBicgSolver (tolerance=1e-10, iterations=1000, pre-
                                         con=None)
```

Bases: *fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver*

The *LinearBicgSolver* is an interface to the biconjugate gradient solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearBicgSolver'
```

```
solver = 'bicg'
```

```
class fipy.solvers.petsc.LinearCGSSolver (tolerance=1e-10, iterations=1000, pre-
                                         con=None)
```

Bases: *fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver*

The *LinearCGSSolver* is an interface to the conjugate gradient squared solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearCGSSolver'
```

```
solver = 'cgs'
```

31.1.2 fipy.solvers.pyAMG package

Subpackages

fipy.solvers.pyAMG.preconditioners package

Submodules

fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner module

```
class fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner.SmoothedAggregationPreconditioner
    Bases: object
    __dict__ = mappingproxy({'__module__': 'fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner'})
    __init__(self)
        Initialize self. See help(type(self)) for accurate signature.
    __module__ = 'fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner'
    __weakref__ = []
        list of weak references to the object (if defined)
```

Module contents

Submodules

fipy.solvers.pyAMG.linearCGSSolver module

```
class fipy.solvers.pyAMG.linearCGSSolver.LinearCGSSolver(tolerance=1e-15, iterations=2000, preconditioner=
    <fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner.SmoothedAggregationPreconditioner object>)
```

Bases: *fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver*

The *LinearCGSSolver* is an interface to the CGS solver in Scipy, using the PyAMG *SmoothedAggregationPreconditioner* by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, optional) –

```
__init__(self, tolerance=1e-15, iterations=2000, preconditioner=
    <fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner.SmoothedAggregationPreconditioner object>)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, optional) –

```
__module__ = 'fipy.solvers.pyAMG.linearCGSSolver'
```


fipy.solvers.pyAMG.linearGMRESSolver module

```
class fipy.solvers.pyAMG.linearGMRESSolver.LinearGMRESSolver (tolerance=1e-15, iterations=2000, preconditioner=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner object>)
```

Bases: `fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver`

The *LinearGMRESSolver* is an interface to the GMRES solver in Scipy, using the PyAMG *SmoothedAggregationPreconditioner* by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, *optional*) –

```
__init__ (tolerance=1e-15, iterations=2000, precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner object>)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, *optional*) –

```
__module__ = 'fipy.solvers.pyAMG.linearGMRESSolver'
```

fipy.solvers.pyAMG.linearGeneralSolver module

```
class fipy.solvers.pyAMG.linearGeneralSolver.LinearGeneralSolver (tolerance=1e-10, iterations=1000, precondition=None)
```

Bases: `fipy.solvers.scipy.scipySolver._ScipySolver`

The *LinearGeneralSolver* is an interface to the generic PyAMG, which solves the arbitrary system $Ax=b$ with the best out-of-the box choice for a solver. See *pyAMG.solve* for details.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.pyAMG.linearGeneralSolver'
```

fipy.solvers.pyAMG.linearLUSolver module

```
class fipy.solvers.pyAMG.linearLUSolver.LinearLUSolver (tolerance=1e-10, iterations=1000, precon=None)
```

Bases: `fipy.solvers.scipy.scipySolver._ScipySolver`

The *LinearLUSolver* solves a linear system of equations using LU-factorization. The *LinearLUSolver* is a wrapper class for the the Scipy *scipy.sparse.linalg.splu* module.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.scipy.linearLUSolver'
```

fipy.solvers.pyAMG.linearPCGSolver module

```
class fipy.solvers.pyAMG.linearPCGSolver.LinearPCGSolver (tolerance=1e-15, iterations=2000, precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner object>)
```

Bases: `fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver`

The *LinearPCGSolver* is an interface to the PCG solver in Scipy, using the PyAMG *SmoothedAggregationPreconditioner* by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, optional) –

```
__init__ (tolerance=1e-15, iterations=2000, precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner object>)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, optional) –

```
__module__ = 'fipy.solvers.pyAMG.linearPCGSolver'
```

Module contents

`fipy.solvers.pyAMG.DefaultSolver`
alias of `fipy.solvers.pyAMG.linearGMRESSolver.LinearGMRESSolver`

`fipy.solvers.pyAMG.DummySolver`
alias of `fipy.solvers.pyAMG.linearGMRESSolver.LinearGMRESSolver`

`fipy.solvers.pyAMG.DefaultAsymmetricSolver`
alias of `fipy.solvers.scipy.linearLUSolver.LinearLUSolver`

`fipy.solvers.pyAMG.GeneralSolver`
alias of `fipy.solvers.pyAMG.linearGeneralSolver.LinearGeneralSolver`

class `fipy.solvers.pyAMG.LinearGMRESSolver` (*tolerance=1e-15, iterations=2000, precondition=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner object>*)

Bases: `fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver`

The *LinearGMRESSolver* is an interface to the GMRES solver in Scipy, using the PyAMG *SmoothedAggregationPreconditioner* by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner, optional*) –

__init__ (*tolerance=1e-15, iterations=2000, precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner object>*)

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner, optional*) –

__module__ = `'fipy.solvers.pyAMG.linearGMRESSolver'`

class `fipy.solvers.pyAMG.LinearCGSSolver` (*tolerance=1e-15, iterations=2000, precondition=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner object>*)

Bases: `fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver`

The *LinearCGSSolver* is an interface to the CGS solver in Scipy, using the PyAMG *SmoothedAggregationPreconditioner* by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner, optional*) –

__init__ (*tolerance=1e-15, iterations=2000, precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner object>*)

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.

```
    • precon (SmoothedAggregationPreconditioner, optional) –  
    __module__ = 'fipy.solvers.pyAMG.linearCGSSolver'  
class fipy.solvers.pyAMG.LinearPCGSolver (tolerance=1e-15, iterations=2000, pre-  
                                         con=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPrecond  
                                         object>)  
Bases: fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver
```

The *LinearPCGSolver* is an interface to the PCG solver in Scipy, using the PyAMG *SmoothedAggregationPreconditioner* by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, *optional*) –

```
__init__ (tolerance=1e-15, iterations=2000, precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner  
         object>)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, *optional*) –

```
__module__ = 'fipy.solvers.pyAMG.linearPCGSolver'
```

```
class fipy.solvers.pyAMG.LinearLUSolver (tolerance=1e-10, iterations=1000, precon=None)  
Bases: fipy.solvers.scipy.scipySolver._ScipySolver
```

The *LinearLUSolver* solves a linear system of equations using LU-factorization. The *LinearLUSolver* is a wrapper class for the the Scipy *scipy.sparse.linalg.splu* module.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.scipy.linearLUSolver'
```

```
class fipy.solvers.pyAMG.LinearGeneralSolver (tolerance=1e-10, iterations=1000, pre-  
                                              con=None)  
Bases: fipy.solvers.scipy.scipySolver._ScipySolver
```

The *LinearGeneralSolver* is an interface to the generic PyAMG, which solves the arbitrary system $Ax=b$ with the best out-of-the box choice for a solver. See *pyAMG.solve* for details.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.pyAMG.linearGeneralSolver'
```

31.1.3 fipy.solvers.pyamgx package

Subpackages

fipy.solvers.pyamgx.preconditioners package

Submodules

fipy.solvers.pyamgx.preconditioners.preconditioners module

Module contents

fipy.solvers.pyamgx.smoothers package

Submodules

fipy.solvers.pyamgx.smoothers.smoothers module

Module contents

Submodules

fipy.solvers.pyamgx.aggregationAMGSolver module

```
class fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver (tolerance=1e-10, iterations=2000, precon=None, smoother={'max_iters': 1, 'solver': 'BLOCK_JACOBI'}, **kwargs)
```

Bases: *fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver*

The *AggregationAMGSolver* is an interface to the aggregation AMG solver in AMGX, with a Jacobi smoother by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner, optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__ (tolerance=1e-10, iterations=2000, precon=None, smoother={'max_iters': 1, 'solver': 'BLOCK_JACOBI'}, **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.

- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner, optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.aggregationAMGSolver'
```

fipy.solvers.pyamgx.classicalAMGSolver module

```
class fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver (tolerance=1e-10, iterations=2000, precon=None, smoother={'max_iters': 1, 'solver': 'BLOCK_JACOBI'}, **kwargs)
```

Bases: *fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver*

The *ClassicalAMGSolver* is an interface to the classical AMG solver in AMGX, with a Jacobi smoother by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner, optional*) –
- **smoother** (*Smoother, optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__ (tolerance=1e-10, iterations=2000, precon=None, smoother={'max_iters': 1, 'solver': 'BLOCK_JACOBI'}, **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner, optional*) –
- **smoother** (*Smoother, optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.classicalAMGSolver'
```

fipy.solvers.pyamgx.linearBiCGStabSolver module

```
class fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver (tolerance=1e-10, iterations=2000, precondition={'max_iters': 1, 'solver': 'BLOCK_JACOBI'}, **kwargs)
```

Bases: `fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver`

The *LinearBiCGStabSolver* is an interface to the PBICGSTAB solver in AMGX, with a Jacobi preconditioner by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner, optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__ (tolerance=1e-10, iterations=2000, precondition={'max_iters': 1, 'solver': 'BLOCK_JACOBI'}, **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner, optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.linearBiCGStabSolver'
```

fipy.solvers.pyamgx.linearCGSolver module

```
class fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver (tolerance=1e-10, iterations=2000, precondition={'max_iters': 1, 'solver': 'BLOCK_JACOBI'}, **kwargs)
```

Bases: `fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver`

The *LinearCGSolver* is an interface to the PCG solver in AMGX, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner, optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__(tolerance=1e-10, iterations=2000, precon={'max_iters': 1, 'solver': 'BLOCK_JACOBI'},
        **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.linearCGSolver'
```

```
fipy.solvers.pyamgx.linearCGSolver.LinearPCGSolver  
alias of fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver
```

fipy.solvers.pyamgx.linearFGMRESSolver module

```
class fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver (tolerance=1e-10, iterations=2000, precon={'max_iters': 1, 'solver': 'BLOCK_JACOBI'}, **kwargs)
```

Bases: *fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver*

The *LinearFGMRESSolver* is an interface to the FGMRES solver in AMGX, with a Jacobi preconditioner by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__(tolerance=1e-10, iterations=2000, precon={'max_iters': 1, 'solver': 'BLOCK_JACOBI'},
        **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.linearFGMRESSolver'
```


fipy.solvers.pyamgx.linearGMRESSolver module

```
class fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver (tolerance=1e-10,
                                                                iterations=2000,
                                                                precon={'max_iters': 1,
                                                                'solver': 'BLOCK_JACOBI'},
                                                                **kwargs)
```

Bases: `fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver`

The *LinearGMRESSolver* is an interface to the GMRES solver in AMGX, with a Jacobi preconditioner by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner, optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__ (tolerance=1e-10, iterations=2000, precon={'max_iters': 1, 'solver': 'BLOCK_JACOBI'},
          **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner, optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.linearGMRESSolver'
```

fipy.solvers.pyamgx.linearLUSolver module

```
class fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver (tolerance=1e-10,
                                                         iterations=1000,
                                                         precon=None)
```

Bases: `fipy.solvers.scipy.scipySolver._ScipySolver`

The *LinearLUSolver* solves a linear system of equations using LU-factorization. The *LinearLUSolver* is a wrapper class for the the Scipy `scipy.sparse.linalg.splu` module.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.scipy.linearLUSolver'
```

fipy.solvers.pyamgx.pyAMGXSolver module

```
class fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver(config_dict, tolerance=1e-10,  
                                                    iterations=2000, precon=None,  
                                                    smoother=None, **kwargs)
```

Bases: *fipy.solvers.solver.Solver*

Parameters

- **config_dict** (*dict*) – AMGX configuration options
- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- **smoother** (*Smoother*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__exit__ (*args)
```

```
__init__ (config_dict, tolerance=1e-10, iterations=2000, precon=None, smoother=None, **kwargs)
```

Parameters

- **config_dict** (*dict*) – AMGX configuration options
- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- **smoother** (*Smoother*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.pyAMGXSolver'
```

Module contents

fipy.solvers.pyamgx.DefaultSolver

alias of *fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver*

fipy.solvers.pyamgx.DummySolver

alias of *fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver*

fipy.solvers.pyamgx.DefaultAsymmetricSolver

alias of *fipy.solvers.scipy.linearLUSolver.LinearLUSolver*

fipy.solvers.pyamgx.GeneralSolver

alias of *fipy.solvers.scipy.linearLUSolver.LinearLUSolver*

```
class fipy.solvers.pyamgx.LinearCGSolver (tolerance=1e-10, iterations=2000,  
                                           precon={'max_iters': 1, 'solver':  
                                           'BLOCK_JACOBI'}, **kwargs)
```

Bases: *fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver*

The *LinearCGSolver* is an interface to the PCG solver in AMGX, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.

- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__ (tolerance=1e-10, iterations=2000, precon={'max_iters': 1, 'solver': 'BLOCK_JACOBI'},  
          **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.linearCGSolver'
```

```
fipy.solvers.pyamgx.LinearPCGSolver
```

```
alias of fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver
```

```
class fipy.solvers.pyamgx.LinearFGMRESSolver (tolerance=1e-10, iterations=2000,  
                                              precon={'max_iters': 1, 'solver':  
                                              'BLOCK_JACOBI'}, **kwargs)
```

```
Bases: fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver
```

The *LinearFGMRESSolver* is an interface to the FGMRES solver in AMGX, with a Jacobi preconditioner by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__ (tolerance=1e-10, iterations=2000, precon={'max_iters': 1, 'solver': 'BLOCK_JACOBI'},  
          **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.linearFGMRESSolver'
```

```
class fipy.solvers.pyamgx.LinearBiCGStabSolver (tolerance=1e-10, iterations=2000,  
                                              precon={'max_iters': 1, 'solver':  
                                              'BLOCK_JACOBI'}, **kwargs)
```

```
Bases: fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver
```

The *LinearBiCGStabSolver* is an interface to the PBICGSTAB solver in AMGX, with a Jacobi preconditioner by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.

- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__(tolerance=1e-10, iterations=2000, precon={'max_iters': 1, 'solver': 'BLOCK_JACOBI'},  
        **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.linearBiCGStabSolver'
```

```
class fipy.solvers.pyamgx.LinearLUSolver (tolerance=1e-10, iterations=1000, pre-  
                                         con=None)  
Bases: fipy.solvers.scipy.scipySolver._ScipySolver
```

The *LinearLUSolver* solves a linear system of equations using LU-factorization. The *LinearLUSolver* is a wrapper class for the the Scipy *scipy.sparse.linalg.splu* module.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.scipy.linearLUSolver'
```

```
class fipy.solvers.pyamgx.AggregationAMGSolver (tolerance=1e-10, iterations=2000, pre-  
                                                con=None, smoother={'max_iters': 1,  
                                                                    'solver': 'BLOCK_JACOBI'}, **kwargs)  
Bases: fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver
```

The *AggregationAMGSolver* is an interface to the aggregation AMG solver in AMGX, with a Jacobi smoother by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__init__(tolerance=1e-10, iterations=2000, precon=None, smoother={'max_iters': 1, 'solver':  
                        'BLOCK_JACOBI'}, **kwargs)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- ****kwargs** – Other AMGX solver options

```
__module__ = 'fipy.solvers.pyamgx.aggregationAMGSolver'
```

31.1.4 fipy.solvers.pysparse package

Subpackages

fipy.solvers.pysparse.preconditioners package

Submodules

fipy.solvers.pysparse.preconditioners.jacobiPreconditioner module

fipy.solvers.pysparse.preconditioners.preconditioner module

fipy.solvers.pysparse.preconditioners.ssorPreconditioner module

Module contents

Submodules

fipy.solvers.pysparse.linearCGSSolver module

fipy.solvers.pysparse.linearGMRESSolver module

fipy.solvers.pysparse.linearJORSolver module

fipy.solvers.pysparse.linearLUSolver module

fipy.solvers.pysparse.linearPCGSolver module

fipy.solvers.pysparse.pysparseSolver module

Module contents

31.1.5 fipy.solvers.scipy package

Submodules

fipy.solvers.scipy.linearBicgstabSolver module

```
class fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver (tolerance=1e-15, iterations=2000, pre-con=None)
```

Bases: fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver

The *LinearBicgstabSolver* is an interface to the Bicgstab solver in Scipy, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.

- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__init__` (*tolerance=1e-15, iterations=2000, precon=None*)

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__module__` = `'fipy.solvers.scipy.linearBicgstabSolver'`

fipy.solvers.scipy.linearCGSSolver module

```
class fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver (tolerance=1e-15,      it-
                                                         erations=2000,      pre-
                                                         con=None)
```

Bases: `fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver`

The *LinearCGSSolver* is an interface to the CGS solver in Scipy, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__init__` (*tolerance=1e-15, iterations=2000, precon=None*)

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__module__` = `'fipy.solvers.scipy.linearCGSSolver'`

fipy.solvers.scipy.linearGMRESSolver module

```
class fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver (tolerance=1e-15,
                                                                iterations=2000,
                                                                precon=None)
```

Bases: `fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver`

The *LinearGMRESSolver* is an interface to the GMRES solver in Scipy, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__init__` (*tolerance=1e-15, iterations=2000, precon=None*)

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

```
__module__ = 'fipy.solvers.scipy.linearGMRESSolver'
```

fipy.solvers.scipy.linearLUSolver module

```
class fipy.solvers.scipy.linearLUSolver.LinearLUSolver (tolerance=1e-10, iterations=1000, precon=None)
```

Bases: fipy.solvers.scipy.scipySolver._ScipySolver

The *LinearLUSolver* solves a linear system of equations using LU-factorization. The *LinearLUSolver* is a wrapper class for the the Scipy *scipy.sparse.linalg.splu* module.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.scipy.linearLUSolver'
```

fipy.solvers.scipy.linearPCGSolver module

```
class fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver (tolerance=1e-15, iterations=2000, precon=None)
```

Bases: fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver

The *LinearPCGSolver* is an interface to the CG solver in Scipy, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

```
__init__ (tolerance=1e-15, iterations=2000, precon=None)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

```
__module__ = 'fipy.solvers.scipy.linearPCGSolver'
```

fipy.solvers.scipy.scipyKrylovSolver module**fipy.solvers.scipy.scipySolver module****Module contents**

`fipy.solvers.scipy.DefaultSolver`

alias of `fipy.solvers.scipy.linearLUSolver.LinearLUSolver`

`fipy.solvers.scipy.DummySolver`

alias of `fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver`

`fipy.solvers.scipy.DefaultAsymmetricSolver`

alias of `fipy.solvers.scipy.linearLUSolver.LinearLUSolver`

`fipy.solvers.scipy.GeneralSolver`

alias of `fipy.solvers.scipy.linearLUSolver.LinearLUSolver`

class `fipy.solvers.scipy.LinearCGSSolver` (*tolerance=1e-15, iterations=2000, precon=None*)

Bases: `fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver`

The *LinearCGSSolver* is an interface to the CGS solver in Scipy, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__init__` (*tolerance=1e-15, iterations=2000, precon=None*)

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__module__` = `'fipy.solvers.scipy.linearCGSSolver'`

class `fipy.solvers.scipy.LinearGMRESSolver` (*tolerance=1e-15, iterations=2000, precon=None*)

Bases: `fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver`

The *LinearGMRESSolver* is an interface to the GMRES solver in Scipy, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__init__` (*tolerance=1e-15, iterations=2000, precon=None*)

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.


```
__module__ = 'fipy.solvers.scipy.linearGMRESSolver'
```

```
class fipy.solvers.scipy.LinearBicgstabSolver (tolerance=1e-15, iterations=2000, pre-
                                             con=None)
```

Bases: `fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver`

The *LinearBicgstabSolver* is an interface to the Bicgstab solver in Scipy, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

```
__init__ (tolerance=1e-15, iterations=2000, precon=None)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

```
__module__ = 'fipy.solvers.scipy.linearBicgstabSolver'
```

```
class fipy.solvers.scipy.LinearLUSolver (tolerance=1e-10, iterations=1000, precon=None)
```

Bases: `fipy.solvers.scipy.scipySolver._ScipySolver`

The *LinearLUSolver* solves a linear system of equations using LU-factorization. The *LinearLUSolver* is a wrapper class for the the Scipy *scipy.sparse.linalg.splu* module.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.scipy.linearLUSolver'
```

```
class fipy.solvers.scipy.LinearPCGSolver (tolerance=1e-15, iterations=2000, pre-
                                           con=None)
```

Bases: `fipy.solvers.scipy.scipyKrylovSolver._ScipyKrylovSolver`

The *LinearPCGSolver* is an interface to the CG solver in Scipy, with no preconditioning by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

```
__init__ (tolerance=1e-15, iterations=2000, precon=None)
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

```
__module__ = 'fipy.solvers.scipy.linearPCGSolver'
```

31.1.6 fipy.solvers.trilinos package

Subpackages

`fipy.solvers.trilinos.comms` package

Submodules

`fipy.solvers.trilinos.comms.epetraCommWrapper` module

`fipy.solvers.trilinos.comms.parallelEpetraCommWrapper` module

`fipy.solvers.trilinos.comms.serialEpetraCommWrapper` module

Module contents

`fipy.solvers.trilinos.preconditioners` package

Submodules

`fipy.solvers.trilinos.preconditioners.domDecompPreconditioner` module

`fipy.solvers.trilinos.preconditioners.icPreconditioner` module

`fipy.solvers.trilinos.preconditioners.jacobiPreconditioner` module

`fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner` module

`fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner` module

`fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner` module

`fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner` module

`fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner` module

`fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner` module

`fipy.solvers.trilinos.preconditioners.preconditioner` module

Module contents

Submodules

`fipy.solvers.trilinos.linearBicgstabSolver` module

`fipy.solvers.trilinos.linearCGSSolver` module

`fipy.solvers.trilinos.linearGMRESSolver` module

`fipy.solvers.trilinos.linearLUSolver` module

`fipy.solvers.trilinos.linearPCGSolver` module

`fipy.solvers.trilinos.trilinosAztecOOSolver` module

`fipy.solvers.trilinos.trilinosMLTest` module

`fipy.solvers.trilinos.trilinosNonlinearSolver` module

`fipy.solvers.trilinos.trilinosSolver` module

Module contents

31.2 Submodules

31.3 `fipy.solvers.pysparseMatrixSolver` module

31.4 `fipy.solvers.solver` module

The iterative solvers may output warnings if the solution is considered unsatisfactory. If you are not interested in these warnings, you can invoke python with a warning filter such as:

```
$ python -Wignore::fipy.SolverConvergenceWarning myscript.py
```

If you are extremely concerned about your preconditioner for some reason, you can abort whenever it has problems with:

```
$ python -Werror::fipy.PreconditionerWarning myscript.py
```

exception `fipy.solvers.solver.SolverConvergenceWarning` (*solver, iter, relres*)

Bases: `Warning`

`__init__` (*solver, iter, relres*)

Initialize self. See `help(type(self))` for accurate signature.

`__module__` = `'fipy.solvers.solver'`

`__str__` ()

Return `str(self)`.

`__weakref__`

list of weak references to the object (if defined)

exception `fipy.solvers.solver.MaximumIterationWarning` (*solver, iter, relres*)

Bases: `fipy.solvers.solver.SolverConvergenceWarning`

`__module__` = `'fipy.solvers.solver'`

```
__str__()
    Return str(self).

exception fipy.solvers.solver.PreconditionerWarning(solver, iter, relres)
    Bases: fipy.solvers.solver.SolverConvergenceWarning

__module__ = 'fipy.solvers.solver'

exception fipy.solvers.solver.IllConditionedPreconditionerWarning(solver, iter,
                                                                    relres)
    Bases: fipy.solvers.solver.PreconditionerWarning

__module__ = 'fipy.solvers.solver'

__str__()
    Return str(self).

exception fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning(solver,
                                                                           iter,
                                                                           rel-
                                                                           res)
    Bases: fipy.solvers.solver.PreconditionerWarning

__module__ = 'fipy.solvers.solver'

__str__()
    Return str(self).

exception fipy.solvers.solver.MatrixIllConditionedWarning(solver, iter, relres)
    Bases: fipy.solvers.solver.SolverConvergenceWarning

__module__ = 'fipy.solvers.solver'

__str__()
    Return str(self).

exception fipy.solvers.solver.StagnatedSolverWarning(solver, iter, relres)
    Bases: fipy.solvers.solver.SolverConvergenceWarning

__module__ = 'fipy.solvers.solver'

__str__()
    Return str(self).

exception fipy.solvers.solver.ScalarQuantityOutOfRangeWarning(solver, iter, rel-
                                                                res)
    Bases: fipy.solvers.solver.SolverConvergenceWarning

__module__ = 'fipy.solvers.solver'

__str__()
    Return str(self).

class fipy.solvers.solver.Solver(tolerance=1e-10, iterations=1000, precon=None)
    Bases: object

    The base LinearXSolver class.
```

Attention: This class is abstract. Always create one of its subclasses.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__dict__ = mappingproxy({'__module__': 'fipy.solvers.solver', '__doc__': '\n The bas
__enter__ ()
__exit__ (exc_type, exc_value, traceback)
__init__ (tolerance=1e-10, iterations=1000, precon=None)
    Create a Solver object.
```

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__module__ = 'fipy.solvers.solver'
__repr__ ()
    Return repr(self).
__weakref__
    list of weak references to the object (if defined)
```

31.5 fipy.solvers.test module

31.6 Module contents

exception `fipy.solvers.SolverConvergenceWarning` (*solver, iter, relres*)

Bases: `Warning`

```
__init__ (solver, iter, relres)
    Initialize self. See help(type(self)) for accurate signature.
__module__ = 'fipy.solvers.solver'
__str__ ()
    Return str(self).
__weakref__
    list of weak references to the object (if defined)
```

exception `fipy.solvers.MaximumIterationWarning` (*solver, iter, relres*)

Bases: `fipy.solvers.solver.SolverConvergenceWarning`

```
__module__ = 'fipy.solvers.solver'
__str__ ()
    Return str(self).
```

exception `fipy.solvers.PreconditionerWarning` (*solver, iter, relres*)

Bases: `fipy.solvers.solver.SolverConvergenceWarning`

```
__module__ = 'fipy.solvers.solver'
```

```
exception fipy.solvers.IllConditionedPreconditionerWarning(solver, iter, relres)
    Bases: fipy.solvers.solver.PreconditionerWarning

    __module__ = 'fipy.solvers.solver'

    __str__()
        Return str(self).

exception fipy.solvers.PreconditionerNotPositiveDefiniteWarning(solver, iter,
                                                                relres)
    Bases: fipy.solvers.solver.PreconditionerWarning

    __module__ = 'fipy.solvers.solver'

    __str__()
        Return str(self).

exception fipy.solvers.MatrixIllConditionedWarning(solver, iter, relres)
    Bases: fipy.solvers.solver.SolverConvergenceWarning

    __module__ = 'fipy.solvers.solver'

    __str__()
        Return str(self).

exception fipy.solvers.StagnatedSolverWarning(solver, iter, relres)
    Bases: fipy.solvers.solver.SolverConvergenceWarning

    __module__ = 'fipy.solvers.solver'

    __str__()
        Return str(self).

exception fipy.solvers.ScalarQuantityOutOfRangeWarning(solver, iter, relres)
    Bases: fipy.solvers.solver.SolverConvergenceWarning

    __module__ = 'fipy.solvers.solver'

    __str__()
        Return str(self).

class fipy.solvers.Solver(tolerance=1e-10, iterations=1000, precon=None)
    Bases: object

    The base LinearXSolver class.
```

Attention: This class is abstract. Always create one of its subclasses.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__dict__ = mappingproxy({'__module__': 'fipy.solvers.solver', '__doc__': '\n The bas
__enter__()
__exit__(exc_type, exc_value, traceback)
```

`__init__` (*tolerance=1e-10, iterations=1000, precon=None*)
Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

`__module__` = `'fipy.solvers.solver'`

`__repr__` ()
Return `repr(self)`.

`__weakref__`
list of weak references to the object (if defined)

`fipy.solvers.DefaultSolver`

alias of `fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver`

class `fipy.solvers.DummySolver` (**args, **kwargs*)

Bases: `fipy.solvers.petsc.petscSolver.PETScSolver`

Solver that doesn't do anything.

PETSc is intolerant of having zeros on the diagonal

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

`__module__` = `'fipy.solvers.petsc.dummySolver'`

`fipy.solvers.DefaultAsymmetricSolver`

alias of `fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver`

`fipy.solvers.GeneralSolver`

alias of `fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver`

class `fipy.solvers.LinearLUSolver` (*tolerance=1e-10, iterations=10, precon='lu'*)

Bases: `fipy.solvers.petsc.petscSolver.PETScSolver`

The *LinearLUSolver* is an interface to the LU preconditioner in PETSc. A direct solve is performed.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Ignored.

`__init__` (*tolerance=1e-10, iterations=10, precon='lu'*)

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Ignored.

```
__module__ = 'fipy.solvers.petsc.linearLUSolver'
```

```
class fipy.solvers.LinearPCGSolver (tolerance=1e-10, iterations=1000, precon=None)
    Bases: fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver
```

The *LinearPCGSolver* is an interface to the cg solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearPCGSolver'
```

```
solver = 'cg'
```

```
class fipy.solvers.LinearGMRESSolver (tolerance=1e-10, iterations=1000, precon=None)
    Bases: fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver
```

The *LinearGMRESSolver* is an interface to the GMRES solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearGMRESSolver'
```

```
solver = 'gmres'
```

```
class fipy.solvers.LinearBicgSolver (tolerance=1e-10, iterations=1000, precon=None)
    Bases: fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver
```

The *LinearBicgSolver* is an interface to the biconjugate gradient solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearBicgSolver'
```

```
solver = 'bicg'
```

```
class fipy.solvers.LinearCGSSolver (tolerance=1e-10, iterations=1000, precon=None)
    Bases: fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver
```

The *LinearCGSSolver* is an interface to the conjugate gradient squared solver in PETSc, using no preconditioner by default.

Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

```
__module__ = 'fipy.solvers.petsc.linearCGSSolver'
```



```
solver = 'cgs'
```


fipy.steppers package

32.1 Submodules

32.2 fipy.steppers.pidStepper module

class fipy.steppers.pidStepper.**PIDStepper** (*vardata=()*, *proportional=0.075*, *integral=0.175*, *derivative=0.01*)

Bases: *fipy.steppers.stepper.Stepper*

Adaptive stepper using a PID controller, based on:

```
@article{PIDpaper,
  author = {A. M. P. Valli and G. F. Carey and A. L. G. A. Coutinho},
  title = {Control strategies for timestep selection in finite element
    simulation of incompressible flows and coupled
    reaction-convection-diffusion processes},
  journal = {Int. J. Numer. Meth. Fluids},
  volume = 47,
  year = 2005,
  pages = {201-231},
}
```

__init__ (*vardata=()*, *proportional=0.075*, *integral=0.175*, *derivative=0.01*)

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.steppers.pidStepper'

32.3 fipy.steppers.pseudoRKQSStepper module

class fipy.steppers.pseudoRKQSStepper.**PseudoRKQSStepper** (*vardata=()*, *safety=0.9*, *pgrow=-0.2*, *pshrink=-0.25*, *errcon=0.000189*)

Bases: *fipy.steppers.stepper.Stepper*

Adaptive stepper based on the rkqs (Runge-Kutta “quality-controlled” stepper) algorithm of Numerical Recipes in C: 2nd Edition, Section 16.2.

Not really appropriate, since we're not doing Runge-Kutta steps in the first place, but works OK.

```
__init__(vardata=(), safety=0.9, pgrow=-0.2, pshrink=-0.25, errcon=0.000189)
    Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.steps.pseudoRKQSStepper'
```

32.4 fipy.steps.stepper module

```
class fipy.steps.stepper.Stepper(vardata=())
    Bases: object

    __dict__ = mappingproxy({'__module__': 'fipy.steps.stepper', '__init__': <function
    __init__(vardata=())
        Initialize self. See help(type(self)) for accurate signature.

    __module__ = 'fipy.steps.stepper'

    __weakref__
        list of weak references to the object (if defined)

    static failFn(vardata, dt, *args, **kwargs)

    step(dt, dtTry=None, dtMin=None, dtPrev=None, sweepFn=None, successFn=None, failFn=None,
        *args, **kwargs)

    static successFn(vardata, dt, dtPrev, elapsed, *args, **kwargs)

    static sweepFn(vardata, dt, *args, **kwargs)
```

32.5 Module contents

`fipy.steps.L1error` (*var*, *matrix*, *RHSvector*)

$$\frac{\|\mathbf{var} - \mathbf{var}^{\text{old}}\|_1}{\|\mathbf{var}^{\text{old}}\|_1}$$

where $\|\vec{x}\|_1$ is the L^1 norm of \vec{x} .

Parameters

- **var** (`CellVariable`) – The *CellVariable* in question.
- **matrix** – (*ignored*)
- **RHSvector** – (*ignored*)

`fipy.steps.L2error` (*var*, *matrix*, *RHSvector*)

$$\frac{\|\mathbf{var} - \mathbf{var}^{\text{old}}\|_2}{\|\mathbf{var}^{\text{old}}\|_2}$$

where $\|\vec{x}\|_2$ is the L^2 norm of \vec{x} .

Parameters

- **var** (`CellVariable`) – The *CellVariable* in question.
- **matrix** – (*ignored*)

- **RHSvector** – (*ignored*)

`fipy.steppers.LINFerror` (*var*, *matrix*, *RHSvector*)

$$\frac{\|\text{var} - \text{var}^{\text{old}}\|_{\infty}}{\|\text{var}^{\text{old}}\|_{\infty}}$$

where $\|\vec{x}\|_{\infty}$ is the L^{∞} norm of \vec{x} .

Parameters

- **var** (*CellVariable*) – The *CellVariable* in question.
- **matrix** – (*ignored*)
- **RHSvector** – (*ignored*)

`fipy.steppers.sweepMonotonic` (*fn*, **args*, ***kwargs*)

Repeatedly calls `fn(*args, **kwargs)()` until the residual returned by `fn()` is no longer decreasing.

Parameters

- **fn** (*function*) – The function to call
- ***args** –
- ****kwargs** –

Returns

Return type `float`

fipy.terms package

33.1 Submodules

33.2 fipy.terms.abstractBinaryTerm module

33.3 fipy.terms.abstractConvectionTerm module

33.4 fipy.terms.abstractDiffusionTerm module

33.5 fipy.terms.abstractUpwindConvectionTerm module

33.6 fipy.terms.advectionTerm module

class fipy.terms.advectionTerm.**AdvectionTerm** (*coeff=None*)
 Bases: *fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm*

The *AdvectionTerm* object constructs the *b* vector contribution for the advection term given by

$$u|\nabla\phi|$$

from the advection equation given by:

$$\frac{\partial\phi}{\partial t} + u|\nabla\phi| = 0$$

The construction of the gradient magnitude term requires upwinding as in the standard *FirstOrderAdvectionTerm*. The higher order terms are incorporated as follows. The formula used here is given by:

$$u_P|\nabla\phi|_P = \max(u_P, 0) \left[\sum_A \min(D_{AP}, 0)^2 \right]^{1/2} + \min(u_P, 0) \left[\sum_A \max(D_{AP}, 0)^2 \right]^{1/2}$$

where,

$$D_{AP} = \frac{\phi_A - \phi_P}{d_{AP}} - \frac{d_{AP}}{2} m(L_A, L_P)$$

and

$$\begin{aligned} m(x, y) &= x && \text{if } |x| \leq |y| \forall xy \geq 0 \\ m(x, y) &= y && \text{if } |x| > |y| \forall xy \geq 0 \\ m(x, y) &= 0 && \text{if } xy < 0 \end{aligned}$$

also,

$$\begin{aligned} L_A &= \frac{\phi_{AA} + \phi_P - 2\phi_A}{d_{AP}^2} \\ L_P &= \frac{\phi_A + \phi_{PP} - 2\phi_P}{d_{AP}^2} \end{aligned}$$

Here are some simple test cases for this problem:

```
>>> from fipy.meshes import Grid1D
>>> from fipy.solvers import *
>>> SparseMatrix = LinearPCGSolver()._matrixClass
>>> mesh = Grid1D(dx = 1., nx = 3)
```

Trivial test:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> coeff = CellVariable(mesh = mesh, value = numerix.zeros(3, 'd'))
>>> v, L, b = AdvectionTerm(0.)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, numerix.zeros(3, 'd'), atol = 1e-10))
True
```

Less trivial test:

```
>>> coeff = CellVariable(mesh = mesh, value = numerix.arange(3))
>>> v, L, b = AdvectionTerm(1.)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((0., -1., -1.)), atol = 1e-10))
True
```

Even less trivial

```
>>> coeff = CellVariable(mesh = mesh, value = numerix.arange(3))
>>> v, L, b = AdvectionTerm(-1.)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((1., 1., 0.)), atol = 1e-10))
True
```

Another trivial test case (more trivial than a trivial test case standing on a harpsichord singing “trivial test cases are here again”)

```
>>> vel = numerix.array((-1, 2, -3))
>>> coeff = CellVariable(mesh = mesh, value = numerix.array((4, 6, 1)))
>>> v, L, b = AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, -vel * numerix.array((2, numerix.sqrt(5**2 + 2**2), 5)), atol = 1e-10))
True
```

Somewhat less trivial test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2)
>>> vel = numerix.array((3, -5, -6, -3))
```

(continues on next page)

(continued from previous page)

```
>>> coeff = CellVariable(mesh = mesh, value = numerix.array((3, 1, 6, 7)))
>>> v, L, b = AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
>>> answer = -vel * numerix.array((2, numerix.sqrt(2**2 + 6**2), 1, 0))
>>> print(numerix.allclose(b, answer, atol = 1e-10))
True
```

For the above test cases the *AdvectionTerm* gives the same result as the *AdvectionTerm*. The following test imposes a quadratic field. The higher order term can resolve this field correctly.

$$\phi = x^2$$

The returned vector b should have the value:

$$-|\nabla\phi| = -\left|\frac{\partial\phi}{\partial x}\right| = -2|x|$$

Build the test case in the following way,

```
>>> mesh = Grid1D(dx = 1., nx = 5)
>>> vel = 1.
>>> coeff = CellVariable(mesh = mesh, value = mesh.cellCenters[0]**2)
>>> v, L, b = __AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
```

The first order term is not accurate. The first and last element are ignored because they don't have any neighbors for higher order evaluation

```
>>> print(numerix.allclose(CellVariable(mesh=mesh,
... value=b).globalValue[1:-1], -2 * mesh.cellCenters.globalValue[0][1:-1]))
False
```

The higher order term is spot on.

```
>>> v, L, b = AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(CellVariable(mesh=mesh,
... value=b).globalValue[1:-1], -2 * mesh.cellCenters.globalValue[0][1:-1]))
True
```

The *AdvectionTerm* will also resolve a circular field with more accuracy,

$$\phi = (x^2 + y^2)^{1/2}$$

Build the test case in the following way,

```
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 10, ny = 10)
>>> vel = 1.
>>> x, y = mesh.cellCenters
>>> r = numerix.sqrt(x**2 + y**2)
>>> coeff = CellVariable(mesh = mesh, value = r)
>>> v, L, b = __AdvectionTerm(1.)._buildMatrix(coeff, SparseMatrix)
>>> error = CellVariable(mesh=mesh, value=b + 1)
>>> ans = CellVariable(mesh=mesh, value=b + 1)
>>> ans[(x > 2) & (x < 8) & (y > 2) & (y < 8)] = 0.123105625618
>>> print((error <= ans).all())
True
```

The maximum error is large (about 12 %) for the first order advection.

```
>>> v, L, b = AdvectionTerm(1.)._buildMatrix(coeff, SparseMatrix)
>>> error = CellVariable(mesh=mesh, value=b + 1)
>>> ans = CellVariable(mesh=mesh, value=b + 1)
>>> ans[(x > 2) & (x < 8) & (y > 2) & (y < 8)] = 0.0201715476598
>>> print((error <= ans).all())
True
```

The maximum error is 2 % when using a higher order contribution.

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.advectionTerm'

33.7 fipy.terms.asymmetricConvectionTerm module

33.8 fipy.terms.binaryTerm module

33.9 fipy.terms.cellTerm module

class fipy.terms.cellTerm.**CellTerm**(*coeff=1.0, var=None*)
Bases: fipy.terms.nonDiffusionTerm._NonDiffusionTerm

Attention: This class is abstract. Always create one of its subclasses.

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__init__(*coeff=1.0, var=None*)

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.cellTerm'

33.10 fipy.terms.centralDiffConvectionTerm module

class fipy.terms.centralDiffConvectionTerm.**CentralDifferenceConvectionTerm**(*coeff=1.0, var=None*)
Bases: fipy.terms.abstractConvectionTerm._AbstractConvectionTerm

This *Term* represents

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the central differencing scheme. For further details see *Numerical Schemes*.

Create a `_AbstractConvectionTerm` object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), ↪
↪ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit ↪
↪ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
```

```
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, ↪
↪ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0., ↪
↪ 0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, ↪
↪ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0., ↪
↪ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, ↪
↪ dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,)))) ↪
↪ solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))). ↪
↪ solve(var=cv2, solver=DummySolver(), dt=1.)
```

Parameters `coeff` (The *Term*'s coefficient value.) –

`__module__` = `'fipy.terms.centralDiffConvectionTerm'`

33.11 `fipy.terms.coupledBinaryTerm` module

33.12 `fipy.terms.diffusionTerm` module

class `fipy.terms.diffusionTerm.DiffusionTerm` (`coeff=1.0, var=None`)

Bases: `fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection`

This term represents a higher order diffusion term. The order of the term is determined by the number of *coeffs*, such that:

```
DiffusionTerm(D1)
```

represents a typical 2nd-order diffusion term of the form

$$\nabla \cdot (D_1 \nabla \phi)$$

and:

```
DiffusionTerm((D1,D2))
```

represents a 4th-order Cahn-Hilliard term of the form

$$\nabla \cdot \{D_1 \nabla [\nabla \cdot (D_2 \nabla \phi)]\}$$

and so on.

Create a *Term*.

Parameters `coeff` (`float` or `CellVariable` or `FaceVariable`) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__module__` = `'fipy.terms.diffusionTerm'`

class `fipy.terms.diffusionTerm.DiffusionTermCorrection` (`coeff=1.0, var=None`)

Bases: `fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm`

Create a *Term*.

Parameters `coeff` (`float` or `CellVariable` or `FaceVariable`) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__module__` = `'fipy.terms.diffusionTermCorrection'`

class `fipy.terms.diffusionTerm.DiffusionTermNoCorrection` (`coeff=1.0, var=None`)

Bases: `fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm`

Create a *Term*.

Parameters `coeff` (`float` or `CellVariable` or `FaceVariable`) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__module__` = `'fipy.terms.diffusionTermNoCorrection'`

33.13 fipy.terms.diffusionTermCorrection module

```
class fipy.terms.diffusionTermCorrection.DiffusionTermCorrection (coeff=1.0,  
                                                             var=None)
```

Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.diffusionTermCorrection'
```

33.14 fipy.terms.diffusionTermNoCorrection module

```
class fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection (coeff=1.0,  
                                                             var=None)
```

Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.diffusionTermNoCorrection'
```

33.15 fipy.terms.explicitDiffusionTerm module

```
class fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm (coeff=1.0,  
                                                             var=None)
```

Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm

The discretization for the *ExplicitDiffusionTerm* is given by

$$\int_V \nabla \cdot (\Gamma \nabla \phi) dV \simeq \sum_f \Gamma_f \frac{\phi_A^{\text{old}} - \phi_P^{\text{old}}}{d_{AP}} A_f$$

where ϕ_A^{old} and ϕ_P^{old} are the old values of the variable. The term is added to the RHS vector and makes no contribution to the solution matrix.

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.explicitDiffusionTerm'
```

33.16 fipy.terms.explicitSourceTerm module

33.17 fipy.terms.explicitUpwindConvectionTerm module

class fipy.terms.explicitUpwindConvectionTerm.**ExplicitUpwindConvectionTerm**(*coeff=1.0*,
var=None)
 Bases: fipy.terms.abstractUpwindConvectionTerm._AbstractUpwindConvectionTerm

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P^{\text{old}} + (1 - \alpha_f) \phi_A^{\text{old}}$ and α_f is calculated using the upwind scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]),
↪ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↪ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
```

```
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
```

(continues on next page)

(continued from previous page)

```

>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↪ 0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↪ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))))
↪ solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).
↪ solve(var=cv2, solver=DummySolver(), dt=1.)

```

Parameters **coeff** (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.explicitUpwindConvectionTerm'
```

33.18 fipy.terms.exponentialConvectionTerm module

class fipy.terms.exponentialConvectionTerm.**ExponentialConvectionTerm** (*coeff=1.0*,
var=None)

Bases: fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the exponential scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```

>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)

```

(continues on next page)

(continued from previous page)

```

__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]),
↳mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↳convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↳ solver=DummySolver(), dt=1.)

```

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↳solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↳0.,  0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↳dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↳0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↳dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,)))).
↳solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).
↳solve(var=cv2, solver=DummySolver(), dt=1.)

```

Parameters *coeff* (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.exponentialConvectionTerm'
```

33.19 fipy.terms.faceTerm module

```

class fipy.terms.faceTerm.FaceTerm(coeff=1.0, var=None)
Bases: fipy.terms.nonDiffusionTerm._NonDiffusionTerm

```

Attention: This class is abstract. Always create one of its subclasses.

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__init__` (*coeff*=1.0, *var*=None)
Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__module__` = 'fipy.terms.faceTerm'

33.20 fipy.terms.firstOrderAdvectionTerm module

class `fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm` (*coeff*=None)
Bases: `fipy.terms.nonDiffusionTerm._NonDiffusionTerm`

The *FirstOrderAdvectionTerm* object constructs the b vector contribution for the advection term given by

$$u|\nabla\phi|$$

from the advection equation given by:

$$\frac{\partial\phi}{\partial t} + u|\nabla\phi| = 0$$

The construction of the gradient magnitude term requires upwinding. The formula used here is given by:

$$u_P|\nabla\phi|_P = \max(u_P, 0) \left[\sum_A \min\left(\frac{\phi_A - \phi_P}{d_{AP}}, 0\right)^2 \right]^{1/2} + \min(u_P, 0) \left[\sum_A \max\left(\frac{\phi_A - \phi_P}{d_{AP}}, 0\right)^2 \right]^{1/2}$$

Here are some simple test cases for this problem:

```
>>> from fipy.meshes import Grid1D
>>> from fipy.solvers import *
>>> SparseMatrix = LinearLUSolver()._matrixClass
>>> mesh = Grid1D(dx = 1., nx = 3)
>>> from fipy.variables.cellVariable import CellVariable
```

Trivial test:

```
>>> var = CellVariable(value = numerix.zeros(3, 'd'), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(0.)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, numerix.zeros(3, 'd'), atol = 1e-10))
True
```

Less trivial test:

```
>>> var = CellVariable(value = numerix.arange(3), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(1.)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((0., -1., -1.)), atol = 1e-10))
True
```

Even less trivial

```
>>> var = CellVariable(value = numerix.arange(3), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(-1.)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((1., 1., 0.)), atol = 1e-10))
True
```

Another trivial test case (more trivial than a trivial test case standing on a harpsichord singing “trivial test cases are here again”)

```
>>> vel = numerix.array((-1, 2, -3))
>>> var = CellVariable(value = numerix.array((4, 6, 1)), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(vel)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, -vel * numerix.array((2, numerix.sqrt(5**2 + 2**2),
↪5))), atol = 1e-10))
True
```

Somewhat less trivial test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2)
>>> vel = numerix.array((3, -5, -6, -3))
>>> var = CellVariable(value = numerix.array((3, 1, 6, 7)), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(vel)._buildMatrix(var, SparseMatrix)
>>> answer = -vel * numerix.array((2, numerix.sqrt(2**2 + 6**2), 1, 0))
>>> print(numerix.allclose(b, answer, atol = 1e-10))
True
```

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__init__ (*coeff=None*)

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.firstOrderAdvectionTerm'

33.21 fipy.terms.hybridConvectionTerm module

class fipy.terms.hybridConvectionTerm.**HybridConvectionTerm** (*coeff=1.0, var=None*)

Bases: fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the hybrid scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
```

(continues on next page)

(continued from previous page)

```

Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), ↪
↪mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit ↪
↪convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
    
```

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, ↪
↪solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0., ↪
↪ 0.,  0.,  0.,  0.],
↪ [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, ↪
↪dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0., ↪
↪ 0.],
↪ [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, ↪
↪dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,)))) ↪
↪.solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))). ↪
↪.solve(var=cv2, solver=DummySolver(), dt=1.)
    
```

Parameters **coeff** (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.hybridConvectionTerm'
```

33.22 fipy.terms.implicitDiffusionTerm module

`fipy.terms.implicitDiffusionTerm.ImplicitDiffusionTerm`
 alias of `fipy.terms.diffusionTerm.DiffusionTerm`

33.23 fipy.terms.implicitSourceTerm module

class `fipy.terms.implicitSourceTerm.ImplicitSourceTerm` (*coeff=0.0, var=None*)
 Bases: `fipy.terms.sourceTerm.SourceTerm`

The *ImplicitSourceTerm* represents

$$\int_V \phi S dV \simeq \phi_P S_P V_P$$

where S is the *coeff* value.

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__module__` = `'fipy.terms.implicitSourceTerm'`

33.24 fipy.terms.nonDiffusionTerm module

33.25 fipy.terms.powerLawConvectionTerm module

class `fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm` (*coeff=1.0, var=None*)
 Bases: `fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm`

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u} \phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the power law scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoefficientError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```

...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]), ↪
↪mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit ↪
↪convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↪ solver=DummySolver(), dt=1.)

```

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv, ↪
↪solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0., ↪
↪ 0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, ↪
↪dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0., ↪
↪ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, ↪
↪dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,)))) ↪
↪solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))). ↪
↪solve(var=cv2, solver=DummySolver(), dt=1.)

```

Parameters **coeff** (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.powerLawConvectionTerm'
```

33.26 fipy.terms.residualTerm module

class `fipy.terms.residualTerm.ResidualTerm`(*equation*, *underRelaxation*=1.0)

Bases: `fipy.terms.explicitSourceTerm._ExplicitSourceTerm`

The *ResidualTerm* is a special form of explicit *SourceTerm* that adds the residual of one equation to another equation. Useful for Newton's method.

Create a *Term*.

Parameters `coeff` (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__init__` (*equation*, *underRelaxation*=1.0)

Create a *Term*.

Parameters `coeff` (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__module__` = `'fipy.terms.residualTerm'`

`__repr__` ()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

33.27 fipy.terms.sourceTerm module

class `fipy.terms.sourceTerm.SourceTerm`(*coeff*=0.0, *var*=None)

Bases: `fipy.terms.cellTerm.CellTerm`

Attention: This class is abstract. Always create one of its subclasses.

Create a *Term*.

Parameters `coeff` (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__init__` (*coeff*=0.0, *var*=None)

Create a *Term*.

Parameters `coeff` (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

`__module__` = `'fipy.terms.sourceTerm'`

33.28 fipy.terms.term module

class `fipy.terms.term.Term` (*coeff=1.0, var=None*)

Bases: `object`

Attention: This class is abstract. Always create one of its subclasses.

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

property **RHSvector**

Return the RHS vector calculated in *solve()* or *sweep()*. The *cacheRHSvector()* method should be called before *solve()* or *sweep()* to cache the vector.

__add__ (*other*)

__and__ (*other*)

__dict__ = `mappingproxy({'__module__': 'fipy.terms.term', '__doc__': '\n .. attention`

__div__ (*other*)

__eq__ (*other*)

Return self==value.

__hash__ ()

Return hash(self).

__init__ (*coeff=1.0, var=None*)

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.term'

__mul__ (*other*)

__neg__ ()

__pos__ ()

__radd__ (*other*)

__rand__ (*other*)

__repr__ ()

Return repr(self).

__rmul__ (*other*)

__rsub__ (*other*)

__sub__ (*other*)

__truediv__ (*other*)

__weakref__

list of weak references to the object (if defined)

cacheMatrix()

Informs *solve()* and *sweep()* to cache their matrix so that *matrix* can return the matrix.

cacheRHSvector()

Informs *solve()* and *sweep()* to cache their right hand side vector so that *getRHSvector()* can return it.

copy()

getDefaultSolver (*var=None, solver=None, *args, **kwargs*)

justErrorVector (*var=None, solver=None, boundaryConditions=(), dt=1.0, underRelaxation=None, residualFn=None*)

Builds the *Term*'s linear system once.

This method also recalculates and returns the error as well as applying under-relaxation.

justErrorVector returns the overlapping local value in parallel (not the non-overlapping value).

```
>>> from fipy.solvers import DummySolver
>>> from fipy import *
>>> m = Grid1D(nx=10)
>>> v = CellVariable(mesh=m)
>>> len(DiffusionTerm().justErrorVector(v, solver=DummySolver())) == m.
↳ numberOfCells
True
```

Parameters

- **var** (*CellVariable*) – Variable to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- **solver** (*Solver*) – Iterative solver to be used to solve the linear system of equations. The default solver depends on the solver package selected.
- **boundaryConditions** (*tuple of BoundaryCondition*) –
- **dt** (*float*) – Timestep size.
- **underRelaxation** (*float*) – Usually a value between 0 and 1 or *None* in the case of no under-relaxation
- **residualFn** (*function*) – Takes *var*, *matrix*, and *RHSvector* arguments, used to customize the residual calculation.

justResidualVector (*var=None, solver=None, boundaryConditions=(), dt=None, underRelaxation=None, residualFn=None*)

Builds the *Term*'s linear system once.

This method also recalculates and returns the residual as well as applying under-relaxation.

justResidualVector returns the overlapping local value in parallel (not the non-overlapping value).

```
>>> from fipy import *
>>> m = Grid1D(nx=10)
>>> v = CellVariable(mesh=m)
>>> len(numerix.asarray(DiffusionTerm().justResidualVector(v))) == m.
↳ numberOfCells
True
```

Parameters

- **var** (*CellVariable*) – *Variable* to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- **solver** (*Solver*) – Iterative solver to be used to solve the linear system of equations. The default solver depends on the solver package selected.
- **boundaryConditions** (*tuple* of *BoundaryCondition*) –
- **dt** (*float*) – Timestep size.
- **underRelaxation** (*float*) – Usually a value between 0 and 1 or *None* in the case of no under-relaxation
- **residualFn** (*function*) – Takes *var*, *matrix*, and *RHSvector* arguments, used to customize the residual calculation.

property matrix

Return the matrix calculated in *solve()* or *sweep()*. The *cacheMatrix()* method should be called before *solve()* or *sweep()* to cache the matrix.

residualVectorAndNorm (*var=None, solver=None, boundaryConditions=(), dt=None, underRelaxation=None, residualFn=None*)

Builds the *Term*'s linear system once.

This method also recalculates and returns the residual as well as applying under-relaxation.

Parameters

- **var** (*CellVariable*) – *Variable* to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- **solver** (*Solver*) – Iterative solver to be used to solve the linear system of equations. The default solver depends on the solver package selected.
- **boundaryConditions** (*tuple* of *BoundaryCondition*) –
- **dt** (*float*) – Timestep size.
- **underRelaxation** (*float*) – Usually a value between 0 and 1 or *None* in the case of no under-relaxation
- **residualFn** (*function*) – Takes *var*, *matrix*, and *RHSvector* arguments, used to customize the residual calculation.

solve (*var=None, solver=None, boundaryConditions=(), dt=None*)

Builds and solves the *Term*'s linear system once. This method does not return the residual. It should be used when the residual is not required.

Parameters

- **var** (*CellVariable*) – *Variable* to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- **solver** (*Solver*) – Iterative solver to be used to solve the linear system of equations. The default solver depends on the solver package selected.
- **boundaryConditions** (*tuple* of *BoundaryCondition*) –
- **dt** (*float*) – Timestep size.

sweep (*var=None, solver=None, boundaryConditions=(), dt=None, underRelaxation=None, residualFn=None, cacheResidual=False, cacheError=False*)

Builds and solves the *Term*'s linear system once. This method also recalculates and returns the residual as well as applying under-relaxation.

Parameters

- **var** (*CellVariable*) – *Variable* to be solved for. Provides the initial condition, the old value and holds the solution on completion.
- **solver** (*Solver*) – Iterative solver to be used to solve the linear system of equations. The default solver depends on the solver package selected.
- **boundaryConditions** (*tuple* of *BoundaryCondition*) –
- **dt** (*float*) – Timestep size.
- **underRelaxation** (*float*) – Usually a value between 0 and 1 or *None* in the case of no under-relaxation
- **residualFn** (*function*) – Takes *var*, *matrix*, and *RHSvector* arguments, used to customize the residual calculation.
- **cacheResidual** (*bool*) – If *True*, calculate and store the residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (*bool*) – If *True*, use the residual vector \vec{r} to solve $\mathbf{L}\vec{e} = \vec{r}$ for the error vector \vec{e} and store it in the *errorVector* member of *Term*

33.29 fipy.terms.test module

33.30 fipy.terms.transientTerm module

class fipy.terms.transientTerm.**TransientTerm** (*coeff=1.0, var=None*)

Bases: *fipy.terms.cellTerm.CellTerm*

The *TransientTerm* represents

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{(\rho_P\phi_P - \rho_P^{\text{old}}\phi_P^{\text{old}})V_P}{\Delta t}$$

where ρ is the *coeff* value.

The following test case verifies that variable coefficients and old coefficient values work correctly. We will solve the following equation

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

The analytic solution is given by

$$\phi = \sqrt{\phi_0^2 + kt},$$

where ϕ_0 is the initial value.

```
>>> phi0 = 1.
>>> k = 1.
>>> dt = 1.
>>> relaxationFactor = 1.5
>>> steps = 2
>>> sweeps = 8
```

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = phi0, hasOld = 1)
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm
```

Relaxation, given by *relaxationFactor*, is required for a converged solution.

```
>>> eq = TransientTerm(var) == ImplicitSourceTerm(-relaxationFactor) \
...      + var * relaxationFactor + k
```

A number of sweeps at each time step are required to let the relaxation take effect.

```
>>> from builtins import range
>>> for step in range(steps):
...     var.updateOld()
...     for sweep in range(sweeps):
...         eq.solve(var, dt = dt)
```

Compare the final result with the analytical solution.

```
>>> from fipy.tools import numerix
>>> print(var.allclose(numerix.sqrt(k * dt * steps + phi0**2)))
1
```

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.transientTerm'
```

33.31 fipy.terms.unaryTerm module

33.32 fipy.terms.upwindConvectionTerm module

class fipy.terms.upwindConvectionTerm.**UpwindConvectionTerm** (*coeff=1.0, var=None*)

Bases: fipy.terms.abstractUpwindConvectionTerm._AbstractUpwindConvectionTerm

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the upwind convection scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
```

(continues on next page)

(continued from previous page)

```

>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]) ,
↪ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↪ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↪ solver=DummySolver(), dt=1.)

```

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↪ 0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↪ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))))
↪ solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).
↪ solve(var=cv2, solver=DummySolver(), dt=1.)

```

Parameters **coeff** (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.upwindConvectionTerm'
```

33.33 fipy.terms.vanLeerConvectionTerm module

class fipy.terms.vanLeerConvectionTerm.**VanLeerConvectionTerm** (*coeff=1.0*,
var=None
 Bases: *fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm*
 Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]),
↪ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↪ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
```

```
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↪ 0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↪ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
```

(continues on next page)

(continued from previous page)

```
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))))).
↳ solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).
↳ solve(var=cv2, solver=DummySolver(), dt=1.)
```

Parameters **coeff** (The *Term*'s coefficient value.) –

__module__ = `'fipy.terms.vanLeerConvectionTerm'`

33.34 Module contents

exception `fipy.terms.ExplicitVariableError` (*s='Terms with explicit Variables cannot mix with Terms with implicit Variables.'*)

Bases: `Exception`

__init__ (*s='Terms with explicit Variables cannot mix with Terms with implicit Variables.'*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = `'fipy.terms'`

__weakref__
list of weak references to the object (if defined)

exception `fipy.terms.TermMultiplyError` (*s='Must multiply terms by int or float.'*)

Bases: `Exception`

__init__ (*s='Must multiply terms by int or float.'*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = `'fipy.terms'`

__weakref__
list of weak references to the object (if defined)

exception `fipy.terms.AbstractBaseClassError` (*s="can't instantiate abstract base class"*)

Bases: `NotImplementedError`

__init__ (*s="can't instantiate abstract base class"*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = `'fipy.terms'`

__weakref__
list of weak references to the object (if defined)

exception `fipy.terms.VectorCoeffError` (*s='The coefficient must be a vector value.'*)

Bases: `TypeError`

__init__ (*s='The coefficient must be a vector value.'*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = `'fipy.terms'`

__weakref__
list of weak references to the object (if defined)

exception `fipy.terms.SolutionVariableNumberError` (*s='Different number of solution variables and equations.'*)

Bases: `Exception`

__init__ (*s='Different number of solution variables and equations.'*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.terms'

__weakref__
list of weak references to the object (if defined)

exception `fipy.terms.SolutionVariableRequiredError` (*s='The solution variable needs to be specified.'*)

Bases: `Exception`

__init__ (*s='The solution variable needs to be specified.'*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.terms'

__weakref__
list of weak references to the object (if defined)

exception `fipy.terms.IncorrectSolutionVariable` (*s='The solution variable is incorrect.'*)

Bases: `Exception`

__init__ (*s='The solution variable is incorrect.'*)
Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.terms'

__weakref__
list of weak references to the object (if defined)

`fipy.terms.ConvectionTerm`

alias of `fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm`

class `fipy.terms.FirstOrderAdvectionTerm` (*coeff=None*)

Bases: `fipy.terms.nonDiffusionTerm._NonDiffusionTerm`

The *FirstOrderAdvectionTerm* object constructs the b vector contribution for the advection term given by

$$u|\nabla\phi|$$

from the advection equation given by:

$$\frac{\partial\phi}{\partial t} + u|\nabla\phi| = 0$$

The construction of the gradient magnitude term requires upwinding. The formula used here is given by:

$$u_P|\nabla\phi|_P = \max(u_P, 0) \left[\sum_A \min\left(\frac{\phi_A - \phi_P}{d_{AP}}, 0\right) \right]^{1/2} + \min(u_P, 0) \left[\sum_A \max\left(\frac{\phi_A - \phi_P}{d_{AP}}, 0\right) \right]^{1/2}$$

Here are some simple test cases for this problem:

```
>>> from fipy.meshes import Grid1D
>>> from fipy.solvers import *
>>> SparseMatrix = LinearLUSolver()._matrixClass
>>> mesh = Grid1D(dx = 1., nx = 3)
>>> from fipy.variables.cellVariable import CellVariable
```

Trivial test:

```
>>> var = CellVariable(value = numerix.zeros(3, 'd'), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(0.)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, numerix.zeros(3, 'd'), atol = 1e-10))
True
```

Less trivial test:

```
>>> var = CellVariable(value = numerix.arange(3), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(1.)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((0., -1., -1.)), atol = 1e-10))
True
```

Even less trivial

```
>>> var = CellVariable(value = numerix.arange(3), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(-1.)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((1., 1., 0.)), atol = 1e-10))
True
```

Another trivial test case (more trivial than a trivial test case standing on a harpsichord singing “trivial test cases are here again”)

```
>>> vel = numerix.array((-1, 2, -3))
>>> var = CellVariable(value = numerix.array((4, 6, 1)), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(vel)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, -vel * numerix.array((2, numerix.sqrt(5**2 + 2**2), 5)), atol = 1e-10))
True
```

Somewhat less trivial test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2)
>>> vel = numerix.array((3, -5, -6, -3))
>>> var = CellVariable(value = numerix.array((3, 1, 6, 7)), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(vel)._buildMatrix(var, SparseMatrix)
>>> answer = -vel * numerix.array((2, numerix.sqrt(2**2 + 6**2), 1, 0))
>>> print(numerix.allclose(b, answer, atol = 1e-10))
True
```

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__init__ (*coeff=None*)

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.firstOrderAdvectionTerm'

class fipy.terms.AdvectionTerm (*coeff=None*)

Bases: *fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm*

The *AdvectionTerm* object constructs the *b* vector contribution for the advection term given by

$$u|\nabla\phi|$$

from the advection equation given by:

$$\frac{\partial \phi}{\partial t} + u|\nabla \phi| = 0$$

The construction of the gradient magnitude term requires upwinding as in the standard *FirstOrderAdvectionTerm*. The higher order terms are incorporated as follows. The formula used here is given by:

$$u_P |\nabla \phi|_P = \max(u_P, 0) \left[\sum_A \min(D_{AP}, 0)^2 \right]^{1/2} + \min(u_P, 0) \left[\sum_A \max(D_{AP}, 0)^2 \right]^{1/2}$$

where,

$$D_{AP} = \frac{\phi_A - \phi_P}{d_{AP}} - \frac{d_{AP}}{2} m(L_A, L_P)$$

and

$$\begin{aligned} m(x, y) &= x && \text{if } |x| \leq |y| \forall xy \geq 0 \\ m(x, y) &= y && \text{if } |x| > |y| \forall xy \geq 0 \\ m(x, y) &= 0 && \text{if } xy < 0 \end{aligned}$$

also,

$$\begin{aligned} L_A &= \frac{\phi_{AA} + \phi_P - 2\phi_A}{d_{AP}^2} \\ L_P &= \frac{\phi_A + \phi_{PP} - 2\phi_P}{d_{AP}^2} \end{aligned}$$

Here are some simple test cases for this problem:

```
>>> from fipy.meshes import Grid1D
>>> from fipy.solvers import *
>>> SparseMatrix = LinearPCGSolver()._matrixClass
>>> mesh = Grid1D(dx = 1., nx = 3)
```

Trivial test:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> coeff = CellVariable(mesh = mesh, value = numerix.zeros(3, 'd'))
>>> v, L, b = AdvectionTerm(0.)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, numerix.zeros(3, 'd'), atol = 1e-10))
True
```

Less trivial test:

```
>>> coeff = CellVariable(mesh = mesh, value = numerix.arange(3))
>>> v, L, b = AdvectionTerm(1.)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((0., -1., -1.)), atol = 1e-10))
True
```

Even less trivial

```
>>> coeff = CellVariable(mesh = mesh, value = numerix.arange(3))
>>> v, L, b = AdvectionTerm(-1.)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((1., 1., 0.)), atol = 1e-10))
True
```

Another trivial test case (more trivial than a trivial test case standing on a harpsichord singing “trivial test cases are here again”)

```
>>> vel = numerix.array((-1, 2, -3))
>>> coeff = CellVariable(mesh = mesh, value = numerix.array((4, 6, 1)))
>>> v, L, b = AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, -vel * numerix.array((2, numerix.sqrt(5**2 + 2**2),
↪5)), atol = 1e-10))
True
```

Somewhat less trivial test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2)
>>> vel = numerix.array((3, -5, -6, -3))
>>> coeff = CellVariable(mesh = mesh, value = numerix.array((3, 1, 6, 7)))
>>> v, L, b = AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
>>> answer = -vel * numerix.array((2, numerix.sqrt(2**2 + 6**2), 1, 0))
>>> print(numerix.allclose(b, answer, atol = 1e-10))
True
```

For the above test cases the *AdvectionTerm* gives the same result as the *AdvectionTerm*. The following test imposes a quadratic field. The higher order term can resolve this field correctly.

$$\phi = x^2$$

The returned vector *b* should have the value:

$$-\left|\nabla\phi\right| = -\left|\frac{\partial\phi}{\partial x}\right| = -2|x|$$

Build the test case in the following way,

```
>>> mesh = Grid1D(dx = 1., nx = 5)
>>> vel = 1.
>>> coeff = CellVariable(mesh = mesh, value = mesh.cellCenters[0]**2)
>>> v, L, b = __AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
```

The first order term is not accurate. The first and last element are ignored because they don’t have any neighbors for higher order evaluation

```
>>> print(numerix.allclose(CellVariable(mesh=mesh,
... value=b).globalValue[1:-1], -2 * mesh.cellCenters.globalValue[0][1:-1]))
False
```

The higher order term is spot on.

```
>>> v, L, b = AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(CellVariable(mesh=mesh,
... value=b).globalValue[1:-1], -2 * mesh.cellCenters.globalValue[0][1:-1]))
True
```

The *AdvectionTerm* will also resolve a circular field with more accuracy,

$$\phi = (x^2 + y^2)^{1/2}$$

Build the test case in the following way,

```

>>> mesh = Grid2D(dx = 1., dy = 1., nx = 10, ny = 10)
>>> vel = 1.
>>> x, y = mesh.cellCenters
>>> r = numerix.sqrt(x**2 + y**2)
>>> coeff = CellVariable(mesh = mesh, value = r)
>>> v, L, b = __AdvectionTerm(1.)._buildMatrix(coeff, SparseMatrix)
>>> error = CellVariable(mesh=mesh, value=b + 1)
>>> ans = CellVariable(mesh=mesh, value=b + 1)
>>> ans[(x > 2) & (x < 8) & (y > 2) & (y < 8)] = 0.123105625618
>>> print((error <= ans).all())
True

```

The maximum error is large (about 12 %) for the first order advection.

```

>>> v, L, b = AdvectionTerm(1.)._buildMatrix(coeff, SparseMatrix)
>>> error = CellVariable(mesh=mesh, value=b + 1)
>>> ans = CellVariable(mesh=mesh, value=b + 1)
>>> ans[(x > 2) & (x < 8) & (y > 2) & (y < 8)] = 0.0201715476598
>>> print((error <= ans).all())
True

```

The maximum error is 2 % when using a higher order contribution.

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.advectionTerm'

class fipy.terms.**TransientTerm** (*coeff=1.0, var=None*)

Bases: *fipy.terms.cellTerm.CellTerm*

The *TransientTerm* represents

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{(\rho_P\phi_P - \rho_P^{\text{old}}\phi_P^{\text{old}})V_P}{\Delta t}$$

where ρ is the *coeff* value.

The following test case verifies that variable coefficients and old coefficient values work correctly. We will solve the following equation

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

The analytic solution is given by

$$\phi = \sqrt{\phi_0^2 + kt},$$

where ϕ_0 is the initial value.

```

>>> phi0 = 1.
>>> k = 1.
>>> dt = 1.
>>> relaxationFactor = 1.5
>>> steps = 2
>>> sweeps = 8

```

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = phi0, hasOld = 1)
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm
```

Relaxation, given by *relaxationFactor*, is required for a converged solution.

```
>>> eq = TransientTerm(var) == ImplicitSourceTerm(-relaxationFactor) \
...      + var * relaxationFactor + k
```

A number of sweeps at each time step are required to let the relaxation take effect.

```
>>> from builtins import range
>>> for step in range(steps):
...     var.updateOld()
...     for sweep in range(sweeps):
...         eq.solve(var, dt = dt)
```

Compare the final result with the analytical solution.

```
>>> from fipy.tools import numerix
>>> print(var.allclose(numerix.sqrt(k * dt * steps + phi0**2)))
1
```

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.transientTerm'

class fipy.terms.DiffusionTerm (*coeff=1.0, var=None*)

Bases: *fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection*

This term represents a higher order diffusion term. The order of the term is determined by the number of *coeffs*, such that:

```
DiffusionTerm(D1)
```

represents a typical 2nd-order diffusion term of the form

$$\nabla \cdot (D_1 \nabla \phi)$$

and:

```
DiffusionTerm((D1,D2))
```

represents a 4th-order Cahn-Hilliard term of the form

$$\nabla \cdot \{D_1 \nabla [\nabla \cdot (D_2 \nabla \phi)]\}$$

and so on.

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.diffusionTerm'
```

```
class fipy.terms.DiffusionTermCorrection (coeff=1.0, var=None)
```

```
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm
```

Create a *Term*.

Parameters `coeff` (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.diffusionTermCorrection'
```

```
class fipy.terms.DiffusionTermNoCorrection (coeff=1.0, var=None)
```

```
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm
```

Create a *Term*.

Parameters `coeff` (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.diffusionTermNoCorrection'
```

```
class fipy.terms.DiffusionTermCorrection (coeff=1.0, var=None)
```

```
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm
```

Create a *Term*.

Parameters `coeff` (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.diffusionTermCorrection'
```

```
class fipy.terms.DiffusionTermNoCorrection (coeff=1.0, var=None)
```

```
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm
```

Create a *Term*.

Parameters `coeff` (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.diffusionTermNoCorrection'
```

```
class fipy.terms.ExplicitDiffusionTerm (coeff=1.0, var=None)
```

```
Bases: fipy.terms.abstractDiffusionTerm._AbstractDiffusionTerm
```

The discretization for the *ExplicitDiffusionTerm* is given by

$$\int_V \nabla \cdot (\Gamma \nabla \phi) dV \simeq \sum_f \Gamma_f \frac{\phi_A^{\text{old}} - \phi_P^{\text{old}}}{d_{AP}} A_f$$

where ϕ_A^{old} and ϕ_P^{old} are the old values of the variable. The term is added to the RHS vector and makes no contribution to the solution matrix.

Create a *Term*.

Parameters `coeff` (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

```
__module__ = 'fipy.terms.explicitDiffusionTerm'
```

```
fipy.terms.ImplicitDiffusionTerm
```

alias of `fipy.terms.diffusionTerm.DiffusionTerm`

class fipy.terms.**ImplicitSourceTerm** (*coeff=0.0, var=None*)

Bases: *fipy.terms.sourceTerm.SourceTerm*

The *ImplicitSourceTerm* represents

$$\int_V \phi S dV \simeq \phi_P S_P V_P$$

where S is the *coeff* value.

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.implicitSourceTerm'

class fipy.terms.**ResidualTerm** (*equation, underRelaxation=1.0*)

Bases: *fipy.terms.explicitSourceTerm._ExplicitSourceTerm*

The *ResidualTerm* is a special form of explicit *SourceTerm* that adds the residual of one equation to another equation. Useful for Newton's method.

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__init__ (*equation, underRelaxation=1.0*)

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.residualTerm'

__repr__ ()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

class fipy.terms.**CentralDifferenceConvectionTerm** (*coeff=1.0, var=None*)

Bases: *fipy.terms.abstractConvectionTerm._AbstractConvectionTerm*

This *Term* represents

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the central differencing scheme. For further details see [Numerical Schemes](#).

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
```

(continues on next page)

(continued from previous page)

```

Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↳ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]),
↳ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↳ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↳ solver=DummySolver(), dt=1.)

```

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↳ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↳ 0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↳ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↳ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↳ dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))))
↳ .solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).
↳ .solve(var=cv2, solver=DummySolver(), dt=1.)

```

Parameters `coeff` (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.centralDiffConvectionTerm'
```

```
class fipy.terms.ExplicitUpwindConvectionTerm(coeff=1.0, var=None)
```

```
Bases: fipy.terms.abstractUpwindConvectionTerm._AbstractUpwindConvectionTerm
```

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P^{\text{old}} + (1 - \alpha_f) \phi_A^{\text{old}}$ and α_f is calculated using the upwind scheme. For further details see *Numerical Schemes*.

Create a `_AbstractConvectionTerm` object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]),
↪ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↪ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
```

```
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↪ 0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↪ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
```

(continues on next page)

(continued from previous page)

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,)))) .
↳ solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))) .
↳ solve(var=cv2, solver=DummySolver(), dt=1.)

```

Parameters **coeff** (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.explicitUpwindConvectionTerm'
```

```
class fipy.terms.ExponentialConvectionTerm(coeff=1.0, var=None)
```

```
Bases: fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm
```

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the exponential scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```

>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↳ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]) ,
↳ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↳ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↳ solver=DummySolver(), dt=1.)

```

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↳ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...

```

(continues on next page)

(continued from previous page)

```

VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↪ 0.,  0.,  0.,  0.],
      [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↪ 0.],
      [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))))).
↪ solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).
↪ solve(var=cv2, solver=DummySolver(), dt=1.)

```

Parameters *coeff* (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.exponentialConvectionTerm'
```

```
class fipy.terms.HybridConvectionTerm(coeff=1.0, var=None)
```

Bases: `fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm`

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the hybrid scheme. For further details see [Numerical Schemes](#).

Create a `_AbstractConvectionTerm` object.

```

>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]),
↪ mesh=UniformGrid1D(dx=1.0, nx=2)))

```

(continues on next page)

(continued from previous page)

```
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↳ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↳ solver=DummySolver(), dt=1.)
```

```
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↳ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↳ 0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↳ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↳ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↳ dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))))
↳ solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).
↳ solve(var=cv2, solver=DummySolver(), dt=1.)
```

Parameters `coeff` (The *Term*'s coefficient value.) –

`__module__` = 'fipy.terms.hybridConvectionTerm'

class fipy.terms.PowerLawConvectionTerm(*coeff=1.0, var=None*)

Bases: fipy.terms.asymmetricConvectionTerm._AsymmetricConvectionTerm

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the power law scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
```

(continues on next page)

(continued from previous page)

```

>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]) ,
↪ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↪ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↪ solver=DummySolver(), dt=1.)

```

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↪ 0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↪ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,)))).
↪ solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).
↪ solve(var=cv2, solver=DummySolver(), dt=1.)

```

Parameters *coeff* (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.powerLawConvectionTerm'
```

```
class fipy.terms.UpwindConvectionTerm(coeff=1.0, var=None)
```

```
    Bases: fipy.terms.abstractUpwindConvectionTerm._AbstractUpwindConvectionTerm
```

The discretization for this *Term* is given by

$$\int_V \nabla \cdot (\vec{u}\phi) dV \simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f$$

where $\phi_f = \alpha_f \phi_P + (1 - \alpha_f) \phi_A$ and α_f is calculated using the upwind convection scheme. For further details see *Numerical Schemes*.

Create a *_AbstractConvectionTerm* object.

```
>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↪ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]) ,
↪ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↪ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
```

```
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↪ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↪ 0.,  0.,  0.,  0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.,  0.,  0.,  0.,
↪ 0.],
[ 0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2,
↪ dy=1.0, ny=1)))
```

(continues on next page)

(continued from previous page)

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))))
↳ solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).
↳ solve(var=cv2, solver=DummySolver(), dt=1.)

```

Parameters **coeff** (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.upwindConvectionTerm'
```

class `fipy.terms.VanLeerConvectionTerm` (*coeff=1.0, var=None*)

Bases: `fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm`

Create a `_AbstractConvectionTerm` object.

```

>>> from fipy import *
>>> m = Grid1D(nx = 2)
>>> cv = CellVariable(mesh = m)
>>> fv = FaceVariable(mesh = m)
>>> vcv = CellVariable(mesh=m, rank=1)
>>> vfv = FaceVariable(mesh=m, rank=1)
>>> __ConvectionTerm(coeff = cv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = vcv)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.
↳ ]]), mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = vfv)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0.,  0.,  0.]]),
↳ mesh=UniformGrid1D(dx=1.0, nx=2)))
>>> __ConvectionTerm(coeff = (1,))
__ConvectionTerm(coeff=(1,))
>>> ExplicitUpwindConvectionTerm(coeff = (0,)).solve(var=cv, solver=DummySolver())
Traceback (most recent call last):
...
TransientTermError: The equation requires a TransientTerm with explicit_
↳ convection.
>>> (TransientTerm(0.) - ExplicitUpwindConvectionTerm(coeff = (0,))).solve(var=cv,
↳ solver=DummySolver(), dt=1.)

```

```

>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = 1)).solve(var=cv,
↳ solver=DummySolver(), dt=1.)
Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,
↳ 0.,  0.,  0.,  0.,  0.],

```

(continues on next page)

(continued from previous page)

```
[ 0., 0., 0., 0., 0., 0., 0.]], mesh=UniformGrid2D(dx=1.0, nx=2,
↳dy=1.0, ny=1)))
>>> __ConvectionTerm(coeff=vfv2)
__ConvectionTerm(coeff=FaceVariable(value=array([[ 0., 0., 0., 0., 0., 0.,
↳0.],
[ 0., 0., 0., 0., 0., 0., 0.]], mesh=UniformGrid2D(dx=1.0, nx=2,
↳dy=1.0, ny=1)))
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = ((0,), (0,))))
↳solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0)))
↳solve(var=cv2, solver=DummySolver(), dt=1.)
```

Parameters **coeff** (The *Term*'s coefficient value.) –

```
__module__ = 'fipy.terms.vanLeerConvectionTerm'
```

```
class fipy.terms.FirstOrderAdvectionTerm(coeff=None)
```

```
Bases: fipy.terms.nonDiffusionTerm._NonDiffusionTerm
```

The *FirstOrderAdvectionTerm* object constructs the b vector contribution for the advection term given by

$$u|\nabla\phi|$$

from the advection equation given by:

$$\frac{\partial\phi}{\partial t} + u|\nabla\phi| = 0$$

The construction of the gradient magnitude term requires upwinding. The formula used here is given by:

$$u_P|\nabla\phi|_P = \max(u_P, 0) \left[\sum_A \min\left(\frac{\phi_A - \phi_P}{d_{AP}}, 0\right)^2 \right]^{1/2} + \min(u_P, 0) \left[\sum_A \max\left(\frac{\phi_A - \phi_P}{d_{AP}}, 0\right)^2 \right]^{1/2}$$

Here are some simple test cases for this problem:

```
>>> from fipy.meshes import Grid1D
>>> from fipy.solvers import *
>>> SparseMatrix = LinearLUSolver()._matrixClass
>>> mesh = Grid1D(dx = 1., nx = 3)
>>> from fipy.variables.cellVariable import CellVariable
```

Trivial test:

```
>>> var = CellVariable(value = numerix.zeros(3, 'd'), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(0)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, numerix.zeros(3, 'd'), atol = 1e-10))
True
```

Less trivial test:

```
>>> var = CellVariable(value = numerix.arange(3), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(1)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((0., -1., -1.)), atol = 1e-10))
True
```

Even less trivial

```
>>> var = CellVariable(value = numerix.arange(3), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(-1.)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((1., 1., 0.)), atol = 1e-10))
True
```

Another trivial test case (more trivial than a trivial test case standing on a harpsichord singing “trivial test cases are here again”)

```
>>> vel = numerix.array((-1, 2, -3))
>>> var = CellVariable(value = numerix.array((4, 6, 1)), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(vel)._buildMatrix(var, SparseMatrix)
>>> print(numerix.allclose(b, -vel * numerix.array((2, numerix.sqrt(5**2 + 2**2), 5), atol = 1e-10))
True
```

Somewhat less trivial test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2)
>>> vel = numerix.array((3, -5, -6, -3))
>>> var = CellVariable(value = numerix.array((3, 1, 6, 7)), mesh = mesh)
>>> v, L, b = FirstOrderAdvectionTerm(vel)._buildMatrix(var, SparseMatrix)
>>> answer = -vel * numerix.array((2, numerix.sqrt(2**2 + 6**2), 1, 0))
>>> print(numerix.allclose(b, answer, atol = 1e-10))
True
```

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__init__ (*coeff=None*)

Create a *Term*.

Parameters *coeff* (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.firstOrderAdvectionTerm'

class fipy.terms.AdvectionTerm(*coeff=None*)

Bases: *fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm*

The *AdvectionTerm* object constructs the *b* vector contribution for the advection term given by

$$u|\nabla\phi|$$

from the advection equation given by:

$$\frac{\partial\phi}{\partial t} + u|\nabla\phi| = 0$$

The construction of the gradient magnitude term requires upwinding as in the standard *FirstOrderAdvectionTerm*. The higher order terms are incorporated as follows. The formula used here is given by:

$$u_P|\nabla\phi|_P = \max(u_P, 0) \left[\sum_A \min(D_{AP}, 0)^2 \right]^{1/2} + \min(u_P, 0) \left[\sum_A \max(D_{AP}, 0)^2 \right]^{1/2}$$

where,

$$D_{AP} = \frac{\phi_A - \phi_P}{d_{AP}} - \frac{d_{AP}}{2} m(L_A, L_P)$$

and

$$\begin{aligned} m(x, y) &= x && \text{if } |x| \leq |y| \forall xy \geq 0 \\ m(x, y) &= y && \text{if } |x| > |y| \forall xy \geq 0 \\ m(x, y) &= 0 && \text{if } xy < 0 \end{aligned}$$

also,

$$\begin{aligned} L_A &= \frac{\phi_{AA} + \phi_P - 2\phi_A}{d_{AP}^2} \\ L_P &= \frac{\phi_A + \phi_{PP} - 2\phi_P}{d_{AP}^2} \end{aligned}$$

Here are some simple test cases for this problem:

```
>>> from fipy.meshes import Grid1D
>>> from fipy.solvers import *
>>> SparseMatrix = LinearPCGSolver()._matrixClass
>>> mesh = Grid1D(dx = 1., nx = 3)
```

Trivial test:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> coeff = CellVariable(mesh = mesh, value = numerix.zeros(3, 'd'))
>>> v, L, b = AdvectionTerm(0.)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, numerix.zeros(3, 'd'), atol = 1e-10))
True
```

Less trivial test:

```
>>> coeff = CellVariable(mesh = mesh, value = numerix.arange(3))
>>> v, L, b = AdvectionTerm(1.)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((0., -1., -1.)), atol = 1e-10))
True
```

Even less trivial

```
>>> coeff = CellVariable(mesh = mesh, value = numerix.arange(3))
>>> v, L, b = AdvectionTerm(-1.)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, numerix.array((1., 1., 0.)), atol = 1e-10))
True
```

Another trivial test case (more trivial than a trivial test case standing on a harpsichord singing “trivial test cases are here again”)

```
>>> vel = numerix.array((-1, 2, -3))
>>> coeff = CellVariable(mesh = mesh, value = numerix.array((4, 6, 1)))
>>> v, L, b = AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(b, -vel * numerix.array((2, numerix.sqrt(5**2 + 2**2), 5))), atol = 1e-10))
True
```

Somewhat less trivial test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2)
>>> vel = numerix.array((3, -5, -6, -3))
```

(continues on next page)

(continued from previous page)

```
>>> coeff = CellVariable(mesh = mesh, value = numerix.array((3, 1, 6, 7)))
>>> v, L, b = AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
>>> answer = -vel * numerix.array((2, numerix.sqrt(2**2 + 6**2), 1, 0))
>>> print(numerix.allclose(b, answer, atol = 1e-10))
True
```

For the above test cases the *AdvectionTerm* gives the same result as the *AdvectionTerm*. The following test imposes a quadratic field. The higher order term can resolve this field correctly.

$$\phi = x^2$$

The returned vector b should have the value:

$$-|\nabla\phi| = -\left|\frac{\partial\phi}{\partial x}\right| = -2|x|$$

Build the test case in the following way,

```
>>> mesh = Grid1D(dx = 1., nx = 5)
>>> vel = 1.
>>> coeff = CellVariable(mesh = mesh, value = mesh.cellCenters[0]**2)
>>> v, L, b = __AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
```

The first order term is not accurate. The first and last element are ignored because they don't have any neighbors for higher order evaluation

```
>>> print(numerix.allclose(CellVariable(mesh=mesh,
... value=b).globalValue[1:-1], -2 * mesh.cellCenters.globalValue[0][1:-1]))
False
```

The higher order term is spot on.

```
>>> v, L, b = AdvectionTerm(vel)._buildMatrix(coeff, SparseMatrix)
>>> print(numerix.allclose(CellVariable(mesh=mesh,
... value=b).globalValue[1:-1], -2 * mesh.cellCenters.globalValue[0][1:-1]))
True
```

The *AdvectionTerm* will also resolve a circular field with more accuracy,

$$\phi = (x^2 + y^2)^{1/2}$$

Build the test case in the following way,

```
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 10, ny = 10)
>>> vel = 1.
>>> x, y = mesh.cellCenters
>>> r = numerix.sqrt(x**2 + y**2)
>>> coeff = CellVariable(mesh = mesh, value = r)
>>> v, L, b = __AdvectionTerm(1.)._buildMatrix(coeff, SparseMatrix)
>>> error = CellVariable(mesh=mesh, value=b + 1)
>>> ans = CellVariable(mesh=mesh, value=b + 1)
>>> ans[(x > 2) & (x < 8) & (y > 2) & (y < 8)] = 0.123105625618
>>> print((error <= ans).all())
True
```

The maximum error is large (about 12 %) for the first order advection.

```
>>> v, L, b = AdvectionTerm(1.)._buildMatrix(coeff, SparseMatrix)
>>> error = CellVariable(mesh=mesh, value=b + 1)
>>> ans = CellVariable(mesh=mesh, value=b + 1)
>>> ans[(x > 2) & (x < 8) & (y > 2) & (y < 8)] = 0.0201715476598
>>> print((error <= ans).all())
True
```

The maximum error is 2 % when using a higher order contribution.

Create a *Term*.

Parameters **coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

__module__ = 'fipy.terms.advectionTerm'

Chapter 34

fipy.tests package

34.1 Submodules

34.2 fipy.tests.doctestPlus module

`fipy.tests.doctestPlus.execButNoTest (name='__main__')`

`fipy.tests.doctestPlus.register_skipper (flag, test, why, skipWarning=True)`

Create a new doctest option flag for skipping tests

Parameters

- **flag** (*str*) – Name of the option flag
- **test** (*function*) – A function which should return *True* if the test should be run
- **why** (*str*) – Explanation for why the test was skipped (to be used in a string “Skipped %(count)d doctest examples because %(why)s”)
- **skipWarning** (*bool*) – Whether or not to report on tests skipped by this flag (default *True*)

`fipy.tests.doctestPlus.report_skips ()`

Print out how many doctest examples were skipped due to flags

`fipy.tests.doctestPlus.testmod (m=None, name=None, globs=None, verbose=None, report=True, optionflags=0, extraglobs=None, raise_on_error=False, exclude_empty=False)`

Test examples in the given module. Return (#failures, #tests).

Largely duplicated from `doctest.testmod()`, but using `_SelectiveDocTestParser`.

Test examples in docstrings in functions and classes reachable from module *m* (or the current module if *m* is not supplied), starting with *m.__doc__*.

Also test examples reachable from dict *m.__test__* if it exists and is not *None*. *m.__test__* maps names to functions, classes and strings; function and class docstrings are tested even if the name is private; strings are tested directly, as if they were docstrings.

Return (#failures, #tests).

See `help(doctest)` for an overview.

Optional keyword arg *name* gives the name of the module; by default use *m.__name__*.

Optional keyword arg *globs* gives a dict to be used as the globals when executing examples; by default, use *m.__dict__*. A copy of this dict is actually used for each docstring, so that each docstring's examples start with a clean slate.

Optional keyword arg *extraglobs* gives a dictionary that should be merged into the globals that are used to execute examples. By default, no extra globals are used. This is new in 2.4.

Optional keyword arg *verbose* prints lots of stuff if true, prints only failures if false; by default, it's true iff *-v* is in *sys.argv*.

Optional keyword arg *report* prints a summary at the end when true, else prints nothing at the end. In verbose mode, the summary is detailed, else very brief (in fact, empty if all tests passed).

Optional keyword arg *optionflags* or's together module constants, and defaults to 0. This is new in 2.3. Possible values (see the docs for details):

```
DONT_ACCEPT_TRUE_FOR_1
DONT_ACCEPT_BLANKLINE
NORMALIZE_WHITESPACE
ELLIPSIS
SKIP
IGNORE_EXCEPTION_DETAIL
REPORT_UDIFF
REPORT_CDIFF
REPORT_NDIFF
REPORT_ONLY_FIRST_FAILURE
```

as well as FiPy's flags:

```
GMSH
SCIPY
TVTK
SERIAL
PARALLEL
PROCESSOR_0
PROCESSOR_0_OF_2
PROCESSOR_1_OF_2
PROCESSOR_0_OF_3
PROCESSOR_1_OF_3
PROCESSOR_2_OF_3
```

Optional keyword arg "raise_on_error" raises an exception on the first unexpected exception or failure. This allows failures to be postmortem debugged.

34.3 fipy.tests.lateImportTest module

34.4 fipy.tests.test module

```
class fipy.tests.test.test(dist, **kw)
```

```
    Bases: setuptools.command.test.test
```

```
    Construct the command for dist, updating vars(self) with any keyword parameters.
```

```
    __module__ = 'fipy.tests.test'
```

```
    description = 'run unit tests after in-place build (deprecated), for FiPy and its exam
```

finalize_options()

Set final values for all the options that this command supports. This is always called as late as possible, ie. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if 'foo' depends on 'bar', then it is safe to set 'foo' from 'bar' as long as 'foo' still has the same value it was assigned in 'initialize_options()'.

This method must be implemented by all command classes.

initialize_options()

Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, 'initialize_options()' implementations are just a bunch of "self.foo = None" assignments.

This method must be implemented by all command classes.

printPackageInfo()**run_tests()**

```
user_options = [('test-module=', 'm', "Run 'test_suite' in specified module"), ('test-
```

34.5 fipy.tests.testProgram module

34.6 Module contents

unit testing scripts no chapter heading

fipy.tools package

35.1 Subpackages

35.1.1 fipy.tools.comms package

Submodules

fipy.tools.comms.abstractCommWrapper module

```
class fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper
```

Bases: `object`

MPI Communicator wrapper

Encapsulates capabilities needed for possibly parallel operations. Some capabilities are not parallel.

Barrier ()

MaxAll (*vec*)

MinAll (*vec*)

Norm2 (*vec*)

property `Nproc`

`__dict__` = `mappingproxy({'__module__': 'fipy.tools.comms.abstractCommWrapper', '__doc__`

`__getstate__` ()

`__init__` ()

Initialize self. See help(type(self)) for accurate signature.

`__module__` = 'fipy.tools.comms.abstractCommWrapper'

`__repr__` ()

Return repr(self).

`__setstate__` (*dict*)

`__weakref__`

list of weak references to the object (if defined)

```
all (a, axis=None)
allclose (a, b, rtol=1e-05, atol=1e-08)
allequal (a, b)
allgather (obj)
any (a, axis=None)
bcast (obj, root=0)
property procID
sum (a, axis=None)
```

fiPy.tools.comms.dummyComm module

```
class fipy.tools.comms.dummyComm.DummyComm
    Bases: fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper

    property Nproc
    __module__ = 'fipy.tools.comms.dummyComm'
    property procID
```

Module contents

35.1.2 fipy.tools.dimensions package

Submodules

fiPy.tools.dimensions.DictWithDefault module

fiPy.tools.dimensions.NumberDict module

fiPy.tools.dimensions.physicalField module

Physical quantities with units.

This module derives from Konrad Hinsén's `PhysicalQuantity` <<http://dirac.cnrs-orleans.fr/ScientificPython/>>.

This module provides a data type that represents a physical quantity together with its unit. It is possible to add and subtract these quantities if the units are compatible, and a quantity can be converted to another compatible unit. Multiplication, subtraction, and raising to integer powers is allowed without restriction, and the result will have the correct unit. A quantity can be raised to a non-integer power only if the result can be represented by integer powers of the base units.

The values of physical constants are taken from the 2002 recommended values from CODATA. Other conversion factors (e.g. for British units) come from Appendix B of NIST Special Publication 811.

Warning: We can't guarantee for the correctness of all entries in the unit table, so use this at your own risk!

Base SI units:

```
m, kg, s, A, K, mol, cd, rad, sr
```

SI prefixes:

```
Y = 1e+24
Z = 1e+21
E = 1e+18
P = 1e+15
T = 1e+12
G = 1e+09
M = 1e+06
k = 1000
h = 100
da = 10
d = 0.1
c = 0.01
m = 0.001
mu = 1e-06
n = 1e-09
p = 1e-12
f = 1e-15
a = 1e-18
z = 1e-21
y = 1e-24
```

Units derived from SI (accepting SI prefixes):

```
1 Bq = 1 1/s
1 C = 1 s*A
1 degC = 1 K
1 F = 1 s**4*A**2/m**2/kg
1 Gy = 1 m**2/s**2
1 H = 1 m**2*kg/s**2/A**2
1 Hz = 1 1/s
1 J = 1 m**2*kg/s**2
1 lm = 1 cd*sr
1 lx = 1 cd*sr/m**2
1 N = 1 m*kg/s**2
1 ohm = 1 m**2*kg/s**3/A**2
1 Pa = 1 kg/m/s**2
1 S = 1 s**3*A**2/m**2/kg
1 Sv = 1 m**2/s**2
1 T = 1 kg/s**2/A
1 V = 1 m**2*kg/s**3/A
1 W = 1 m**2*kg/s**3
1 Wb = 1 m**2*kg/s**2/A
```

Other units that accept SI prefixes:

```
1 eV = 1.60217653e-19 m**2*kg/s**2
```

Additional units and constants:

```
1 acres = 4046.8564224 m**2
1 amu = 1.6605402e-27 kg
1 Ang = 1e-10 m
1 atm = 101325.0 kg/m/s**2
```

(continues on next page)

(continued from previous page)

```

1 b = 1e-28 m
1 bar = 100000.0 kg/m/s**2
1 Bohr = 5.291772081145378e-11 m
1 Btui = 1055.05585262 m**2*kg/s**2
1 c = 299792458.0 m/s
1 cal = 4.184 m**2*kg/s**2
1 cali = 4.1868 m**2*kg/s**2
1 cl = 1.0000000000000003e-05 m**3
1 cup = 0.00023658825600000004 m**3
1 d = 86400.0 s
1 deg = 0.017453292519943295 rad
1 degF = 0.5555555555555556 K
1 degR = 0.5555555555555556 K
1 dl = 0.00010000000000000003 m**3
1 dyn = 1e-05 m*kg/s**2
1 e = 1.60217653e-19 s*A
1 eps0 = 8.85418781762039e-12 s**4*A**2/m**3/kg
1 erg = 1e-07 m**2*kg/s**2
1 floz = 2.9573532000000005e-05 m**3
1 ft = 0.3048 m
1 g = 0.001 kg
1 galUK = 0.0045460900000000002 m**3
1 galUS = 0.0037854120960000006 m**3
1 gn = 9.80665 m/s**2
1 Grav = 6.6742e-11 m**3/kg/s**2
1 h = 3600.0 s
1 ha = 10000.0 m**2
1 Hartree = 4.35974417680088e-18 m**2*kg/s**2
1 hbar = 1.0545716823644548e-34 m**2*kg/s
1 hpEl = 746.0 m**2*kg/s**3
1 hplanck = 6.6260693e-34 m**2*kg/s
1 hpUK = 745.7 m**2*kg/s**3
1 inch = 0.025400000000000002 m
1 invcm = 1.9864456023253395e-23 m**2*kg/s**2
1 kB = 1.3806505e-23 m**2*kg/s**2/K
1 kcal = 4184.0 m**2*kg/s**2
1 kcali = 4186.8 m**2*kg/s**2
1 Ken = 1.3806505e-23 m**2*kg/s**2
1 l = 0.0010000000000000002 m**3
1 lb = 0.45359237 kg
1 lyr = 9460730472580800.0 m
1 me = 9.1093826e-31 kg
1 mi = 1609.344 m
1 min = 60.0 s
1 ml = 1.0000000000000002e-06 m**3
1 mp = 1.67262171e-27 kg
1 mu0 = 1.2566370614359173e-06 m*kg/s**2/A**2
1 Nav = 6.0221415e+23 1/mol
1 nmi = 1852.0 m
1 oz = 0.028349523125 kg
1 psi = 6894.75729316836 kg/m/s**2
1 pt = 0.0004731765120000001 m**3
1 qt = 0.0009463530240000002 m**3
1 tbsps = 1.4786766000000002e-05 m**3
1 ton = 907.18474 kg
1 Torr = 133.32236842105263 kg/m/s**2
1 tsp = 4.9289220000000005e-06 m**3

```

(continues on next page)

(continued from previous page)

```

1 wk = 604800.0 s
1 yd = 0.9144000000000001 m
1 yr = 31536000.0 s
1 yrJul = 31557600.0 s
1 yrSid = 31558152.959999997 s

```

class fipy.tools.dimensions.physicalField.**PhysicalField**(*value*, *unit=None*, *array=None*)

Bases: `object`

Physical field or quantity with units

Physical Fields can be constructed in one of two ways:

- *PhysicalField(*value*, *unit*)*, where **value** is a number of arbitrary type and **unit** is a string containing the unit name

```

>>> print(PhysicalField(value = 10., unit = 'm'))
10.0 m

```

- *PhysicalField(*string*)*, where **string** contains both the value and the unit. This form is provided to make interactive use more convenient

```

>>> print(PhysicalField(value = "10. m"))
10.0 m

```

Dimensionless quantities, with a *unit* of 1, can be specified in several ways

```

>>> print(PhysicalField(value = "1"))
1.0 1
>>> print(PhysicalField(value = 2., unit = " "))
2.0 1
>>> print(PhysicalField(value = 2.))
2.0 1

```

Physical arrays are also possible (and are the reason this code was adapted from Konrad Hinsén's original `PhysicalQuantity`). The *value* can be a `Numeric array`:

```

>>> a = numerix.array(((3., 4.), (5., 6.)))
>>> print(PhysicalField(value = a, unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m

```

or a *tuple*:

```

>>> print(PhysicalField(value = ((3., 4.), (5., 6.)), unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m

```

or as a single value to be applied to every element of a supplied array:

```

>>> print(PhysicalField(value = 2., unit = "m", array = a))
[[ 2.  2.]
 [ 2.  2.]] m

```

Every element in an array has the same unit, which is stored only once for the whole array.

__abs__()Return the absolute value of the quantity. The *unit* is unchanged.

```
>>> print(abs(PhysicalField(((3., -2.), (-1., 4.)), 'm')))
[[ 3.  2.]
 [ 1.  4.]] m
```

__add__(other)

Add two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'm'))
10.01 km
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

__array__(t=None)Return a dimensionless *PhysicalField* as a *Numeric* array.

```
>>> print(NumPyArray(PhysicalField(((2., 3.), (4., 5.)), "m/m")))
[[ 2.  3.]
 [ 4.  5.]]
```

As a special case, fields with angular units are converted to base units (radians) and then assumed dimensionless.

```
>>> print(NumPyArray(PhysicalField(((2., 3.), (4., 5.)), "deg")))
[[ 0.03490659  0.05235988]
 [ 0.06981317  0.08726646]]
```

If the array is not dimensionless, the numerical value in its base units is returned.

```
>>> NumPyArray(PhysicalField(((2., 3.), (4., 5.)), "mm"))
array([[ 0.002,  0.003],
       [ 0.004,  0.005]])
```

__array_priority__ = 100.0**__array_wrap__(arr, context=None)**

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples of ufuncs.

```
>>> from fipy.tools.dimensions.physicalField import PhysicalField
>>> print(type(NumPyArray([1.0, 2.0]) * PhysicalField([1.0, 2.0], unit="m
↳")))
<class 'fipy.tools.dimensions.physicalField.PhysicalField'>
```

```
>>> print(type(NumPyArray([1.0, 2.0]) * PhysicalField([1.0, 2.0])))
<class 'fipy.tools.dimensions.physicalField.PhysicalField'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(isinstance(Gamma(PhysicalField([1.0, 2.0])), type(NumPyArray(1))))
1
```

`__bool__()`

Test if the quantity is zero.

Should this only pass if the unit offset is zero?

`__dict__ = mappingproxy({'__module__': 'fipy.tools.dimensions.physicalField', '__doc__': None, '__weakref__': None, '__dict__': None, '__delattr__': None, '__dir__': None, '__eq__': None, '__float__': None, '__ge__': None, '__getattribute__': None, '__getitem__': None, '__gt__': None, '__hash__': None, '__init__': None, '__init_subclass__': None, '__le__': None, '__len__': None, '__lt__': None, '__ne__': None, '__new__': None, '__reduce__': None, '__reduce_ex__': None, '__repr__': None, '__setattr__': None, '__sizeof__': None, '__str__': None, '__subclasshook__': None, '__weakref__': None})`

`__div__ (other)`

Divide two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print(PhysicalField(10., 'm') / PhysicalField(2., 's'))
5.0 m/s
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *1*. This facilitates passing physical quantities to packages such as [Numeric](#) that cannot use units, while ensuring the quantities have the desired units

```
>>> print((PhysicalField(1., 'inch')
...       / PhysicalField(1., 'mm')))
25.4
```

`__eq__ (other)`

Return self==value.

`__float__()`

Return a dimensionless *PhysicalField* quantity as a float.

```
>>> float(PhysicalField("2. m/m"))
2.0
```

As a special case, quantities with angular units are converted to base units (radians) and then assumed dimensionless.

```
>>> print(numerix.round_(float(PhysicalField("2. deg")), 6))
0.034907
```

If the quantity is not dimensionless, the conversion fails.

```
>>> float(PhysicalField("2. m"))
Traceback (most recent call last):
...
TypeError: Not possible to convert a PhysicalField with dimensions to float
```

Just as a [Numeric](#) array cannot be cast to float, neither can *PhysicalField* arrays

```
>>> float(PhysicalField(((2., 3.), (4., 5.)), "m/m"))
Traceback (most recent call last):
...
TypeError: only ...-1 arrays can be converted to Python scalars
```

`__ge__ (other)`

Return self>=value.

`__getitem__ (index)`

Return the specified element of the array. The unit of the result will be the unit of the array.

```
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> print(a[1, 1])
6.0 m
```

__gt__ (*other*)

Compare *self* to *other*, returning an array of Boolean values corresponding to the test against each element.

```
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> print(numerix.allclose(a > PhysicalField("13 ft"),
...                        [[False, True], [ True, True]]))
True
```

Appropriately formatted dimensional quantity strings can also be compared.

```
>>> print(numerix.allclose(a > "13 ft",
...                        [[False, True], [ True, True]]))
True
```

Arrays are compared element to element

```
>>> print(numerix.allclose(a > PhysicalField(((3., 13.), (17., 6.)), "ft"),
...                        [[ True, True], [False, True]]))
True
```

Units must be compatible

```
>>> print(a > PhysicalField("1 lb"))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

And so must array dimensions

```
>>> print(a > PhysicalField(((3., 13., 4.), (17., 6., 2.)), "ft"))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

__hash__ ()

Return hash(self).

__init__ (*value*, *unit=None*, *array=None*)

Physical Fields can be constructed in one of two ways:

- *PhysicalField(*value**, **unit**), where **value** is a number of arbitrary type and **unit** is a string containing the unit name

```
>>> print(PhysicalField(value = 10., unit = 'm'))
10.0 m
```

- *PhysicalField(*string*)*, where **string** contains both the value and the unit. This form is provided to make interactive use more convenient

```
>>> print(PhysicalField(value = "10. m"))
10.0 m
```

Dimensionless quantities, with a *unit* of 1, can be specified in several ways


```
>>> print(PhysicalField(value = "1"))
1.0 1
>>> print(PhysicalField(value = 2., unit = " "))
2.0 1
>>> print(PhysicalField(value = 2.))
2.0 1
```

Physical arrays are also possible (and are the reason this code was adapted from Konrad Hinsén's original `PhysicalQuantity`). The *value* can be a `Numeric array`:

```
>>> a = numerix.array(((3., 4.), (5., 6.)))
>>> print(PhysicalField(value = a, unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or a *tuple*:

```
>>> print(PhysicalField(value = ((3., 4.), (5., 6.)), unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or as a single value to be applied to every element of a supplied array:

```
>>> print(PhysicalField(value = 2., unit = "m", array = a))
[[ 2.  2.]
 [ 2.  2.]] m
```

Every element in an array has the same unit, which is stored only once for the whole array.

__le__ (*other*)

Return self<=value.

__len__ ()

__lt__ (*other*)

Return self<value.

__mod__ (*other*)

Return the remainder of dividing two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print(PhysicalField(11., 'm') % PhysicalField(2., 's'))
1.0 m/s
```

__module__ = 'fipy.tools.dimensions.physicalField'

__mul__ (*other*)

Multiply two physical quantities. The unit of the result is the product of the units of the operands.

```
>>> print(PhysicalField(10., 'N') * PhysicalField(10., 'm') ==
↳ PhysicalField(100., 'N*m'))
True
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *1*. This facilitates passing physical quantities to packages such as Numeric that cannot use units, while ensuring the quantities have the desired units.

```
>>> print((PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
```

`__ne__(other)`

Return self!=value.

`__neg__()`

Return the negative of the quantity. The *unit* is unchanged.

```
>>> print(-PhysicalField(((3., -2.), (-1., 4.)), 'm'))
[[-3.  2.]
 [ 1. -4.]] m
```

`__nonzero__()`

Test if the quantity is zero.

Should this only pass if the unit offset is zero?

`__pos__()`

`__pow__(other)`

Raise a *PhysicalField* to a power. The unit is raised to the same power.

```
>>> print(PhysicalField(10., 'm')**2)
100.0 m**2
```

`__radd__(other)`

Add two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'm'))
10.01 km
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

`__rdiv__(other)`

`__repr__()`

Return representation of a physical quantity suitable for re-use

```
>>> PhysicalField(value = 3., unit = "eV")
PhysicalField(3.0, 'eV')
```

`__rmul__(other)`

Multiply two physical quantities. The unit of the result is the product of the units of the operands.

```
>>> print(PhysicalField(10., 'N') * PhysicalField(10., 'm') ==
↪PhysicalField(100., 'N*m'))
True
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *I*. This facilitates passing physical quantities to packages such as Numeric that cannot use units, while ensuring the quantities have the desired units.

```
>>> print((PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
```

`__rpow__(other)`

`__rsub__(other)`

__rtruediv__ (*other*)

__setitem__ (*index, value*)

Assign the specified element of the array, performing appropriate conversions.

```
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> a[0, 1] = PhysicalField("6 ft")
>>> print(a)
[[ 3.      1.8288]
 [ 5.      6.      ]] m
>>> a[1, 0] = PhysicalField("2 min")
Traceback (most recent call last):
...
TypeError: Incompatible units
```

__str__ ()

Return human-readable form of a physical quantity

```
>>> print(PhysicalField(value = 3., unit = "eV"))
3.0 eV
```

__sub__ (*other*)

Subtract two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') - PhysicalField(10., 'm'))
9.99 km
>>> print(PhysicalField(10., 'km') - PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

__truediv__ (*other*)

Divide two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print(PhysicalField(10., 'm') / PhysicalField(2., 's'))
5.0 m/s
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *I*. This facilitates passing physical quantities to packages such as [Numeric](#) that cannot use units, while ensuring the quantities have the desired units

```
>>> print((PhysicalField(1., 'inch')
...       / PhysicalField(1., 'mm'))
25.4
```

__weakref__

list of weak references to the object (if defined)

add (*other*)

Add two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'm'))
10.01 km
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'J'))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: Incompatible units
```

allclose (*other*, *atol*=None, *rtol*=1e-08)

This function tests whether or not *self* and *other* are equal subject to the given relative and absolute tolerances. The formula used is:

```
| self - other | < atol + rtol * | other |
```

This means essentially that both elements are small compared to *atol* or their difference divided by *other*'s value is small compared to *rtol*.

allequal (*other*)

This function tests whether or not *self* and *other* are exactly equal.

arccos ()

Return the inverse cosine of the *PhysicalField* in radians

```
>>> print(PhysicalField(0).arccos().allclose("1.57079632679 rad"))
1
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1 m").arccos(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arccosh ()

Return the inverse hyperbolic cosine of the *PhysicalField*

```
>>> print(numerix.allclose(PhysicalField(2).arccosh(),
...                        1.31695789692))
1
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1. m").arccosh(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arcsin ()

Return the inverse sine of the *PhysicalField* in radians

```
>>> print(PhysicalField(1).arcsin().allclose("1.57079632679 rad"))
1
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1 m").arcsin(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arctan()

Return the arctangent of the *PhysicalField* in radians

```
>>> print(numerix.round_(PhysicalField(1).arctan(), 6))
0.785398
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1 m").arctan(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arctan2(*other*)

Return the arctangent of *self* divided by *other* in radians

```
>>> print(numerix.round_(PhysicalField(2.).arctan2(PhysicalField(5.)), 6))
0.380506
```

The input *PhysicalField* objects must be in the same dimensions

```
>>> print(numerix.round_(PhysicalField(2.54, "cm").arctan2(PhysicalField(1.,
↪ "inch")), 6))
0.785398
```

```
>>> print(numerix.round_(PhysicalField(2.).arctan2(PhysicalField("5. m")), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arctanh()

Return the inverse hyperbolic tangent of the *PhysicalField*

```
>>> print(PhysicalField(0.5).arctanh())
0.549306144334
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1 m").arctanh(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

ceil()

Return the smallest integer greater than or equal to the *PhysicalField*.

```
>>> print(PhysicalField(2.2, "m").ceil())
3.0 m
```

conjugate()

Return the complex conjugate of the *PhysicalField*.

```
>>> print(PhysicalField(2.2 - 3j, "ohm").conjugate() == PhysicalField(2.2 + ↵
↪ 3j, "ohm"))
True
```

convertToUnit (*unit*)

Changes the unit to *unit* and adjusts the value such that the combination is equivalent. The new unit is by a string containing its name. The new unit must be compatible with the previous unit of the object.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> e.convertToUnit('kcal/mol')
>>> print(e)
1694.27557621 kcal/mol
```

copy ()

Make a duplicate.

```
>>> a = PhysicalField(1, unit = 'inch')
>>> b = a.copy()
```

The duplicate will not reflect changes made to the original

```
>>> a.convertToUnit('cm')
>>> print(a)
2.54 cm
>>> print(b)
1 inch
```

Likewise for arrays

```
>>> a = PhysicalField(numerix.array((0, 1, 2)), unit = 'm')
>>> b = a.copy()
>>> a[0] = 3
>>> print(a)
[3 1 2] m
>>> print(b)
[0 1 2] m
```

cos ()

Return the cosine of the *PhysicalField*

```
>>> print(numerix.round_(PhysicalField(2*numerix.pi/6, "rad").cos(), 6))
0.5
>>> print(numerix.round_(PhysicalField(60., "deg").cos(), 6))
0.5
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(60., "m").cos()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

cosh ()

Return the hyperbolic cosine of the *PhysicalField*

```
>>> PhysicalField(0.).cosh()
1.0
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60., "m").cosh()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

divide (other)

Divide two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print(PhysicalField(10., 'm') / PhysicalField(2., 's'))
5.0 m/s
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *1*. This facilitates passing physical quantities to packages such as [Numeric](#) that cannot use units, while ensuring the quantities have the desired units

```
>>> print((PhysicalField(1., 'inch')
...       / PhysicalField(1., 'mm'))
25.4
```

dot (other)

Return the dot product of *self* with *other*. The resulting unit is the product of the units of *self* and *other*.

```
>>> v = PhysicalField(((5., 6.), (7., 8.)), "m")
>>> print(PhysicalField(((1., 2.), (3., 4.)), "m").dot(v))
[ 26.  44.] m**2
```

floor ()

Return the largest integer less than or equal to the *PhysicalField*.

```
>>> print(PhysicalField(2.2, "m").floor())
2.0 m
```

getsctype (default=None)

Returns the NumPy sctype of the underlying array.

```
>>> PhysicalField(1, 'm').getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↪array(1))
True
>>> PhysicalField(1., 'm').getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↪array(1.))
True
>>> PhysicalField((1, 1.), 'm').getsctype() == numerix.NUMERIX.
↪obj2sctype(numerix.array((1., 1.)))
True
```

inBaseUnits ()

Return the quantity with all units reduced to their base SI elements.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

inDimensionless ()

Returns the numerical value of a dimensionless quantity.

```
>>> print(PhysicalField(((2., 3.), (4., 5.))).inDimensionless())
[[ 2.  3.]
 [ 4.  5.]]
```

It's an error to convert a quantity with units

```
>>> print(PhysicalField(((2., 3.), (4., 5.))), "m").inDimensionless()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

inRadians()

Converts an angular quantity to radians and returns the numerical value.

```
>>> print(PhysicalField(((2., 3.), (4., 5.))), "rad").inRadians()
[[ 2.  3.]
 [ 4.  5.]]
>>> print(PhysicalField(((2., 3.), (4., 5.))), "deg").inRadians()
[[ 0.03490659  0.05235988]
 [ 0.06981317  0.08726646]]
```

As a special case, assumes a dimensionless quantity is already in radians.

```
>>> print(PhysicalField(((2., 3.), (4., 5.))).inRadians()
[[ 2.  3.]
 [ 4.  5.]]
```

It's an error to convert a quantity with non-angular units

```
>>> print(PhysicalField(((2., 3.), (4., 5.))), "m").inRadians()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

inSIUnits()

Return the quantity with all units reduced to SI-compatible elements.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print(e.inSIUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

inUnitsOf(*units)

Returns one or more *PhysicalField* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *PhysicalField*.

```
>>> freeze = PhysicalField('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *PhysicalField* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.


```

>>> t = PhysicalField(314159., 's')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d',
↳ 'h', 'min', 's')),
...                                     ['3.0 d', '15.0 h
↳ ', '15.0 min', '59.0 s']]),
...                                     True))
1

```

isCompatible (*unit*)

itemset (*value*)

Assign the value of a scalar array, performing appropriate conversions.

```

>>> a = PhysicalField(4., "m")
>>> a.itemset(PhysicalField("6 ft"))
>>> print(a.allclose("1.8288 m"))
1
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> try:
...     a.itemset(PhysicalField("6 ft"))
... except IndexError:
...     # NumPy 1.7 has changed the exception type
...     raise ValueError("can only place a scalar for an array of size 1")
Traceback (most recent call last):
...
ValueError: can only convert an array of size 1 to a Python scalar
>>> a.itemset(PhysicalField("2 min"))
Traceback (most recent call last):
...
TypeError: Incompatible units

```

property **itemsize**

log ()

Return the natural logarithm of the *PhysicalField*

```

>>> print(numerix.round_(PhysicalField(10).log(), 6))
2.302585

```

The input *PhysicalField* must be dimensionless

```

>>> print(numerix.round_(PhysicalField("1. m").log(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units

```

log10 ()

Return the base-10 logarithm of the *PhysicalField*

```

>>> print(numerix.round_(PhysicalField(10.).log10(), 6))
1.0

```

The input *PhysicalField* must be dimensionless

```

>>> print(numerix.round_(PhysicalField("1. m").log10(), 6))
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...
TypeError: Incompatible units
```

multiply (*other*)

Multiply two physical quantities. The unit of the result is the product of the units of the operands.

```
>>> print(PhysicalField(10., 'N') * PhysicalField(10., 'm') ==
↳ PhysicalField(100., 'N*m'))
True
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *I*. This facilitates passing physical quantities to packages such as Numeric that cannot use units, while ensuring the quantities have the desired units.

```
>>> print((PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
```

property numericValue

Return the *PhysicalField* without units, after conversion to base SI units.

```
>>> print(meritix.round_(PhysicalField("1 inch").numericValue, 6))
0.0254
```

put (*indices*, *values*)

put is the opposite of *take*. The values of *self* at the locations specified in *indices* are set to the corresponding value of *values*.

The *indices* can be any integer sequence object with values suitable for indexing into the flat form of *self*. The *values* must be any sequence of values that can be converted to the typecode of *self*.

```
>>> f = PhysicalField((1., 2., 3.), "m")
>>> f.put((2, 0), PhysicalField((2., 3.), "inch"))
>>> print(f)
[ 0.0762  2.          0.0508] m
```

The units of *values* must be compatible with *self*.

```
>>> f.put(1, PhysicalField(3, "kg"))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

ravel ()**reshape** (*shape*)

Changes the shape of *self* to that specified in *shape*

```
>>> print(PhysicalField((1., 2., 3., 4.), "m").reshape((2, 2)))
[[ 1.  2.]
 [ 3.  4.]] m
```

The new shape must have the same size as the existing one.

```
>>> print(PhysicalField((1., 2., 3., 4.), "m").reshape((2, 3)))
Traceback (most recent call last):
...
ValueError: total size of new array must be unchanged
```

property shape

Tuple of array dimensions.

sign()

Return the sign of the quantity. The *unit* is unchanged.

```
>>> from fipy.tools.numerix import sign
>>> print(sign(PhysicalField(((3., -2.), (-1., 4.)), 'm')))
[[ 1. -1.]
 [-1.  1.]
```

sin()

Return the sine of the *PhysicalField*

```
>>> print(PhysicalField(numerix.pi/6, "rad").sin())
0.5
>>> print(PhysicalField(30., "deg").sin())
0.5
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(30., "m").sin()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

sinh()

Return the hyperbolic sine of the *PhysicalField*

```
>>> PhysicalField(0.).sinh()
0.0
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60., "m").sinh()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

sqrt()

Return the square root of the *PhysicalField*

```
>>> print(PhysicalField("100. m**2").sqrt())
10.0 m
```

The resulting unit must be integral

```
>>> print(PhysicalField("100. m").sqrt())
Traceback (most recent call last):
...
TypeError: Illegal exponent
```

subtract (other)

Subtract two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') - PhysicalField(10., 'm'))
9.99 km
>>> print(PhysicalField(10., 'km') - PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

sum (*index=0*)

Returns the sum of all of the elements in *self* along the specified axis (first axis by default).

```
>>> print(PhysicalField(((1., 2.), (3., 4.)), "m").sum())
[ 4.  6.] m
>>> print(PhysicalField(((1., 2.), (3., 4.)), "m").sum(1))
[ 3.  7.] m
```

take (*indices, axis=0*)

Return the elements of *self* specified by the elements of *indices*. The resulting *PhysicalField* array has the same units as the original.

```
>>> print(PhysicalField((1., 2., 3.), "m").take((2, 0)))
[ 3.  1.] m
```

The optional third argument specifies the axis along which the selection occurs, and the default value (as in the example above) is 0, the first axis.

```
>>> print(PhysicalField(((1., 2., 3.), (4., 5., 6.)), "m").take((2, 0), axis_
↳ = 1))
[[ 3.  1.]
 [ 6.  4.]] m
```

tan ()

Return the tangent of the *PhysicalField*

```
>>> numerix.round_(PhysicalField(numerix.pi/4, "rad").tan(), 6)
1.0
>>> numerix.round_(PhysicalField(45, "deg").tan(), 6)
1.0
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(45., "m").tan()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

tanh ()

Return the hyperbolic tangent of the *PhysicalField*

```
>>> print(numerix.allclose(PhysicalField(1.).tanh(), 0.761594155956))
True
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60., "m").tanh()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

tostring (*max_line_width=75, precision=8, suppress_small=False, separator=' '*)

Return human-readable form of a physical quantity

```
>>> p = PhysicalField(value = (3., 3.14159), unit = "eV")
>>> print(p.tostring(precision = 3, separator = '|'))
[ 3.    | 3.142] eV
```

property unit

Return the unit object of *self*.

```
>>> PhysicalField("1 m").unit
<PhysicalUnit m>
```

class `fipy.tools.dimensions.physicalField.PhysicalUnit` (*names, factor, powers, offset=0*)

Bases: `object`

A *PhysicalUnit* represents the units of a *PhysicalField*.

This class is not generally not instantiated by users of this module, but rather it is created in the process of constructing a *PhysicalField*.

Parameters

- **names** (*str*) – Name of the unit
- **factor** (*float*) – Multiplier between the unit and the fundamental SI unit
- **powers** (*array_like of float*) – Nine elements representing the fundamental SI units of ["m", "kg", "s", "A", "K", "mol", "cd", "rad", "sr"]
- **offset** (*float*) – Displacement between the zero-point of the unit and the zero-point of the corresponding fundamental SI unit.

__dict__ = `mappingproxy({'__module__': 'fipy.tools.dimensions.physicalField', '__doc__`

`__div__` (*other*)

Divide one unit by another

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.unit / b.unit
<PhysicalUnit m/ft>
>>> a.unit / b.inBaseUnits().unit
<PhysicalUnit 1>
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.unit / d.unit
<PhysicalUnit s/Hz>
>>> c.unit / d.inBaseUnits().unit
<PhysicalUnit s**2/1>
```

or divide units by numbers

```
>>> a.unit / 3.
<PhysicalUnit m/3.0>
```

Units must have zero offset to be divided

```
>>> e = PhysicalField("1. J")
>>> f = PhysicalField("25. degC")
>>> e.unit / f.unit
Traceback (most recent call last):
...
TypeError: cannot divide units with non-zero offset
>>> e.unit / f.inBaseUnits().unit
<PhysicalUnit J/K>
```

`__eq__` (*other*)

Determine if units are identical

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.unit == b.unit
0
>>> a.unit == b.inBaseUnits().unit
1
```

Units can only be compared with other units

```
>>> a.unit == 3
Traceback (most recent call last):
...
TypeError: PhysicalUnits can only be compared with other PhysicalUnits
```

`__ge__` (*other*)

Return self>=value.

`__gt__` (*other*)

Return self>value.

`__hash__` = None

`__init__` (*names, factor, powers, offset=0*)

This class is not generally not instantiated by users of this module, but rather it is created in the process of constructing a *PhysicalField*.

Parameters

- **names** (*str*) – Name of the unit
- **factor** (*float*) – Multiplier between the unit and the fundamental SI unit
- **powers** (*array_like* of *float*) – Nine elements representing the fundamental SI units of ["m", "kg", "s", "A", "K", "mol", "cd", "rad", "sr"]
- **offset** (*float*) – Displacement between the zero-point of the unit and the zero-point of the corresponding fundamental SI unit.

`__le__` (*other*)

Return self<=value.

`__lt__` (*other*)

Return self<value.

`__module__` = 'fipy.tools.dimensions.physicalField'

`__mul__` (*other*)

Multiply units together

```

>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.unit * b.unit == _findUnit('ft*m')
True
>>> a.unit * b.inBaseUnits().unit
<PhysicalUnit m**2>
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.unit * d.unit == _findUnit('Hz*s')
True
>>> c.unit * d.inBaseUnits().unit
<PhysicalUnit 1>

```

or multiply units by numbers

```

>>> a.unit * 3.
<PhysicalUnit m*3.0>

```

Units must have zero offset to be multiplied

```

>>> e = PhysicalField("1. kB")
>>> f = PhysicalField("25. degC")
>>> e.unit * f.unit
Traceback (most recent call last):
...
TypeError: cannot multiply units with non-zero offset
>>> e.unit * f.inBaseUnits().unit
<PhysicalUnit kB*K>

```

__ne__ (*other*)

Return self!=value.

__pow__ (*other*)

Raise a unit to an integer power

```

>>> a = PhysicalField("1. m")
>>> a.unit**2
<PhysicalUnit m**2>
>>> a.unit**-2
<PhysicalUnit 1/m**2>

```

Non-integer powers are not supported

```

>>> a.unit**0.5
Traceback (most recent call last):
...
TypeError: Illegal exponent

```

Units must have zero offset to be exponentiated

```

>>> b = PhysicalField("25. degC")
>>> b.unit**2
Traceback (most recent call last):
...
TypeError: cannot exponentiate units with non-zero offset
>>> b.inBaseUnits().unit**2
<PhysicalUnit K**2>

```

__rdiv__ (*other*)

Divide something by a unit

```
>>> a = PhysicalField("1. m")
>>> 3. / a.unit
<PhysicalUnit 3.0/m>
```

Units must have zero offset to be divided

```
>>> b = PhysicalField("25. degC")
>>> 3. / b.unit
Traceback (most recent call last):
...
TypeError: cannot divide units with non-zero offset
>>> 3. / b.inBaseUnits().unit
<PhysicalUnit 3.0/K>
```

__repr__ ()

Return representation of a physical unit

```
>>> PhysicalUnit('m', 1., [1, 0, 0, 0, 0, 0, 0, 0, 0])
<PhysicalUnit m>
```

__rmul__ (*other*)

Multiply units together

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.unit * b.unit == _findUnit('ft*m')
True
>>> a.unit * b.inBaseUnits().unit
<PhysicalUnit m**2>
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.unit * d.unit == _findUnit('Hz*s')
True
>>> c.unit * d.inBaseUnits().unit
<PhysicalUnit 1>
```

or multiply units by numbers

```
>>> a.unit * 3.
<PhysicalUnit m*3.0>
```

Units must have zero offset to be multiplied

```
>>> e = PhysicalField("1. kB")
>>> f = PhysicalField("25. degC")
>>> e.unit * f.unit
Traceback (most recent call last):
...
TypeError: cannot multiply units with non-zero offset
>>> e.unit * f.inBaseUnits().unit
<PhysicalUnit kB*K>
```

__rtruediv__ (*other*)

Divide something by a unit


```
>>> a = PhysicalField("1. m")
>>> 3. / a.unit
<PhysicalUnit 3.0/m>
```

Units must have zero offset to be divided

```
>>> b = PhysicalField("25. degC")
>>> 3. / b.unit
Traceback (most recent call last):
...
TypeError: cannot divide units with non-zero offset
>>> 3. / b.inBaseUnits().unit
<PhysicalUnit 3.0/K>
```

__str__()

Return representation of a physical unit

```
>>> PhysicalUnit('m', 1., [1, 0, 0, 0, 0, 0, 0, 0, 0])
<PhysicalUnit m>
```

__truediv__ (other)

Divide one unit by another

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.unit / b.unit
<PhysicalUnit m/ft>
>>> a.unit / b.inBaseUnits().unit
<PhysicalUnit 1>
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.unit / d.unit
<PhysicalUnit s/Hz>
>>> c.unit / d.inBaseUnits().unit
<PhysicalUnit s**2/1>
```

or divide units by numbers

```
>>> a.unit / 3.
<PhysicalUnit m/3.0>
```

Units must have zero offset to be divided

```
>>> e = PhysicalField("1. J")
>>> f = PhysicalField("25. degC")
>>> e.unit / f.unit
Traceback (most recent call last):
...
TypeError: cannot divide units with non-zero offset
>>> e.unit / f.inBaseUnits().unit
<PhysicalUnit J/K>
```

__weakref__

list of weak references to the object (if defined)

conversionFactorTo (other)

Return the multiplication factor between two physical units

```
>>> a = PhysicalField("1. mm")
>>> b = PhysicalField("1. inch")
>>> print(numerix.round_(b.unit.conversionFactorTo(a.unit), 6))
25.4
```

Units must have the same fundamental SI units

```
>>> c = PhysicalField("1. K")
>>> c.unit.conversionFactorTo(a.unit)
Traceback (most recent call last):
...
TypeError: Incompatible units
```

If units have different offsets, they must have the same factor

```
>>> d = PhysicalField("1. degC")
>>> c.unit.conversionFactorTo(d.unit)
1.0
>>> e = PhysicalField("1. degF")
>>> c.unit.conversionFactorTo(e.unit)
Traceback (most recent call last):
...
TypeError: Unit conversion (K to degF) cannot be expressed as a simple_
↳multiplicative factor
```

conversionTupleTo (*other*)

Return a *tuple* of the multiplication factor and offset between two physical units

```
>>> a = PhysicalField("1. K").unit
>>> b = PhysicalField("1. degF").unit
>>> from builtins import str
>>> [str(numerix.round_(element, 6)) for element in b.conversionTupleTo(a)]
['0.555556', '459.67']
```

isAngle ()

Returns *True* if the unit is an angle

```
>>> PhysicalField("1. deg").unit.isAngle()
1
>>> PhysicalField("1. rad").unit.isAngle()
1
>>> PhysicalField("1. inch").unit.isAngle()
0
```

isCompatible (*other*)

Returns a list of which fundamental SI units are compatible between *self* and *other*

```
>>> a = PhysicalField("1. mm")
>>> b = PhysicalField("1. inch")
>>> print(numerix.allclose(a.unit.isCompatible(b.unit),
...                        [True, True, True, True, True, True, True, True,
↳True]))
True
>>> c = PhysicalField("1. K")
>>> print(numerix.allclose(a.unit.isCompatible(c.unit),
...                        [False, True, True, True, False, True, True, True,
↳True]))
```

(continues on next page)

(continued from previous page)

```
True
```

isDimensionless()Returns *True* if the unit is dimensionless

```
>>> PhysicalField("1. m/m").unit.isDimensionless()
1
>>> PhysicalField("1. inch").unit.isDimensionless()
0
```

isDimensionlessOrAngle()Returns *True* if the unit is dimensionless or an angle

```
>>> PhysicalField("1. m/m").unit.isDimensionlessOrAngle()
1
>>> PhysicalField("1. deg").unit.isDimensionlessOrAngle()
1
>>> PhysicalField("1. rad").unit.isDimensionlessOrAngle()
1
>>> PhysicalField("1. inch").unit.isDimensionlessOrAngle()
0
```

isInverseAngle()Returns *True* if the 1 divided by the unit is an angle

```
>>> PhysicalField("1. deg**-1").unit.isInverseAngle()
1
>>> PhysicalField("1. 1/rad").unit.isInverseAngle()
1
>>> PhysicalField("1. inch").unit.isInverseAngle()
0
```

name()

Return the name of the unit

```
>>> PhysicalField("1. m").unit.name()
'm'
>>> (PhysicalField("1. m") / PhysicalField("1. s"))
... / PhysicalField("1. s")).unit.name()
'm/s**2'
```

setName(name)Set the name of the unit to *name*

```
>>> a = PhysicalField("1. m/s").unit
>>> a
<PhysicalUnit m/s>
>>> a.setName('meterpersecond')
>>> a
<PhysicalUnit meterpersecond>
```

Module contents

35.2 Submodules

35.3 fipy.tools.debug module

`fipy.tools.debug.PRINT (label, arg="", stall=True)`

35.4 fipy.tools.decorators module

`fipy.tools.decorators.deprecate (*args, **kwargs)`

Issues a generic *DeprecationWarning*.

This function may also be used as a decorator.

Parameters

- **func** (*function*) – The function to be deprecated.
- **old_name** (*str*, *optional*) – The name of the function to be deprecated. Default is *None*, in which case the name of *func* is used.
- **new_name** (*str*, *optional*) – The new name for the function. Default is *None*, in which case the deprecation message is that *old_name* is deprecated. If given, the deprecation message is that *old_name* is deprecated and *new_name* should be used instead.
- **message** (*str*, *optional*) – Additional explanation of the deprecation. Displayed in the docstring after the warning.

Returns *old_func* – The deprecated function.

Return type *function*

35.5 fipy.tools.dump module

`fipy.tools.dump.write (data, filename=None, extension="", communicator=SerialPETScCommWrapper())`

Pickle an object and write it to a file. Wrapper for *cPickle.dump()*.

Test to check pickling and unpickling.

```
>>> from fipy.meshes import Grid1D
>>> old = Grid1D(nx = 2)
>>> f, tempfile = write(old)
>>> new = read(tempfile, f)
>>> print (old.numberOfCells == new.numberOfCells)
True
```

Parameters

- **data** – Object to be pickled.
- **filename** (*str*) – Name of the file to place the pickled object. If *filename* is *None* then a temporary file will be used and the file object and file name will be returned as a tuple

- **extension** (*str*) – Used if *filename* is not given.
- **communicator** (*CommWrapper*) – A duck-typed object with *procID* and *Nproc* attributes is sufficient

```
fipy.tools.dump.read(filename, fileobject=None, communicator=SerialPETScCommWrapper(),
                    mesh_unmangle=False)
```

Read a pickled object from a file. Returns the unpickled object. Wrapper for *cPickle.load()*.

Parameters

- **filename** (*str*) – Name of the file to unpickle the object from.
- **fileobject** (*file*) – Used to remove temporary files
- **communicator** (*CommWrapper*) – A duck-typed object with *procID* and *Nproc* attributes is sufficient
- **mesh_unmangle** (*bool*) – Whether to correct improper pickling of non-uniform meshes (ticket:243)

35.6 fipy.tools.inline module

35.7 fipy.tools.numerix module

Replacement module for NumPy

Attention: This module should be the only place in the code where `numpy` is explicitly imported and you should always import this module and not `numpy` in your own code. The documentation for `numpy` remains canonical for all functions and classes not explicitly documented here.

The functions provided in this module replace and augment the *NumPy* module. The functions work with *Variables*, arrays or numbers. For example, create a *Variable*.

```
>>> from fipy.variables.variable import Variable
>>> var = Variable(value=0)
```

Take the tangent of such a variable. The returned value is itself a *Variable*.

```
>>> v = tan(var)
>>> v
tan(Variable(value=array(0)))
>>> print(float(v))
0.0
```

Take the tangent of a int.

```
>>> tan(0)
0.0
```

Take the tangent of an array.

```
>>> print(tan(array((0, 0, 0))))
[ 0.  0.  0.]
```

`fipy.tools.numerix.dot(a1, a2, axis=0)`

return array of vector dot-products of *v1* and *v2* for arrays *a1* and *a2* of vectors *v1* and *v2*

We can't use `numpy.dot()` on an array of vectors

Test that *Variables* are returned as *Variables*.

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(nx=2, ny=1)
>>> from fipy.variables.cellVariable import CellVariable
>>> v1 = CellVariable(mesh=mesh, value=((0, 1), (2, 3)), rank=1)
>>> v2 = CellVariable(mesh=mesh, value=((0, 1), (2, 3)), rank=1)
>>> dot(v1, v2)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> dot(v2, v1)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print(rank(dot(v2, v1)))
0
>>> print(dot(v1, v2))
[ 4 10]
>>> dot(v1, v1)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print(dot(v1, v1))
[ 4 10]
>>> v3 = array(((0, 1), (2, 3)))
>>> print(isinstance(dot(v3, v3), type(array(1))))
1
>>> print(dot(v3, v3))
[ 4 10]
```

`fipy.tools.numerix.isclose(first, second, rtol=1e-05, atol=1e-08)`

Returns which elements of *first* and *second* are equal, subject to the given relative and absolute tolerances, such that:

$$|first - second| < atol + rtol * |second|$$

This means essentially that both elements are small compared to `atol` or their difference divided by *second*'s value is small compared to `rtol`.

`fipy.tools.numerix.allclose(first, second, rtol=1e-05, atol=1e-08)`

Tests whether or not *first* and *second* are equal, subject to the given relative and absolute tolerances, such that:

$$|first - second| < atol + rtol * |second|$$

This means essentially that both elements are small compared to `atol` or their difference divided by *second*'s value is small compared to `rtol`.

`fipy.tools.numerix.all(a, axis=None, out=None)`

Test whether all array elements along a given axis evaluate to True.

Parameters

- **a** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (*int, optional*) – Axis along which an logical AND is performed. The default (*axis = None*) is to perform a logical AND over a flattened input array. *axis* may be negative, in which case it counts from the last to the first axis.
- **out** (*ndarray, optional*) – Alternative output array in which to place the result. It must have the same shape as the expected output and the type is preserved.

`fipy.tools.numerix.put(arr, ids, values)`

The opposite of *take*. The values of *arr* at the locations specified by *ids* are set to the corresponding value of *values*.

The following is to test improvements to puts with masked arrays. Places in the code were assuming incorrect put behavior.

```
>>> maskValue = 999999
```

```
>>> arr = zeros(3, 'l')
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values) ## this should work
>>> print(arr)
[0 0 4]
```

```
>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print(arr) ## works as expected
[-- 5 4]
```

```
>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((maskValue, 2), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print(arr) ## should be [-- 5 --] maybe??
[-- 5 999999]
```

`fipy.tools.numerix.reshape(arr, shape)`

Change the shape of *arr* to *shape*, as long as the product of all the lengths of all the axes is constant (the total number of elements does not change).

`fipy.tools.numerix.sum(arr, axis=0)`

The sum of all the elements of *arr* along the specified axis.

`fipy.tools.numerix.take(a, indices, axis=0, fill_value=None)`

Selects the elements of *a* corresponding to *indices*.

`fipy.tools.numerix.all(a, axis=None, out=None)`

Test whether all array elements along a given axis evaluate to True.

Parameters

- **a** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (*int, optional*) – Axis along which an logical AND is performed. The default (*axis = None*) is to perform a logical AND over a flattened input array. *axis* may be negative, in which case it counts from the last to the first axis.
- **out** (*ndarray, optional*) – Alternative output array in which to place the result. It must have the same shape as the expected output and the type is preserved.

`fipy.tools.numerix.put(arr, ids, values)`

The opposite of *take*. The values of *arr* at the locations specified by *ids* are set to the corresponding value of *values*.

The following is to test improvements to puts with masked arrays. Places in the code were assuming incorrect put behavior.

```
>>> maskValue = 999999
```

```
>>> arr = zeros(3, 'l')
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values) ## this should work
>>> print(arr)
[0 0 4]
```

```
>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print(arr) ## works as expected
[-- 5 4]
```

```
>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((maskValue, 2), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print(arr) ## should be [-- 5 --] maybe??
[-- 5 999999]
```

`fipy.tools.numerix.reshape` (*arr*, *shape*)

Change the shape of *arr* to *shape*, as long as the product of all the lengths of all the axes is constant (the total number of elements does not change).

`fipy.tools.numerix.sum` (*arr*, *axis=0*)

The sum of all the elements of *arr* along the specified axis.

`fipy.tools.numerix.take` (*a*, *indices*, *axis=0*, *fill_value=None*)

Selects the elements of *a* corresponding to *indices*.

`fipy.tools.numerix.all` (*a*, *axis=None*, *out=None*)

Test whether all array elements along a given axis evaluate to True.

Parameters

- **a** (*array_like*) – Input array or object that can be converted to an array.
- **axis** (*int*, *optional*) – Axis along which an logical AND is performed. The default (*axis=None*) is to perform a logical AND over a flattened input array. *axis* may be negative, in which case it counts from the last to the first axis.
- **out** (*ndarray*, *optional*) – Alternative output array in which to place the result. It must have the same shape as the expected output and the type is preserved.

`fipy.tools.numerix.allclose` (*first*, *second*, *rtol=1e-05*, *atol=1e-08*)

Tests whether or not *first* and *second* are equal, subject to the given relative and absolute tolerances, such that:

```
|first - second| < atol + rtol * |second|
```

This means essentially that both elements are small compared to *atol* or their difference divided by *second*'s value is small compared to *rtol*.

`fipy.tools.numerix.allegal` (*first*, *second*)

Returns *true* if every element of *first* is equal to the corresponding element of *second*.

`fipy.tools.numerix.dot(a1, a2, axis=0)`

return array of vector dot-products of *v1* and *v2* for arrays *a1* and *a2* of vectors *v1* and *v2*

We can't use `numpy.dot()` on an array of vectors

Test that *Variables* are returned as *Variables*.

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(nx=2, ny=1)
>>> from fipy.variables.cellVariable import CellVariable
>>> v1 = CellVariable(mesh=mesh, value=((0, 1), (2, 3)), rank=1)
>>> v2 = CellVariable(mesh=mesh, value=((0, 1), (2, 3)), rank=1)
>>> dot(v1, v2)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> dot(v2, v1)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print(rank(dot(v2, v1)))
0
>>> print(dot(v1, v2))
[ 4 10]
>>> dot(v1, v1)._variableClass
<class 'fipy.variables.cellVariable.CellVariable'>
>>> print(dot(v1, v1))
[ 4 10]
>>> v3 = array(((0, 1), (2, 3)))
>>> print(isinstance(dot(v3, v3), type(array(1))))
1
>>> print(dot(v3, v3))
[ 4 10]
```

`fipy.tools.numerix.getShape(arr)`

Return the shape of *arr*

```
>>> getShape(1)
()
>>> getShape(1.)
()
>>> from fipy.variables.variable import Variable
>>> getShape(Variable(1))
()
>>> getShape(Variable(1.))
()
>>> getShape(Variable(1., unit="m"))
()
>>> getShape(Variable("1 m"))
()
```

`fipy.tools.numerix.getUnit(arr)`

`fipy.tools.numerix.isclose(first, second, rtol=1e-05, atol=1e-08)`

Returns which elements of *first* and *second* are equal, subject to the given relative and absolute tolerances, such that:

$$|first - second| < atol + rtol * |second|$$

This means essentially that both elements are small compared to *atol* or their difference divided by *second*'s value is small compared to *rtol*.

`fipy.tools.numerix.isFloat(arr)`

`fipy.tools.numerix.isInt(arr)`

`fipy.tools.numerix.L1norm(arr)`

Taxicab or Manhattan norm of *arr*

$\|arr\|_1 = \sum_{j=1}^n |arr_j|$ is the L^1 norm of *arr*.

Parameters *arr* (*ndarray*) –

`fipy.tools.numerix.L2norm(arr)`

Euclidean norm of *arr*

$\|arr\|_2 = \sqrt{\sum_{j=1}^n |arr_j|^2}$ is the L^2 norm of *arr*.

Parameters *arr* (*ndarray*) –

`fipy.tools.numerix.LINFnorm(arr)`

Infinity norm of *arr*

$\|arr\|_\infty = [\sum_{j=1}^n |arr_j|^\infty]^\infty = \max_j |arr_j|$ is the L^∞ norm of *arr*.

Parameters *arr* (*ndarray*) –

`fipy.tools.numerix.nearest(data, points, max_mem=10000000.0)`

find the indices of *data* that are closest to *points*

```
>>> from fipy import *
>>> m0 = Grid2D(dx=(.1, 1., 10.), dy=(.1, 1., 10.))
>>> m1 = Grid2D(nx=2, ny=2, dx=5., dy=5.)
>>> print(nearest(m0.cellCenters.globalValue, m1.cellCenters.globalValue))
[4 5 7 8]
>>> print(nearest(m0.cellCenters.globalValue, m1.cellCenters.globalValue, max_
↪mem=100))
[4 5 7 8]
>>> print(nearest(m0.cellCenters.globalValue, m1.cellCenters.globalValue, max_
↪mem=10000))
[4 5 7 8]
```

`fipy.tools.numerix.put(arr, ids, values)`

The opposite of *take*. The values of *arr* at the locations specified by *ids* are set to the corresponding value of *values*.

The following is to test improvements to puts with masked arrays. Places in the code were assuming incorrect put behavior.

```
>>> maskValue = 999999
```

```
>>> arr = zeros(3, 'l')
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values) ## this should work
>>> print(arr)
[0 0 4]
```

```
>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((2, maskValue), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print(arr) ## works as expected
[-- 5 4]
```

```

>>> arr = MA.masked_values((maskValue, 5, 10), maskValue)
>>> ids = MA.masked_values((maskValue, 2), maskValue)
>>> values = MA.masked_values((4, maskValue), maskValue)
>>> put(arr, ids, values)
>>> print(arr) ## should be [-- 5 --] maybe??
[-- 5 999999]

```

`fipy.tools.numerix.rank(a)`

Get the rank of sequence *a* (the number of dimensions, not a matrix rank) The rank of a scalar is zero.

Note: The rank of a *MeshVariable* is for any single element. E.g., A *CellVariable* containing scalars at each cell, and defined on a 9 element *Grid1D*, has rank 0. If it is defined on a 3x3 *Grid2D*, it is still rank 0.

`fipy.tools.numerix.reshape(arr, shape)`

Change the shape of *arr* to *shape*, as long as the product of all the lengths of all the axes is constant (the total number of elements does not change).

`fipy.tools.numerix.sqrtDot(a1, a2)`

Return array of square roots of vector dot-products for arrays *a1* and *a2* of vectors *v1* and *v2*

Usually used with *v1==v2* to return magnitude of *v1*.

`fipy.tools.numerix.sum(arr, axis=0)`

The sum of all the elements of *arr* along the specified axis.

`fipy.tools.numerix.take(a, indices, axis=0, fill_value=None)`

Selects the elements of *a* corresponding to *indices*.

`fipy.tools.numerix.tostring(arr, max_line_width=75, precision=8, suppress_small=False, separator=',', array_output=0)`

Returns a textual representation of a number or field of numbers. Each dimension is indicated by a pair of matching square brackets (*[]*), within which each subset of the field is output. The orientation of the dimensions is as follows: the last (rightmost) dimension is always horizontal, so that the frequent rank-1 fields use a minimum of screen real-estate. The next-to-last dimension is displayed vertically if present and any earlier dimension is displayed with additional bracket divisions.

```

>>> from fipy import Variable
>>> print(tostring(Variable((1, 0, 11.2345)), precision=1))
[ 1.   0.  11.2]
>>> print(tostring(array((1, 2)), precision=5))
[1 2]
>>> print(tostring(array((1.12345, 2.79)), precision=2))
[ 1.12  2.79]
>>> print(tostring(1))
1
>>> print(tostring(array(1)))
1
>>> print(tostring(array([1.23345]), precision=2))
[ 1.23]
>>> print(tostring(array([1]), precision=2))
[1]
>>> print(tostring(1.123456, precision=2))
1.12
>>> print(tostring(array(1.123456), precision=3))
1.123

```

Parameters

- **max_line_width** (*int*) – Maximum number of characters used in a single line. Default is *sys.output_line_width* or 77.
- **precision** (*int*) – Number of digits after the decimal point. Default is *sys.float_output_precision* or 8.
- **suppress_small** (*bool*) – Whether small values should be suppressed (and output as 0). Default is *sys.float_output_suppress_small* or *False*.
- **separator** (*str*) – What character string to place between two numbers.
- **array_output** (*bool*) – *unused*

35.8 fipy.tools.parser module

`fipy.tools.parser.parse` (*larg*, *action=None*, *type=None*, *default=None*)

This is a wrapper function for the python *optparse* module. Unfortunately *optparse* does not allow command line arguments to be ignored. See the documentation for *optparse* for more details. Returns the argument value.

Parameters

- **larg** (*str*) – Argument to be parsed.
- **action** ({'store', 'store_true', 'store_false', 'store_const', 'append', 'count', 'callback'}) – Basic type of action to be taken when this argument is encountered at the command line. See <https://docs.python.org/2/library/argparse.html#action>
- **type** (*type*) – Type to which the command-line argument should be converted
- **default** – Value produced if the argument is absent from the command line

35.9 fipy.tools.test module

35.10 fipy.tools.vector module

Vector utility functions that are inexplicably absent from Numeric

`fipy.tools.vector.putAdd` (*vector*, *ids*, *additionVector*)

This is a temporary replacement for *Numeric.put* as it was not doing what we thought it was doing.

`fipy.tools.vector.prune` (*array*, *shift*, *start=0*, *axis=0*)

removes elements with indices $i = \text{start} + \text{shift} * n$ where $n = 0, 1, 2, \dots$

```
>>> prune(numerix.arange(10), 3, 5)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> prune(numerix.arange(10), 3, 2)
array([0, 1, 3, 4, 6, 7, 9])
>>> prune(numerix.arange(10), 3)
array([1, 2, 4, 5, 7, 8])
>>> prune(numerix.arange(4, 7), 3)
array([5, 6])
```

35.11 fipy.tools.vitals module

class `fipy.tools.vitals.Vitals`

Bases: `xml.dom.minidom.Document`

Returns XML formatted information about current FiPy environment

__init__ ()

Initialize self. See `help(type(self))` for accurate signature.

__module__ = `'fipy.tools.vitals'`

__str__ ()

Return `str(self)`.

appendChild (*child*)

appendInfo (*name*, *svnpath=None*, ***kwargs*)

append some additional information, possibly about a project under a separate svn repository

childNodes

dictToXML (*d*, *name*)

doctype

save (*fname*)

svn (**args*)

svncmd (*cmd*, **args*)

tupleToXML (*t*, *name*, *keys=None*)

35.12 Module contents

class `fipy.tools.PhysicalField` (*value*, *unit=None*, *array=None*)

Bases: `object`

Physical field or quantity with units

Physical Fields can be constructed in one of two ways:

- `PhysicalField(*value*, *unit*)`, where **value** is a number of arbitrary type and **unit** is a string containing the unit name

```
>>> print(PhysicalField(value = 10., unit = 'm'))
10.0 m
```

- `PhysicalField(*string*)`, where **string** contains both the value and the unit. This form is provided to make interactive use more convenient

```
>>> print(PhysicalField(value = "10. m"))
10.0 m
```

Dimensionless quantities, with a *unit* of 1, can be specified in several ways

```
>>> print(PhysicalField(value = "1"))
1.0 1
>>> print(PhysicalField(value = 2., unit = " "))
```

(continues on next page)

(continued from previous page)

```
2.0 1
>>> print(PhysicalField(value = 2.))
2.0 1
```

Physical arrays are also possible (and are the reason this code was adapted from [Konrad Hinsén's](#) original `PhysicalQuantity`). The *value* can be a `Numeric array`:

```
>>> a = numerix.array(((3., 4.), (5., 6.)))
>>> print(PhysicalField(value = a, unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or a *tuple*:

```
>>> print(PhysicalField(value = ((3., 4.), (5., 6.)), unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or as a single value to be applied to every element of a supplied array:

```
>>> print(PhysicalField(value = 2., unit = "m", array = a))
[[ 2.  2.]
 [ 2.  2.]] m
```

Every element in an array has the same unit, which is stored only once for the whole array.

__abs__()

Return the absolute value of the quantity. The *unit* is unchanged.

```
>>> print(abs(PhysicalField(((3., -2.), (-1., 4.)), 'm')))
[[ 3.  2.]
 [ 1.  4.]] m
```

__add__ (other)

Add two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'm'))
10.01 km
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

__array__ (t=None)

Return a dimensionless *PhysicalField* as a `Numeric array`.

```
>>> print(numerix.array(PhysicalField(((2., 3.), (4., 5.)), "m/m")))
[[ 2.  3.]
 [ 4.  5.]]
```

As a special case, fields with angular units are converted to base units (radians) and then assumed dimensionless.

```
>>> print(numerix.array(PhysicalField(((2., 3.), (4., 5.)), "deg")))
[[ 0.03490659  0.05235988]
 [ 0.06981317  0.08726646]]
```

If the array is not dimensionless, the numerical value in its base units is returned.

```
>>> numerix.array(PhysicalField((2., 3.), (4., 5.)), "mm")
array([[ 0.002,  0.003],
       [ 0.004,  0.005]])
```

__array_priority__ = 100.0

__array_wrap__ (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples of ufuncs.

```
>>> from fipy.tools.dimensions.physicalField import PhysicalField
>>> print(type(numerix.array([1.0, 2.0]) * PhysicalField([1.0, 2.0], unit="m
↳ ")))
<class 'fipy.tools.dimensions.physicalField.PhysicalField'>
```

```
>>> print(type(numerix.array([1.0, 2.0]) * PhysicalField([1.0, 2.0])))
<class 'fipy.tools.dimensions.physicalField.PhysicalField'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(isinstance(Gamma(PhysicalField([1.0, 2.0])), type(numerix.
↳ array(1))))
1
```

__bool__ ()

Test if the quantity is zero.

Should this only pass if the unit offset is zero?

__dict__ = mappingproxy({'__module__': 'fipy.tools.dimensions.physicalField', '__doc__

__div__ (*other*)

Divide two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print(PhysicalField(10., 'm') / PhysicalField(2., 's'))
5.0 m/s
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *I*. This facilitates passing physical quantities to packages such as [Numeric](#) that cannot use units, while ensuring the quantities have the desired units

```
>>> print((PhysicalField(1., 'inch')
...       / PhysicalField(1., 'mm')))
25.4
```

__eq__ (*other*)

Return self==value.

__float__ ()

Return a dimensionless *PhysicalField* quantity as a float.

```
>>> float(PhysicalField("2. m/m"))
2.0
```

As a special case, quantities with angular units are converted to base units (radians) and then assumed dimensionless.

```
>>> print(numerix.round_(float(PhysicalField("2. deg")), 6))
0.034907
```

If the quantity is not dimensionless, the conversion fails.

```
>>> float(PhysicalField("2. m"))
Traceback (most recent call last):
...
TypeError: Not possible to convert a PhysicalField with dimensions to float
```

Just as a `Numeric` array cannot be cast to float, neither can *PhysicalField* arrays

```
>>> float(PhysicalField(((2., 3.), (4., 5.)), "m/m"))
Traceback (most recent call last):
...
TypeError: only ...-1 arrays can be converted to Python scalars
```

`__ge__` (*other*)

Return `self >= value`.

`__getitem__` (*index*)

Return the specified element of the array. The unit of the result will be the unit of the array.

```
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> print(a[1, 1])
6.0 m
```

`__gt__` (*other*)

Compare *self* to *other*, returning an array of Boolean values corresponding to the test against each element.

```
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> print(numerix.allclose(a > PhysicalField("13 ft"),
...                          [[False, True], [ True, True]]))
True
```

Appropriately formatted dimensional quantity strings can also be compared.

```
>>> print(numerix.allclose(a > "13 ft",
...                          [[False, True], [ True, True]]))
True
```

Arrays are compared element to element

```
>>> print(numerix.allclose(a > PhysicalField(((3., 13.), (17., 6.)), "ft"),
...                          [[ True, True], [False, True]]))
True
```

Units must be compatible

```
>>> print(a > PhysicalField("1 lb"))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

And so must array dimensions


```
>>> print(a > PhysicalField(((3., 13., 4.), (17., 6., 2.)), "ft"))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

__hash__()
Return hash(self).

__init__(value, unit=None, array=None)
Physical Fields can be constructed in one of two ways:

- *PhysicalField(*value*, *unit*)*, where **value** is a number of arbitrary type and **unit** is a string containing the unit name

```
>>> print(PhysicalField(value = 10., unit = 'm'))
10.0 m
```

- *PhysicalField(*string*)*, where **string** contains both the value and the unit. This form is provided to make interactive use more convenient

```
>>> print(PhysicalField(value = "10. m"))
10.0 m
```

Dimensionless quantities, with a *unit* of 1, can be specified in several ways

```
>>> print(PhysicalField(value = "1"))
1.0 1
>>> print(PhysicalField(value = 2., unit = " "))
2.0 1
>>> print(PhysicalField(value = 2.))
2.0 1
```

Physical arrays are also possible (and are the reason this code was adapted from Konrad Hinsen's original *PhysicalQuantity*). The *value* can be a *Numeric array*:

```
>>> a = numerix.array(((3., 4.), (5., 6.)))
>>> print(PhysicalField(value = a, unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or a *tuple*:

```
>>> print(PhysicalField(value = ((3., 4.), (5., 6.)), unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or as a single value to be applied to every element of a supplied array:

```
>>> print(PhysicalField(value = 2., unit = "m", array = a))
[[ 2.  2.]
 [ 2.  2.]] m
```

Every element in an array has the same unit, which is stored only once for the whole array.

__le__(other)
Return self<=value.

__len__()

`__lt__ (other)`

Return self<value.

`__mod__ (other)`

Return the remainder of dividing two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print(PhysicalField(11., 'm') % PhysicalField(2., 's'))
1.0 m/s
```

`__module__ = 'fipy.tools.dimensions.physicalField'`

`__mul__ (other)`

Multiply two physical quantities. The unit of the result is the product of the units of the operands.

```
>>> print(PhysicalField(10., 'N') * PhysicalField(10., 'm') ==
↳ PhysicalField(100., 'N*m'))
True
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *I*. This facilitates passing physical quantities to packages such as Numeric that cannot use units, while ensuring the quantities have the desired units.

```
>>> print((PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
```

`__ne__ (other)`

Return self!=value.

`__neg__ ()`

Return the negative of the quantity. The *unit* is unchanged.

```
>>> print(-PhysicalField(((3., -2.), (-1., 4.)), 'm'))
[[-3.  2.]
 [ 1. -4.]] m
```

`__nonzero__ ()`

Test if the quantity is zero.

Should this only pass if the unit offset is zero?

`__pos__ ()`

`__pow__ (other)`

Raise a *PhysicalField* to a power. The unit is raised to the same power.

```
>>> print(PhysicalField(10., 'm')**2)
100.0 m**2
```

`__radd__ (other)`

Add two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'm'))
10.01 km
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

`__rdiv__ (other)`

`__repr__ ()`

Return representation of a physical quantity suitable for re-use

```
>>> PhysicalField(value = 3., unit = "eV")
PhysicalField(3.0, 'eV')
```

`__rmul__ (other)`

Multiply two physical quantities. The unit of the result is the product of the units of the operands.

```
>>> print(PhysicalField(10., 'N') * PhysicalField(10., 'm') ==
PhysicalField(100., 'N*m'))
True
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *I*. This facilitates passing physical quantities to packages such as Numeric that cannot use units, while ensuring the quantities have the desired units.

```
>>> print((PhysicalField(10., 's') * PhysicalField(2., 'Hz'))
20.0
```

`__rpow__ (other)`

`__rsub__ (other)`

`__rtruediv__ (other)`

`__setitem__ (index, value)`

Assign the specified element of the array, performing appropriate conversions.

```
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> a[0, 1] = PhysicalField("6 ft")
>>> print(a)
[[ 3.      1.8288]
 [ 5.      6.     ]] m
>>> a[1, 0] = PhysicalField("2 min")
Traceback (most recent call last):
...
TypeError: Incompatible units
```

`__str__ ()`

Return human-readable form of a physical quantity

```
>>> print(PhysicalField(value = 3., unit = "eV"))
3.0 eV
```

`__sub__ (other)`

Subtract two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') - PhysicalField(10., 'm'))
9.99 km
>>> print(PhysicalField(10., 'km') - PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

__truediv__ (*other*)

Divide two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print(PhysicalField(10., 'm') / PhysicalField(2., 's'))
5.0 m/s
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *I*. This facilitates passing physical quantities to packages such as [Numeric](#) that cannot use units, while ensuring the quantities have the desired units

```
>>> print((PhysicalField(1., 'inch')
...       / PhysicalField(1., 'mm')))
25.4
```

__weakref__

list of weak references to the object (if defined)

add (*other*)

Add two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'm'))
10.01 km
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

allclose (*other*, *atol*=None, *rtol*=1e-08)

This function tests whether or not *self* and *other* are equal subject to the given relative and absolute tolerances. The formula used is:

```
| self - other | < atol + rtol * | other |
```

This means essentially that both elements are small compared to *atol* or their difference divided by *other*'s value is small compared to *rtol*.

allequal (*other*)

This function tests whether or not *self* and *other* are exactly equal.

arccos ()

Return the inverse cosine of the *PhysicalField* in radians

```
>>> print(PhysicalField(0).arccos().allclose("1.57079632679 rad"))
1
```

The input *PhysicalField* must be dimensionless

```
>>> print( Numerix.round_(PhysicalField("1 m").arccos(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arccosh ()

Return the inverse hyperbolic cosine of the *PhysicalField*

```
>>> print(numerix.allclose(PhysicalField(2).arccosh(),
...                        1.31695789692))
1
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1. m").arccosh(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arcsin()

Return the inverse sine of the *PhysicalField* in radians

```
>>> print(PhysicalField(1).arcsin().allclose("1.57079632679 rad"))
1
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1 m").arcsin(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arctan()

Return the arctangent of the *PhysicalField* in radians

```
>>> print(numerix.round_(PhysicalField(1).arctan(), 6))
0.785398
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1 m").arctan(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arctan2(*other*)

Return the arctangent of *self* divided by *other* in radians

```
>>> print(numerix.round_(PhysicalField(2.).arctan2(PhysicalField(5.)), 6))
0.380506
```

The input *PhysicalField* objects must be in the same dimensions

```
>>> print(numerix.round_(PhysicalField(2.54, "cm").arctan2(PhysicalField(1.,
↳ "inch")), 6))
0.785398
```

```
>>> print(numerix.round_(PhysicalField(2.).arctan2(PhysicalField("5. m")), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

arctanh()

Return the inverse hyperbolic tangent of the *PhysicalField*

```
>>> print(PhysicalField(0.5).arctanh())
0.549306144334
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1 m").arctanh(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

ceil()

Return the smallest integer greater than or equal to the *PhysicalField*.

```
>>> print(PhysicalField(2.2, "m").ceil())
3.0 m
```

conjugate()

Return the complex conjugate of the *PhysicalField*.

```
>>> print(PhysicalField(2.2 - 3j, "ohm").conjugate() == PhysicalField(2.2 +
↵3j, "ohm"))
True
```

convertToUnit (unit)

Changes the unit to *unit* and adjusts the value such that the combination is equivalent. The new unit is by a string containing its name. The new unit must be compatible with the previous unit of the object.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> e.convertToUnit('kcal/mol')
>>> print(e)
1694.27557621 kcal/mol
```

copy()

Make a duplicate.

```
>>> a = PhysicalField(1, unit = 'inch')
>>> b = a.copy()
```

The duplicate will not reflect changes made to the original

```
>>> a.convertToUnit('cm')
>>> print(a)
2.54 cm
>>> print(b)
1 inch
```

Likewise for arrays

```
>>> a = PhysicalField(numerix.array((0, 1, 2)), unit = 'm')
>>> b = a.copy()
>>> a[0] = 3
>>> print(a)
[3 1 2] m
>>> print(b)
[0 1 2] m
```

cos()

Return the cosine of the *PhysicalField*

```
>>> print(numerix.round_(PhysicalField(2*numerix.pi/6, "rad").cos(), 6))
0.5
>>> print(numerix.round_(PhysicalField(60., "deg").cos(), 6))
0.5
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(60., "m").cos()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

cosh()

Return the hyperbolic cosine of the *PhysicalField*

```
>>> PhysicalField(0.).cosh()
1.0
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60., "m").cosh()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

divide (*other*)

Divide two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print(PhysicalField(10., 'm') / PhysicalField(2., 's'))
5.0 m/s
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *1*. This facilitates passing physical quantities to packages such as [Numeric](#) that cannot use units, while ensuring the quantities have the desired units

```
>>> print((PhysicalField(1., 'inch')
...       / PhysicalField(1., 'mm')))
25.4
```

dot (*other*)

Return the dot product of *self* with *other*. The resulting unit is the product of the units of *self* and *other*.

```
>>> v = PhysicalField(((5., 6.), (7., 8.)), "m")
>>> print(PhysicalField(((1., 2.), (3., 4.)), "m").dot(v))
[ 26.  44.] m**2
```

floor()

Return the largest integer less than or equal to the *PhysicalField*.

```
>>> print(PhysicalField(2.2, "m").floor())
2.0 m
```

getsctype (*default=None*)

Returns the NumPy sctype of the underlying array.

```
>>> PhysicalField(1, 'm').getscdtype() == numerix.NUMERIX.obj2scdtype(numerix.
↪array(1))
True
>>> PhysicalField(1., 'm').getscdtype() == numerix.NUMERIX.obj2scdtype(numerix.
↪array(1.))
True
>>> PhysicalField((1, 1.), 'm').getscdtype() == numerix.NUMERIX.
↪obj2scdtype(numerix.array((1., 1.)))
True
```

inBaseUnits()

Return the quantity with all units reduced to their base SI elements.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

inDimensionless()

Returns the numerical value of a dimensionless quantity.

```
>>> print(PhysicalField(((2., 3.), (4., 5.))).inDimensionless())
[[ 2.  3.]
 [ 4.  5.]]
```

It's an error to convert a quantity with units

```
>>> print(PhysicalField(((2., 3.), (4., 5.))), "m").inDimensionless()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

inRadians()

Converts an angular quantity to radians and returns the numerical value.

```
>>> print(PhysicalField(((2., 3.), (4., 5.))), "rad").inRadians()
[[ 2.  3.]
 [ 4.  5.]]
>>> print(PhysicalField(((2., 3.), (4., 5.))), "deg").inRadians()
[[ 0.03490659  0.05235988]
 [ 0.06981317  0.08726646]]
```

As a special case, assumes a dimensionless quantity is already in radians.

```
>>> print(PhysicalField(((2., 3.), (4., 5.))).inRadians()
[[ 2.  3.]
 [ 4.  5.]]
```

It's an error to convert a quantity with non-angular units

```
>>> print(PhysicalField(((2., 3.), (4., 5.))), "m").inRadians()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

inSIUnits()

Return the quantity with all units reduced to SI-compatible elements.


```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print(e.inSIUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

inUnitsOf(*units)

Returns one or more *PhysicalField* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *PhysicalField*.

```
>>> freeze = PhysicalField('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *PhysicalField* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = PhysicalField(314159., 's')
>>> from builtins import zip
>>> print( Numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d',
↳ 'h', 'min', 's'),
                                                                    ['3.0 d', '15.0 h
...                                                                    ],
↳ ', '15.0 min', '59.0 s'])],
...                                     True))
1
```

isCompatible(unit)**itemset(value)**

Assign the value of a scalar array, performing appropriate conversions.

```
>>> a = PhysicalField(4., "m")
>>> a.itemset(PhysicalField("6 ft"))
>>> print(a.allclose("1.8288 m"))
1
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> try:
...     a.itemset(PhysicalField("6 ft"))
... except IndexError:
...     # NumPy 1.7 has changed the exception type
...     raise ValueError("can only place a scalar for an array of size 1")
Traceback (most recent call last):
...
ValueError: can only convert an array of size 1 to a Python scalar
>>> a.itemset(PhysicalField("2 min"))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

property itemsize**log()**

Return the natural logarithm of the *PhysicalField*

```
>>> print(Numerix.round_(PhysicalField(10).log(), 6))
2.302585
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1. m").log(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

log10()

Return the base-10 logarithm of the *PhysicalField*

```
>>> print(numerix.round_(PhysicalField(10.).log10(), 6))
1.0
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round_(PhysicalField("1. m").log10(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

multiply (other)

Multiply two physical quantities. The unit of the result is the product of the units of the operands.

```
>>> print(PhysicalField(10., 'N') * PhysicalField(10., 'm') ==
↳ PhysicalField(100., 'N*m'))
True
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *I*. This facilitates passing physical quantities to packages such as Numeric that cannot use units, while ensuring the quantities have the desired units.

```
>>> print((PhysicalField(10., 's') * PhysicalField(2., 'Hz')))
20.0
```

property numericValue

Return the *PhysicalField* without units, after conversion to base SI units.

```
>>> print(numerix.round_(PhysicalField("1 inch").numericValue, 6))
0.0254
```

put (indices, values)

put is the opposite of *take*. The values of *self* at the locations specified in *indices* are set to the corresponding value of *values*.

The *indices* can be any integer sequence object with values suitable for indexing into the flat form of *self*. The *values* must be any sequence of values that can be converted to the typecode of *self*.

```
>>> f = PhysicalField((1., 2., 3.), "m")
>>> f.put((2, 0), PhysicalField((2., 3.), "inch"))
>>> print(f)
[ 0.0762  2.      0.0508] m
```

The units of *values* must be compatible with *self*.

```
>>> f.put(1, PhysicalField(3, "kg"))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

ravel()

reshape(*shape*)

Changes the shape of *self* to that specified in *shape*

```
>>> print(PhysicalField((1., 2., 3., 4.), "m").reshape((2, 2)))
[[ 1.  2.]
 [ 3.  4.]] m
```

The new shape must have the same size as the existing one.

```
>>> print(PhysicalField((1., 2., 3., 4.), "m").reshape((2, 3)))
Traceback (most recent call last):
...
ValueError: total size of new array must be unchanged
```

property shape

Tuple of array dimensions.

sign()

Return the sign of the quantity. The *unit* is unchanged.

```
>>> from fipy.tools.numerix import sign
>>> print(sign(PhysicalField((3., -2.), (-1., 4.)), 'm'))
[[ 1. -1.]
 [-1.  1.]]
```

sin()

Return the sine of the *PhysicalField*

```
>>> print(PhysicalField(numerix.pi/6, "rad").sin())
0.5
>>> print(PhysicalField(30., "deg").sin())
0.5
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(30., "m").sin()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

sinh()

Return the hyperbolic sine of the *PhysicalField*

```
>>> PhysicalField(0.).sinh()
0.0
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60., "m").sinh()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

sqrt()

Return the square root of the *PhysicalField*

```
>>> print(PhysicalField("100. m**2").sqrt())
10.0 m
```

The resulting unit must be integral

```
>>> print(PhysicalField("100. m").sqrt())
Traceback (most recent call last):
...
TypeError: Illegal exponent
```

subtract (*other*)

Subtract two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') - PhysicalField(10., 'm'))
9.99 km
>>> print(PhysicalField(10., 'km') - PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

sum (*index=0*)

Returns the sum of all of the elements in *self* along the specified axis (first axis by default).

```
>>> print(PhysicalField(((1., 2.), (3., 4.)), "m").sum())
[ 4.  6.] m
>>> print(PhysicalField(((1., 2.), (3., 4.)), "m").sum(1))
[ 3.  7.] m
```

take (*indices, axis=0*)

Return the elements of *self* specified by the elements of *indices*. The resulting *PhysicalField* array has the same units as the original.

```
>>> print(PhysicalField((1., 2., 3.), "m").take((2, 0)))
[ 3.  1.] m
```

The optional third argument specifies the axis along which the selection occurs, and the default value (as in the example above) is 0, the first axis.

```
>>> print(PhysicalField(((1., 2., 3.), (4., 5., 6.)), "m").take((2, 0), axis_
↪ = 1))
[[ 3.  1.]
 [ 6.  4.]] m
```

tan ()

Return the tangent of the *PhysicalField*

```
>>> numerix.round_(PhysicalField(numerix.pi/4, "rad").tan(), 6)
1.0
>>> numerix.round_(PhysicalField(45, "deg").tan(), 6)
1.0
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(45., "m").tan()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: Incompatible units
```

tanh()Return the hyperbolic tangent of the *PhysicalField*

```
>>> print(numerix.allclose(PhysicalField(1.).tanh(), 0.761594155956))
True
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60., "m").tanh()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

tostring (*max_line_width=75, precision=8, suppress_small=False, separator=' '*)

Return human-readable form of a physical quantity

```
>>> p = PhysicalField(value = (3., 3.14159), unit = "eV")
>>> print(p.tostring(precision = 3, separator = '|'))
[ 3.    | 3.142] eV
```

property unitReturn the unit object of *self*.

```
>>> PhysicalField("1 m").unit
<PhysicalUnit m>
```

class fipy.tools.Vitals

Bases: xml.dom.minidom.Document

Returns XML formatted information about current FiPy environment

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

__module__ = 'fipy.tools.vitals'**__str__** ()

Return str(self).

appendChild (*child*)**appendInfo** (*name, svnpath=None, **kwargs*)

append some additional information, possibly about a project under a separate svn repository

childNodes**dictToXML** (*d, name*)**doctype****save** (*fname*)**svn** (**args*)**svncmd** (*cmd, *args*)**tupleToXML** (*t, name, keys=None*)

Chapter 36

fipy.variables package

36.1 Submodules

36.2 fipy.variables.addOverFacesVariable module

36.3 fipy.variables.arithmeticCellToFaceVariable module

36.4 fipy.variables.betaNoiseVariable module

class fipy.variables.betaNoiseVariable.BetaNoiseVariable(*mesh*, *alpha*, *beta*, *name*="", *hasOld*=0)

Bases: *fipy.variables.noiseVariable.NoiseVariable*

Represents a beta distribution of random numbers with the probability distribution

$$x^{\alpha-1} \frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α , a rate parameter β , and $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform Cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = BetaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), alpha = alpha,
↪ beta = beta)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.01, nx = 100)
```

and compare to a Gaussian distribution

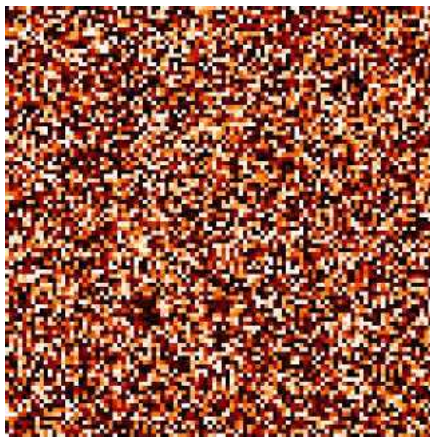
```
>>> from fipy.variables.cellVariable import CellVariable
>>> betadist = CellVariable(mesh = histogram.mesh)
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> betadist = ((Gamma(alpha + beta) / (Gamma(alpha) * Gamma(beta)))
...             * x**(alpha - 1) * (1 - x)**(beta - 1))
```

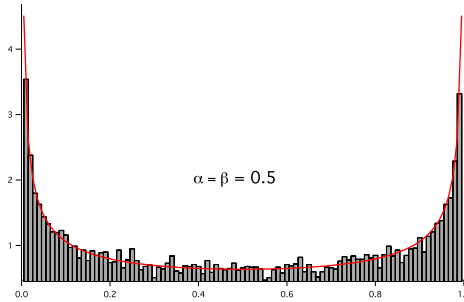
```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=1)
...     histoplot = Viewer(vars=(histogram, betadist),
...                           datamin=0, datamax=1.5)
```

```
>>> from fipy.tools.numerix import arange
```

```
>>> for a in arange(0.5, 5, 0.5):
...     alpha.value = a
...     for b in arange(0.5, 5, 0.5):
...         beta.value = b
...         if __name__ == '__main__':
...             import sys
...             print("alpha: %g, beta: %g" % (alpha, beta), file=sys.stderr)
...             viewer.plot()
...             histoplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```





Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **alpha** (*float*) – The parameter α .
- **beta** (*float*) – The parameter β .

`__init__(mesh, alpha, beta, name="", hasOld=0)`

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **alpha** (*float*) – The parameter α .
- **beta** (*float*) – The parameter β .

`__module__ = 'fipy.variables.betaNoiseVariable'`

`random()`

36.5 fipy.variables.binaryOperatorVariable module

36.6 fipy.variables.cellToFaceVariable module

36.7 fipy.variables.cellVariable module

class `fipy.variables.cellVariable.CellVariable` (*mesh*, *name=""*, *value=0.0*, *rank=None*, *elementshape=None*, *unit=None*, *hasOld=0*)

Bases: `fipy.variables.meshVariable._MeshVariable`

Represents the field of values of a variable on a *Mesh*.

A *CellVariable* can be pickled to persistent storage (disk) for later use:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 10, ny = 10)
```

```
>>> var = CellVariable(mesh = mesh, value = 1., hasOld = 1, name = 'test')
>>> x, y = mesh.cellCenters
>>> var.value = (x * y)
```

```
>>> from fipy.tools import dump
>>> (f, filename) = dump.write(var, extension = '.gz')
>>> unPickledVar = dump.read(filename, f)
```

```
>>> print(var.allclose(unPickledVar, atol = 1e-10, rtol = 1e-10))
1
```

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float* or *array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple* of *int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable

__call__ (*points=None, order=0, nearestCellIDs=None*)

Interpolates the *CellVariable* to a set of points using a method that has a memory requirement on the order of *Ncells* by *Npoints* in general, but uses only *Ncells* when the *CellVariable*'s mesh is a *UniformGrid* object.

Tests

```
>>> from fipy import *
>>> m = Grid2D(nx=3, ny=2)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> print(v(((0., 1.1, 1.2), (0., 1., 1.))))
[ 0.5  1.5  1.5]
>>> print(v(((0., 1.1, 1.2), (0., 1., 1.)), order=1))
[ 0.25  1.1  1.2 ]
>>> m0 = Grid2D(nx=2, ny=2, dx=1., dy=1.)
>>> m1 = Grid2D(nx=4, ny=4, dx=.5, dy=.5)
>>> x, y = m0.cellCenters
>>> v0 = CellVariable(mesh=m0, value=x * y)
>>> print(v0(m1.cellCenters.globalValue))
[ 0.25  0.25  0.75  0.75  0.25  0.25  0.75  0.75  0.75  0.75  2.25  2.25
  0.75  0.75  2.25  2.25]
>>> print(v0(m1.cellCenters.globalValue, order=1))
[ 0.125  0.25  0.5  0.625  0.25  0.375  0.875  1.  0.5  0.875
  1.875  2.25  0.625  1.  2.25  2.625]
```

Parameters

- **points** (*tuple* or *list* of *tuple*) – A point or set of points in the format (X, Y, Z)
- **order** (*{0, 1}*) – The order of interpolation, default is 0
- **nearestCellIDs** (*array_like*) – Optional argument if user can calculate own nearest cell IDs array, shape should be same as points

__getstate__ ()

Used internally to collect the necessary information to *pickle* the *CellVariable* to persistent storage.

`__init__` (*mesh*, *name*="", *value*=0.0, *rank*=None, *elementshape*=None, *unit*=None, *hasOld*=0)

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float* or *array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple* of *int*) – the shape of each element of this variable Default: *rank* * (*mesh.dim*,)
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable

`__module__` = 'fipy.variables.cellVariable'

`__setstate__` (*dict*)

Used internally to create a new *CellVariable* from pickled persistent storage.

property arithmeticFaceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

property cellVolumeAverage

Return the cell-volume-weighted average of the *CellVariable*:

$$\langle \phi \rangle_{\text{vol}} = \frac{\sum_{\text{cells}} \phi_{\text{cell}} V_{\text{cell}}}{\sum_{\text{cells}} V_{\text{cell}}}$$

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .1)
>>> var = CellVariable(value = (1, 2, 6), mesh = mesh)
>>> print(var.cellVolumeAverage)
3.0
```

constrain (*value*, *where=None*)

Constrains the *CellVariable* to *value* at a location specified by *where*.

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> v.constrain(0., where=m.facesLeft)
>>> v.faceGrad.constrain([1.], where=m.facesRight)
>>> print(v.faceGrad)
[[ 1.  1.  1.  1.]]
>>> print(v.faceValue)
[ 0.  1.  2.  2.5]
```

Changing the constraint changes the dependencies

```
>>> v.constrain(1., where=m.facesLeft)
>>> print(v.faceGrad)
[[-1.  1.  1.  1.]]
>>> print(v.faceValue)
[ 1.  1.  2.  2.5]
```

Constraints can be *Variable*

```
>>> c = Variable(0.)
>>> v.constrain(c, where=m.facesLeft)
>>> print(v.faceGrad)
[[ 1.  1.  1.  1.]]
>>> print(v.faceValue)
[ 0.  1.  2.  2.5]
>>> c.value = 1.
>>> print(v.faceGrad)
[[-1.  1.  1.  1.]]
>>> print(v.faceValue)
[ 1.  1.  2.  2.5]
```

Constraints can have a *Variable* mask.

```
>>> v = CellVariable(mesh=m)
>>> mask = FaceVariable(mesh=m, value=m.facesLeft)
>>> v.constrain(1., where=mask)
>>> print(v.faceValue)
[ 1.  0.  0.  0.]
>>> mask[:] = mask | m.facesRight
>>> print(v.faceValue)
[ 1.  0.  0.  1.]
```

copy ()

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0, 1, 2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

property faceGradAverage

Deprecated since version 3.3: use `grad.arithmeticFaceValue()` instead

Return $\nabla\phi$ as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

property faceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
```

(continues on next page)

(continued from previous page)

```
>>> print( numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

property gaussGrad

Return $\frac{1}{V_p} \sum_f \vec{n}_f \phi_f A_f$ as a rank-1 *CellVariable* (first-order gradient).

property globalValue

Concatenate and return values from all processors

When running on a single processor, the result is identical to *value*.

property grad

Return $\nabla \phi$ as a rank-1 *CellVariable* (first-order gradient).

property harmonicFaceValue

Returns a *FaceVariable* whose value corresponds to the harmonic interpolation of the adjacent cells:

$$\phi_f = \frac{\phi_1 \phi_2}{(\phi_2 - \phi_1) \frac{d_{f2}}{d_{12}} + \phi_1}$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (0.5 / 1.) + L)
>>> print( numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (1.0 / 3.0) + L)
>>> print( numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (5.0 / 55.0) + L)
>>> print( numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

property leastSquaresGrad

Return $\nabla \phi$, which is determined by solving for $\nabla \phi$ in the following matrix equation,

$$\nabla \phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n} \cdot \nabla \phi)_{AP}$$

The matrix equation is derived by minimizing the following least squares sum,

$$F(\phi_x, \phi_y) = \sqrt{\sum_f (d_{AP} \vec{n}_{AP} \cdot \nabla \phi - d_{AP} (\vec{n}_{AP} \cdot \nabla \phi)_{AP})^2}$$

Tests

```
>>> from fipy import Grid2D
>>> m = Grid2D(nx=2, ny=2, dx=0.1, dy=2.0)
>>> print(numerix.allclose(CellVariable(mesh=m, value=(0, 1, 3, 6)).
↪leastSquaresGrad.globalValue, \
...                                     [[8.0, 8.0, 24.0, 24.0],
...                                     [1.2, 2.0, 1.2, 2.0]]))
True
```

```
>>> from fipy import Grid1D
>>> print(numerix.allclose(CellVariable(mesh=Grid1D(dx=(2.0, 1.0, 0.5)),
...                                     value=(0, 1, 2)).leastSquaresGrad.
↪globalValue, [[0.461538461538, 0.8, 1.2]]))
True
```

property minmodFaceValue

Returns a *FaceVariable* with a value that is the minimum of the absolute values of the adjacent cells. If the values are of opposite sign then the result is zero:

$$\phi_f = \begin{cases} \phi_1 & \text{when } |\phi_1| \leq |\phi_2|, \\ \phi_2 & \text{when } |\phi_2| < |\phi_1|, \\ 0 & \text{when } \phi_1\phi_2 < 0 \end{cases}$$

```
>>> from fipy import *
>>> print(CellVariable(mesh=Grid1D(nx=2), value=(1, 2)).minmodFaceValue)
[1 1 2]
>>> print(CellVariable(mesh=Grid1D(nx=2), value=(-1, -2)).minmodFaceValue)
[-1 -1 -2]
>>> print(CellVariable(mesh=Grid1D(nx=2), value=(-1, 2)).minmodFaceValue)
[-1 0 2]
```

property old

Return the values of the *CellVariable* from the previous solution sweep.

Combinations of *CellVariable*'s should also return old values.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> var1 = CellVariable(mesh = mesh, value = (2, 3), hasOld = 1)
>>> var2 = CellVariable(mesh = mesh, value = (3, 4))
>>> v = var1 * var2
>>> print(v)
[ 6 12]
>>> var1.value = ((3, 2))
>>> print(v)
[9 8]
>>> print(v.old)
[ 6 12]
```

The following small test is to correct for a bug when the operator does not just use variables.

```
>>> v1 = var1 * 3
>>> print(v1)
[9 6]
>>> print(v1.old)
[6 9]
```

release (*constraint*)
Remove *constraint* from *self*

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> c = Constraint(0., where=m.facesLeft)
>>> v.constrain(c)
>>> print(v.faceValue)
[ 0.  1.  2.  2.5]
>>> v.release(constraint=c)
>>> print(v.faceValue)
[ 0.5  1.  2.  2.5]
```

setValue (*value*, *unit=None*, *where=None*)
Set the value of the Variable. Can take a masked array.

```
>>> a = Variable((1, 2, 3))
>>> a.setValue(5, where=(1, 0, 1))
>>> print(a)
[5 2 5]
```

```
>>> b = Variable((4, 5, 6))
>>> a.setValue(b, where=(1, 0, 1))
>>> print(a)
[4 2 6]
>>> print(b)
[4 5 6]
>>> a.value = 3
>>> print(a)
[3 3 3]
```

```
>>> b = numerix.array((3, 4, 5))
>>> a.value = b
>>> a[:] = 1
>>> print(b)
[3 4 5]
```

```
>>> a.setValue((4, 5, 6), where=(1, 0))
Traceback (most recent call last):
....
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

updateOld ()
Set the values of the previous solution sweep to the current values.

```
>>> from fipy import *
>>> v = CellVariable(mesh=Grid1D(), hasOld=False)
>>> v.updateOld()
Traceback (most recent call last):
....
AssertionError: The updateOld method requires the CellVariable to have an old_
↳value. Set hasOld to True when instantiating the CellVariable.
```


36.8 fipy.variables.constant module

36.9 fipy.variables.constraintMask module

36.10 fipy.variables.coupledCellVariable module

36.11 fipy.variables.distanceVariable module

class fipy.variables.distanceVariable.DistanceVariable (*mesh, name="", value=0.0, unit=None, hasOld=0*)

Bases: *fipy.variables.cellVariable.CellVariable*

A *DistanceVariable* object calculates ϕ so it satisfies,

$$|\nabla\phi| = 1$$

using the fast marching method with an initial condition defined by the zero level set. The solution can either be first or second order.

Here we will define a few test cases. Firstly a 1D test case

```
>>> from fipy.meshes import Grid1D
>>> from fipy.tools import serialComm
>>> mesh = Grid1D(dx = .5, nx = 8, communicator=serialComm)
>>> from .distanceVariable import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = (-1., -1., -1., -1., 1., 1., 1., 1.))
>>> var.calcDistanceFunction()
>>> answer = (-1.75, -1.25, -.75, -0.25, 0.25, 0.75, 1.25, 1.75)
>>> print(var.allclose(answer))
1
```

A 1D test case with very small dimensions.

```
>>> dx = 1e-10
>>> mesh = Grid1D(dx = dx, nx = 8, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., -1., -1., -1., 1., 1., 1., 1.))
>>> var.calcDistanceFunction()
>>> answer = numerix.arange(8) * dx - 3.5 * dx
>>> print(var.allclose(answer))
1
```

A 2D test case to test *_calcTrialValue* for a pathological case.

```
>>> dx = 1.
>>> dy = 2.
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = 2, ny = 3, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., 1., 1., 1., -1., 1.))

>>> var.calcDistanceFunction()
>>> vbl = -dx * dy / numerix.sqrt(dx**2 + dy**2) / 2.
>>> vbr = dx / 2
```

(continues on next page)

(continued from previous page)

```

>>> vml = dy / 2.
>>> crossProd = dx * dy
>>> dsq = dx**2 + dy**2
>>> top = vbr * dx**2 + vml * dy**2
>>> sqrt = crossProd**2 * (dsq - (vbr - vml)**2)
>>> sqrt = numerix.sqrt(max(sqrt, 0))
>>> vmr = (top + sqrt) / dsq
>>> answer = (vbl, vbr, vml, vmr, vbl, vbr)
>>> print(var.allclose(answer))
1

```

The *extendVariable* method solves the following equation for a given *extensionVariable*.

$$\nabla u \cdot \nabla \phi = 0$$

using the fast marching method with an initial condition defined at the zero level set.

```

>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., 1., 1., 1.))
>>> var.calcDistanceFunction()
>>> extensionVar = CellVariable(mesh = mesh, value = (-1, .5, 2, -1))
>>> tmp = 1 / numerix.sqrt(2)
>>> print(var.allclose((-tmp / 2, 0.5, 0.5, 0.5 + tmp)))
1
>>> var.extendVariable(extensionVar, order=1)
>>> print(extensionVar.allclose((1.25, .5, 2, 1.25)))
1
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., 1., 1.,
...                                             1., 1., 1.,
...                                             1., 1., 1.))
>>> var.calcDistanceFunction(order=1)
>>> extensionVar = CellVariable(mesh = mesh, value = (-1., .5, -1.,
...                                                    2., -1., -1.,
...                                                    -1., -1., -1.))

```

```

>>> v1 = 0.5 + tmp
>>> v2 = 1.5
>>> tmp1 = (v1 + v2) / 2 + numerix.sqrt(2. - (v1 - v2)**2) / 2
>>> tmp2 = tmp1 + 1 / numerix.sqrt(2)
>>> print(var.allclose((-tmp / 2, 0.5, 1.5, 0.5, 0.5 + tmp,
...                    tmp1, 1.5, tmp1, tmp2)))
1
>>> answer = (1.25, .5, .5, 2, 1.25, 0.9544, 2, 1.5456, 1.25)
>>> var.extendVariable(extensionVar, order=1)
>>> print(extensionVar.allclose(answer, rtol = 1e-4))
1

```

Test case for a bug that occurs when initializing the distance variable at the interface. Currently it is assumed that adjacent cells that are opposite sign neighbors have perpendicular normal vectors. In fact the two closest cells could have opposite normals.

```

>>> mesh = Grid1D(dx = 1., nx = 3, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., 1., -1.))
>>> var.calcDistanceFunction()

```

(continues on next page)

(continued from previous page)

```
>>> print(var.allclose((-0.5, 0.5, -0.5)))
1
```

Testing second order. This example failed with Scikit-fmm.

```
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 4, ny = 4, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., -1., 1., 1.,
...                                             -1., -1., 1., 1.,
...                                             1., 1., 1., 1.,
...                                             1, 1, 1, 1))
>>> var.calcDistanceFunction(order=2)
>>> answer = [-1.30473785, -0.5, 0.5, 1.49923009,
...          -0.5, -0.35355339, 0.5, 1.45118446,
...          0.5, 0.5, 0.97140452, 1.76215286,
...          1.49923009, 1.45118446, 1.76215286, 2.33721352]
>>> print(numerix.allclose(var, answer, rtol=1e-9))
True
```

**** A test for a bug in both LSMLIB and Scikit-fmm ****

The following test gives different result depending on whether LSMLIB or Scikit-fmm is used. There is a deeper problem that is related to this issue. When a value becomes “known” after previously being a “trial” value it updates its neighbors’ values. In a second order scheme the neighbors one step away also need to be updated (if the in between cell is “known” and the far cell is a “trial” cell), but are not in either package. By luck (due to trial values having the same value), the values calculated in Scikit-fmm for the following example are correct although an example that didn’t work for Scikit-fmm could also be constructed.

```
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 4, ny = 4, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., -1., -1., -1.,
...                                             1., 1., -1., -1.,
...                                             1., 1., -1., -1.,
...                                             1., 1., -1., -1.))
>>> var.calcDistanceFunction(order=2)
>>> var.calcDistanceFunction(order=2)
>>> answer = [-0.5,      -0.58578644, -1.08578644, -1.85136395,
...          0.5,      0.29289322,  -0.58578644, -1.54389939,
...          1.30473785, 0.5,      -0.5,      -1.5,
...          1.49547948, 0.5,      -0.5,      -1.5]
```

The 3rd and 7th element are different for LSMLIB. This is because the 15th element is not “known” when the “trial” value for the 7th element is calculated. Scikit-fmm calculates the values in a slightly different order so gets a seemingly better answer, but this is just chance.

```
>>> print(numerix.allclose(var, answer, rtol=1e-9))
True
```

Creates a *distanceVariable* object.

Parameters

- **mesh** (*Mesh*) – The mesh that defines the geometry of this variable.
- **name** (*str*) – The name of the variable.
- **value** (*float* or *array_like*) – The initial value.
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable
- **hasOld** (*bool*) – Whether the variable maintains an old value.

`__init__` (*mesh*, *name*="", *value*=0.0, *unit*=None, *hasOld*=0)
Creates a *distanceVariable* object.

Parameters

- **mesh** (*Mesh*) – The mesh that defines the geometry of this variable.
- **name** (*str*) – The name of the variable.
- **value** (*float* or *array_like*) – The initial value.
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable
- **hasOld** (*bool*) – Whether the variable maintains an old value.

`__module__` = 'fipy.variables.distanceVariable'

`calcDistanceFunction` (*order*=2)
Calculates the *distanceVariable* as a distance function.

Parameters *order* ({1, 2}) – The order of accuracy for the distance function calculation

property `cellInterfaceAreas`

Returns the length of the interface that crosses the cell

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-1.5, -0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh, value=(0, 0., 1., 0))
>>> print( Numerix.allclose(distanceVariable.cellInterfaceAreas,
...                           answer))
...
True
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (1.5, 0.5, 1.5,
...                                               0.5, -0.5, 0.5,
...                                               1.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                         value=(0, 1, 0, 1, 0, 1, 0, 1, 0))
>>> print( Numerix.allclose(distanceVariable.cellInterfaceAreas, answer))
True
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-0.5, 0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                         value=(0, Numerix.sqrt(2) / 4, Numerix.sqrt(2) / 4,
...                               0))
>>> print( Numerix.allclose(distanceVariable.cellInterfaceAreas,
...                           answer))
...
True
```

Test to check that the circumference of a circle is, in fact, $2\pi r$.

```
>>> mesh = Grid2D(dx = 0.05, dy = 0.05, nx = 20, ny = 20)
>>> r = 0.25
>>> x, y = mesh.cellCenters
>>> rad = numerix.sqrt((x - .5)**2 + (y - .5)**2) - r
>>> distanceVariable = DistanceVariable(mesh = mesh, value = rad)
>>> print(numerix.allclose(distanceVariable.cellInterfaceAreas.sum(), 1.
↪57984690073))
1
```

extendVariable (*extensionVariable*, *order=2*)

Calculates the extension of *extensionVariable* from the zero level set.

Parameters **extensionVariable** (*CellVariable*) – The variable to extend from the zero level set.

getLSMshape ()

36.12 fipy.variables.exponentialNoiseVariable module

class `fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable` (*mesh*, *mean=0.0*, *name=""*, *hasOld=0*)

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents an exponential distribution of random numbers with the probability distribution

$$\mu^{-1}e^{-\frac{x}{\mu}}$$

with a mean parameter μ .

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform Cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> mean = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = ExponentialNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), mean = ↪
↪mean)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 100)
```

and compare to a Gaussian distribution

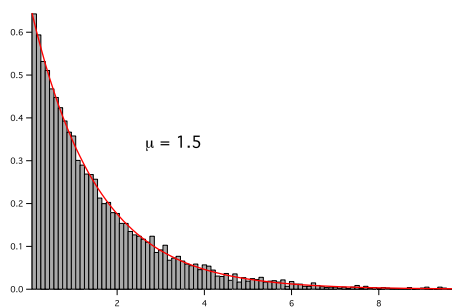
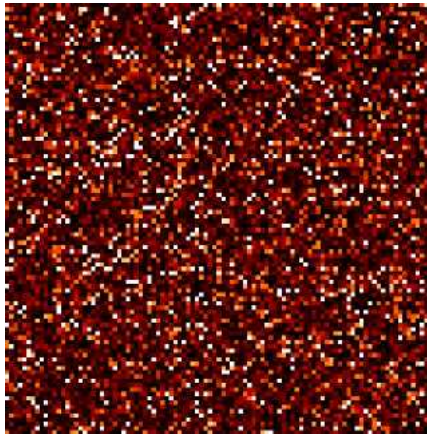
```
>>> from fipy.variables.cellVariable import CellVariable
>>> expdist = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=5)
...     histplot = Viewer(vars=(histogram, expdist),
...                           datamin=0, datamax=1.5)
```

```
>>> from fipy.tools.numerix import arange, exp
```

```
>>> for mu in arange(0.5, 3, 0.5):
...     mean.value = (mu)
...     expdist.value = ((1/mean)*exp(-x/mean))
...     if __name__ == '__main__':
...         import sys
...         print("mean: %g" % mean, file=sys.stderr)
...         viewer.plot()
...         histplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```



Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **mean** (*float*) – The mean of the distribution μ .

`__init__(mesh, mean=0.0, name="", hasOld=0)`

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.

- **mean** (*float*) – The mean of the distribution μ .

```
__module__ = 'fipy.variables.exponentialNoiseVariable'
random()
```

36.13 fipy.variables.faceGradContributionsVariable module

36.14 fipy.variables.faceGradVariable module

36.15 fipy.variables.faceVariable module

```
class fipy.variables.faceVariable.FaceVariable (mesh, name="", value=0.0, rank=None,
                                                elementshape=None, unit=None,
                                                cached=1)
```

Bases: `fipy.variables.meshVariable._MeshVariable`

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float or array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple of int*) – the shape of each element of this variable Default: `rank * (mesh.dim,)`
- **unit** (*str or PhysicalUnit*) – The physical units of the variable

```
__module__ = 'fipy.variables.faceVariable'
```

```
copy()
```

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0, 1, 2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

property divergencethe divergence of *self*, \vec{u} ,

$$\nabla \cdot \vec{u} \approx \frac{\sum_f (\vec{u} \cdot \hat{n})_f A_f}{V_P}$$

Returns **divergence** – one rank lower than *self***Return type** *fipy.variables.cellVariable.CellVariable***Examples**

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=3, ny=2)
>>> from builtins import range
>>> var = CellVariable(mesh=mesh, value=list(range(3*2)))
>>> print(var.faceGrad.divergence)
[ 4.  3.  2. -2. -3. -4.]
```

property globalValue**setValue** (*value*, *unit=None*, *where=None*)

Set the value of the Variable. Can take a masked array.

```
>>> a = Variable((1, 2, 3))
>>> a.setValue(5, where=(1, 0, 1))
>>> print(a)
[5 2 5]
```

```
>>> b = Variable((4, 5, 6))
>>> a.setValue(b, where=(1, 0, 1))
>>> print(a)
[4 2 6]
>>> print(b)
[4 5 6]
>>> a.value = 3
>>> print(a)
[3 3 3]
```

```
>>> b = numerix.array((3, 4, 5))
>>> a.value = b
>>> a[:] = 1
>>> print(b)
[3 4 5]
```

```
>>> a.setValue((4, 5, 6), where=(1, 0))
Traceback (most recent call last):
....
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```


36.16 fipy.variables.gammaNoiseVariable module

class `fipy.variables.gammaNoiseVariable.GammaNoiseVariable` (*mesh*, *shape*, *rate*, *name*="", *hasOld*=0)

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a gamma distribution of random numbers with the probability distribution

$$x^{\alpha-1} \frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α , a rate parameter β , and $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform Cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = GammaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), shape = alpha,
↪rate = beta)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 300)
```

and compare to a Gaussian distribution

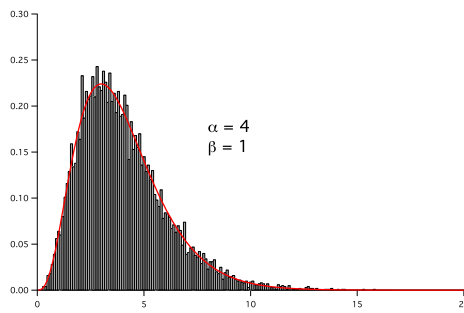
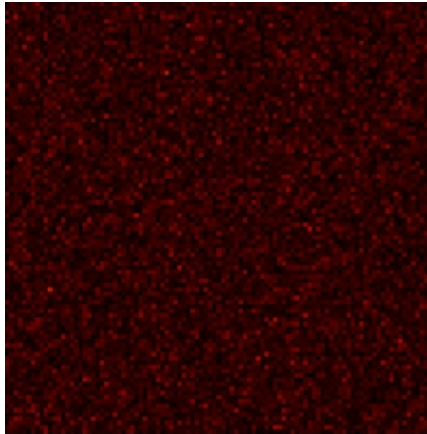
```
>>> from fipy.variables.cellVariable import CellVariable
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> from fipy.tools.numerix import exp
>>> gammadist = (x**(alpha - 1) * (beta**alpha * exp(-beta * x)) / Gamma(alpha))
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=30)
...     histoplot = Viewer(vars=(histogram, gammadist),
...                           datamin=0, datamax=1)
```

```
>>> from fipy.tools.numerix import arange
```

```
>>> for shape in arange(1, 8, 1):
...     alpha.value = shape
...     for rate in arange(0.5, 2.5, 0.5):
...         beta.value = rate
...         if __name__ == '__main__':
...             import sys
...             print("alpha: %g, beta: %g" % (alpha, beta), file=sys.stderr)
...             viewer.plot()
...             histoplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```



Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **shape** (*float*) – The shape parameter, α .
- **rate** (*float*) – The rate or inverse scale parameter, β .

```
__init__(mesh, shape, rate, name="", hasOld=0)
```

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **shape** (*float*) – The shape parameter, α .
- **rate** (*float*) – The rate or inverse scale parameter, β .

```
__module__ = 'fipy.variables.gammaNoiseVariable'
```

```
random()
```

36.17 fipy.variables.gaussCellGradVariable module

36.18 fipy.variables.gaussianNoiseVariable module

```
class fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable(mesh,
                                                                name="",
                                                                mean=0.0,
                                                                variance=1.0,
                                                                hasOld=0)
```

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a normal (Gaussian) distribution of random numbers with mean μ and variance $\langle \eta(\vec{r})\eta(\vec{r}') \rangle = \sigma^2$, which has the probability distribution

$$\frac{1}{\sigma\sqrt{2\pi}} \exp -\frac{(x-\mu)^2}{2\sigma^2}$$

For example, the variance of thermal noise that is uncorrelated in space and time is often expressed as

$$\langle \eta(\vec{r}, t)\eta(\vec{r}', t') \rangle = Mk_B T \delta(\vec{r} - \vec{r}') \delta(t - t')$$

which can be obtained with:

```
sigmaSqr = Mobility * kBoltzmann * Temperature / (mesh.cellVolumes * timeStep)
GaussianNoiseVariable(mesh = mesh, variance = sigmaSqr)
```

Note: If the time step will change as the simulation progresses, either through use of an adaptive iterator or by making manual changes at different stages, remember to declare *timeStep* as a *Variable* and to change its value with its *setValue()* method.

```
>>> import sys
>>> from fipy.tools.numerix import *
```

```
>>> mean = 0.
>>> variance = 4.
```

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(3)
```

We generate noise on a non-uniform Cartesian mesh with cell dimensions of x^2 and y^3 .

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = arange(0.1, 5., 0.1)**2, dy = arange(0.1, 3., 0.1)**3)
>>> from fipy.variables.cellVariable import CellVariable
>>> volumes = CellVariable(mesh=mesh, value=mesh.cellVolumes)
>>> noise = GaussianNoiseVariable(mesh = mesh, mean = mean,
...                               variance = variance / volumes)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise * sqrt(volumes),
...                               dx = 0.1, nx = 600, offset = -30)
...                               dx = 0.1, nx = 600, offset = -30)
```

and compare to a Gaussian distribution

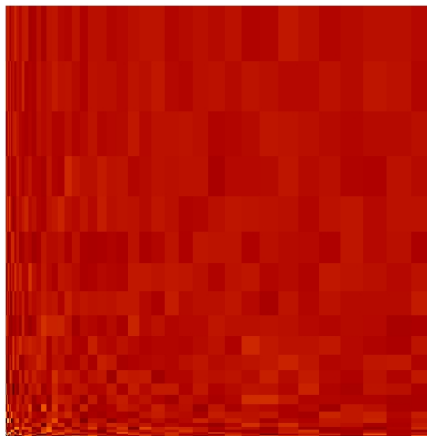
```
>>> gauss = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
>>> gauss.value = ((1/(sqrt(variance * 2 * pi))) * exp(-(x - mean)**2 / (2 *
↪variance)))
```

```
>>> if __name__ == '__main__':
...     from fipy.viewers import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=-5, datamax=5)
...     histoplot = Viewer(vars=(histogram, gauss))
```

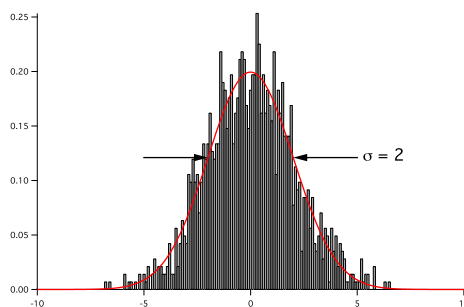
```
>>> from builtins import range
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```

Note that the noise exhibits larger amplitude in the small cells than in the large ones



but that the root-volume-weighted histogram is Gaussian.



Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **mean** (*float*) – The mean of the noise distribution, μ .
- **variance** (*float*) – The variance of the noise distribution, σ^2 .

`__init__` (*mesh*, *name=""*, *mean=0.0*, *variance=1.0*, *hasOld=0*)

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **mean** (*float*) – The mean of the noise distribution, μ .
- **variance** (*float*) – The variance of the noise distribution, σ^2 .

`__module__` = `'fipy.variables.gaussianNoiseVariable'`

`parallelRandom` ()

36.19 `fipy.variables.harmonicCellToFaceVariable` module

36.20 `fipy.variables.histogramVariable` module

`class` `fipy.variables.histogramVariable.HistogramVariable` (*distribution*, *dx=1.0*,
nx=None, *offset=0.0*)

Bases: `fipy.variables.cellVariable.CellVariable`

Produces a histogram of the values of the supplied distribution.

Parameters

- **distribution** (*array_like* or *Variable*) – The collection of values to sample.
- **dx** (*float*) – The bin size
- **nx** (*int*) – The number of bins
- **offset** (*float*) – The position of the first bin

`__init__` (*distribution*, *dx=1.0*, *nx=None*, *offset=0.0*)

Produces a histogram of the values of the supplied distribution.

Parameters

- **distribution** (*array_like* or *Variable*) – The collection of values to sample.
- **dx** (*float*) – The bin size
- **nx** (*int*) – The number of bins
- **offset** (*float*) – The position of the first bin

`__module__` = `'fipy.variables.histogramVariable'`

36.21 fipy.variables.interfaceAreaVariable module

36.22 fipy.variables.interfaceFlagVariable module

36.23 fipy.variables.leastSquaresCellGradVariable module

36.24 fipy.variables.levelSetDiffusionVariable module

36.25 fipy.variables.meshVariable module

36.26 fipy.variables.minmodCellToFaceVariable module

36.27 fipy.variables.modCellGradVariable module

36.28 fipy.variables.modCellToFaceVariable module

36.29 fipy.variables.modFaceGradVariable module

36.30 fipy.variables.modPhysicalField module

36.31 fipy.variables.modularVariable module

```
class fipy.variables.modularVariable.ModularVariable(mesh, name="", value=0.0,
                                                    rank=None, elementsshape=None, unit=None,
                                                    hasOld=0)
```

Bases: *fipy.variables.cellVariable.CellVariable*

The *ModularVariable* defines a variable that exists on the circle between $-\pi$ and π

The following examples show how *ModularVariable* works. When subtracting the answer wraps back around the circle.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.tools import numerix
>>> pi = numerix.pi
>>> v1 = ModularVariable(mesh = mesh, value = (2*pi/3, -2*pi/3))
>>> v2 = ModularVariable(mesh = mesh, value = -2*pi/3)
>>> print(numerix.allclose(v2 - v1, (2*pi/3, 0)))
1
```

Obtaining the arithmetic face value.

```
>>> print(numerix.allclose(v1.arithmeticFaceValue, (2*pi/3, pi, -2*pi/3)))
1
```

Obtaining the gradient.

```
>>> print(numerix.allclose(v1.grad, ((pi/3, pi/3),)))
1
```

Obtaining the gradient at the faces.

```
>>> print(numerix.allclose(v1.faceGrad, ((0, 2*pi/3, 0),)))
1
```

Obtaining the gradient at the faces but without modular arithmetic.

```
>>> print(numerix.allclose(v1.faceGradNoMod, ((0, -4*pi/3, 0),)))
1
```

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float* or *array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple* of *int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable

```
__module__ = 'fipy.variables.modularVariable'
```

```
__rsub__(other)
```

```
__sub__(other)
```

property arithmeticFaceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

Adjusted for a *ModularVariable*

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* (second-order gradient). Adjusted for a *ModularVariable*

property faceGradNoMod

Return $\nabla\phi$ as a rank-1 *FaceVariable* (second-order gradient). Not adjusted for a *ModularVariable*

property grad

Return $\nabla\phi$ as a rank-1 *CellVariable* (first-order gradient). Adjusted for a *ModularVariable*

updateOld()

Set the values of the previous solution sweep to the current values. Test case due to bug.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> var = ModularVariable(mesh=mesh, value=1., hasOld=1)
>>> var.updateOld()
>>> var[:] = 2
>>> answer = CellVariable(mesh=mesh, value=1.)
```

(continues on next page)

(continued from previous page)

```
>>> print(var.old.allclose(answer))
True
```

36.32 fipy.variables.noiseVariable module

class fipy.variables.noiseVariable.NoiseVariable (mesh, name="", hasOld=0)

Bases: *fipy.variables.cellVariable.CellVariable*

Attention: This class is abstract. Always create one of its subclasses.

A generic base class for sources of noise distributed over the cells of a mesh.

In the event that the noise should be conserved, use:

```
<Specific>NoiseVariable(...).faceGrad.divergence
```

The *seed()* and *get_seed()* functions of the *fipy.tools.numerix.random* module can be set and query the random number generated used by all *NoiseVariable* objects.

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float or array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple of int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str or PhysicalUnit*) – The physical units of the variable

__init__ (mesh, name="", hasOld=0)

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float or array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple of int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str or PhysicalUnit*) – The physical units of the variable

__module__ = 'fipy.variables.noiseVariable'

copy ()

Copy the value of the *NoiseVariable* to a static *CellVariable*.

parallelRandom ()


```
random()
scramble()
    Generate a new random distribution.
```

36.33 fipy.variables.operatorVariable module

36.34 fipy.variables.scharfetterGummelFaceVariable module

```
class fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable(var,
                                                                                   bound-
                                                                                   aryCon-
                                                                                   di-
                                                                                   tions=())
```

Bases: fipy.variables.cellToFaceVariable._CellToFaceVariable

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float or array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple of int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str or PhysicalUnit*) – The physical units of the variable

```
__init__(var, boundaryConditions=())
```

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float or array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple of int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str or PhysicalUnit*) – The physical units of the variable

```
__module__ = 'fipy.variables.scharfetterGummelFaceVariable'
```

36.35 fipy.variables.surfactantConvectionVariable module

class fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable (*distanceVar*)
 Bases: *fipy.variables.faceVariable.FaceVariable*

Convection coefficient for the *ConservativeSurfactantEquation*. The coefficient only has a value for a negative *distanceVar*.

Simple one dimensional test:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(nx = 3, ny = 1, dx = 1., dy = 1.)
>>> from fipy.variables.distanceVariable import DistanceVariable
>>> distanceVar = DistanceVariable(mesh, value = (-.5, .5, 1.5))
>>> ## answer = numerix.zeros((2, mesh.numberofFaces), 'd')
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[0, 7] = -1
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).globalValue,
↪answer))
True
```

Change the dimensions:

```
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .25)
>>> distanceVar = DistanceVariable(mesh, value = (-.25, .25, .75))
>>> answer[0, 7] = -.5
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).globalValue,
↪answer))
True
```

Two dimensional example:

```
>>> mesh = Grid2D(nx = 2, ny = 2, dx = 1., dy = 1.)
>>> distanceVar = DistanceVariable(mesh, value = (-1.5, -.5, -.5, .5))
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[1, 2] = -.5
>>> answer[1, 3] = -1
>>> answer[0, 7] = -.5
>>> answer[0, 10] = -1
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).globalValue,
↪answer))
True
```

Larger grid:

```
>>> mesh = Grid2D(nx = 3, ny = 3, dx = 1., dy = 1.)
>>> distanceVar = DistanceVariable(mesh, value = (1.5, .5, 1.5,
...                                              .5, -.5, .5,
...                                              1.5, .5, 1.5))
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[1, 4] = .25
>>> answer[1, 7] = -.25
>>> answer[0, 17] = .25
>>> answer[0, 18] = -.25
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).globalValue,
↪answer))
True
```

`__init__` (*distanceVar*)

Simple one dimensional test:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(nx = 3, ny = 1, dx = 1., dy = 1.)
>>> from fipy.variables.distanceVariable import DistanceVariable
>>> distanceVar = DistanceVariable(mesh, value = (-.5, .5, 1.5))
>>> ## answer = numerix.zeros((2, mesh.numberOfFaces), 'd')
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[0, 7] = -1
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).
↳globalValue, answer))
True
```

Change the dimensions:

```
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .25)
>>> distanceVar = DistanceVariable(mesh, value = (-.25, .25, .75))
>>> answer[0, 7] = -.5
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).
↳globalValue, answer))
True
```

Two dimensional example:

```
>>> mesh = Grid2D(nx = 2, ny = 2, dx = 1., dy = 1.)
>>> distanceVar = DistanceVariable(mesh, value = (-1.5, -.5, -.5, .5))
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[1, 2] = -.5
>>> answer[1, 3] = -1
>>> answer[0, 7] = -.5
>>> answer[0, 10] = -1
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).
↳globalValue, answer))
True
```

Larger grid:

```
>>> mesh = Grid2D(nx = 3, ny = 3, dx = 1., dy = 1.)
>>> distanceVar = DistanceVariable(mesh, value = (1.5, .5, 1.5,
...                                              .5, -.5, .5,
...                                              1.5, .5, 1.5))
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[1, 4] = .25
>>> answer[1, 7] = -.25
>>> answer[0, 17] = .25
>>> answer[0, 18] = -.25
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).
↳globalValue, answer))
True
```

`__module__` = 'fipy.variables.surfactantConvectionVariable'

36.36 fipy.variables.surfactantVariable module

```
class fipy.variables.surfactantVariable.SurfactantVariable (value=0.0, distanceVar=None,
name='surfactant variable', hasOld=False)
```

Bases: *fipy.variables.cellVariable.CellVariable*

The *SurfactantVariable* maintains a conserved volumetric concentration on cells adjacent to, but in front of, the interface. The *value* argument corresponds to the initial concentration of surfactant on the interface (moles divided by area). The value held by the *SurfactantVariable* is actually a volume density (moles divided by volume).

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> from fipy.variables.distanceVariable import DistanceVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (-1.5, -0.5, 0.5, 941.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                       distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable, (0, 0., 1., 0)))
1
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (1.5, 0.5, 1.5,
...                                           0.5, -0.5, 0.5,
...                                           1.5, 0.5, 1.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                       distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable, (0, 1, 0, 1, 0, 1, 0, 1, 0)))
1
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (-0.5, 0.5, 0.5, 1.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                       distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable,
...                         (0, numerix.sqrt(2), numerix.sqrt(2), 0)))
1
```

Parameters

- **value** (*float* or *array_like*) – The initial value.
- **distanceVar** (*DistanceVariable*) –
- **name** (*str*) – The name of the variable.

`__init__` (*value=0.0, distanceVar=None, name='surfactant variable', hasOld=False*)

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> from fipy.variables.distanceVariable import DistanceVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (-1.5, -0.5, 0.5, 941.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                       distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable, (0, 0., 1., 0)))
1
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (1.5, 0.5, 1.5,
...                                           0.5, -0.5, 0.5,
...                                           1.5, 0.5, 1.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                       distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable, (0, 1, 0, 1, 0, 1, 0, 1, 0)))
1
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (-0.5, 0.5, 0.5, 1.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                       distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable,
...                         (0, numerix.sqrt(2), numerix.sqrt(2), 0)))
1
```

Parameters

- **value** (*float or array_like*) – The initial value.
- **distanceVar** (*DistanceVariable*) –
- **name** (*str*) – The name of the variable.

`__module__` = 'fipy.variables.surfactantVariable'

`copy()`

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0, 1, 2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

property interfaceVar

Returns the *SurfactantVariable* rendered as an *_InterfaceSurfactantVariable* which evaluates the surfactant concentration as an area concentration the interface rather than a volumetric concentration.

36.37 fipy.variables.test module

Test numeric implementation of the mesh

36.38 fipy.variables.unaryOperatorVariable module

36.39 fipy.variables.uniformNoiseVariable module

```
class fipy.variables.uniformNoiseVariable.UniformNoiseVariable(mesh, name="",
                                                                minimum=0.0,
                                                                maximum=1.0,
                                                                hasOld=0)
```

Bases: *fipy.variables.noiseVariable.NoiseVariable*

Represents a uniform distribution of random numbers.

We generate noise on a uniform Cartesian mesh

```
>>> from fipy.meshes import Grid2D
>>> noise = UniformNoiseVariable(mesh=Grid2D(nx=100, ny=100))
```

and histogram the noise

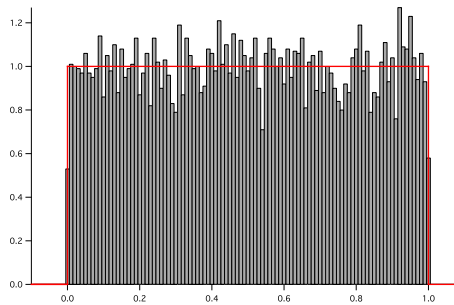
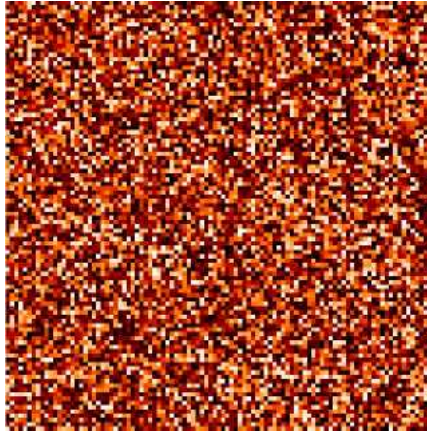
```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution=noise, dx=0.01, nx=120, offset=-.1)
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=0, datamax=1)
...     histoplot = Viewer(vars=histogram)
```

```

>>> from builtins import range
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()

```



Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **minimum** (*float*) – The minimum (not-inclusive) value of the distribution.
- **maximum** (*float*) – The maximum (not-inclusive) value of the distribution.

__init__ (*mesh*, *name=""*, *minimum=0.0*, *maximum=1.0*, *hasOld=0*)

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **minimum** (*float*) – The minimum (not-inclusive) value of the distribution.
- **maximum** (*float*) – The maximum (not-inclusive) value of the distribution.

__module__ = 'fipy.variables.uniformNoiseVariable'

random ()

36.40 fipy.variables.variable module

class fipy.variables.variable.**Variable**(*value=0.0, unit=None, array=None, name="", cached=1*)

Bases: `object`

Lazily evaluated quantity with units.

Using a *Variable* in a mathematical expression will create an automatic dependency *Variable*, e.g.,

```
>>> a = Variable(value=3)
>>> b = 4 * a
>>> b
(Variable(value=array(3)) * 4)
>>> b()
12
```

Changes to the value of a *Variable* will automatically trigger changes in any dependent *Variable* objects

```
>>> a.setValue(5)
>>> b
(Variable(value=array(5)) * 4)
>>> print(b())
20
```

Create a *Variable*.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3, 'm'))
>>> Variable(value=3, unit="m", array=numeric.zeros((3, 2), '1'))
Variable(value=PhysicalField(array([[3, 3],
[3, 3],
[3, 3]]), 'm'))
```

Parameters

- **value** (int or float or *array_like*) –
- **unit** (*str* or `PhysicalUnit`) – The physical units of the variable
- **array** (*ndarray*, *optional*) – The storage array for the *Variable*
- **name** (*str*) – The user-readable name of the *Variable*
- **cached** (*bool*) – whether to cache or always recalculate the value

__abs__()

Following test it to fix a bug with C inline string using *abs()* instead of *fabs()*

```
>>> print(abs(Variable(2.3) - Variable(1.2)))
1.1
```

__add__(*other*)

__and__(*other*)

This test case has been added due to a weird bug that was appearing.


```

>>> a = Variable(value=(0, 0, 1, 1))
>>> b = Variable(value=(0, 1, 0, 1))
>>> print(numerix.equal((a == 0) & (b == 1), [False, True, False, False])).
↳all()
True
>>> print(a & b)
[0 0 0 1]
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx=4)
>>> from fipy.variables.cellVariable import CellVariable
>>> a = CellVariable(value=(0, 0, 1, 1), mesh=mesh)
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> print(numerix.allegal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]

```

__array__ (*t=None*)

Attempt to convert the *Variable* to a numerix array object

```

>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]

```

A dimensional *Variable* will convert to the numeric value in its base units

```

>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])

```

__array_priority__ = 100.0

__array_wrap__ (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```

>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>

```

```

>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>

```

__bool__ ()

```

>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is
↳ambiguous. Use a.any() or a.all()

```

__call__ ()

“Evaluate” the *Variable* and return its value

```
>>> a = Variable(value=3)
>>> print(a())
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b()
7
```

__dict__ = `mappingproxy({'__module__': 'fipy.variables.variable', '__doc__': '\n Laz`

__div__(*other*)

__eq__(*other*)

Test if a *Variable* is equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a == 4)
>>> b
(Variable(value=array(3)) == 4)
>>> b()
0
```

__float__()

__ge__(*other*)

Test if a *Variable* is greater than or equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a >= 4)
>>> b
(Variable(value=array(3)) >= 4)
>>> b()
0
>>> a.value = 4
>>> print(b())
1
>>> a.value = 5
>>> print(b())
1
```

__getitem__(*index*)

“Evaluate” the *Variable* and return the specified element

```
>>> a = Variable(value=((3., 4.), (5., 6.)), unit="m") + "4 m"
>>> print(a[1, 1])
10.0 m
```

It is an error to slice a *Variable* whose *value* is not sliceable

```
>>> Variable(value=3)[2]
Traceback (most recent call last):
...
IndexError: 0-d arrays can't be indexed
```

__getstate__()

Used internally to collect the necessary information to pickle the *Variable* to persistent storage.

__gt__ (*other*)Test if a *Variable* is greater than another quantity

```

>>> a = Variable(value=3)
>>> b = (a > 4)
>>> b
(Variable(value=array(3)) > 4)
>>> print(b())
0
>>> a.value = 5
>>> print(b())
1

```

__hash__ ()

Return hash(self).

__init__ (*value=0.0, unit=None, array=None, name="", cached=1*)Create a *Variable*.

```

>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3, 'm'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3, 2), '1'))
Variable(value=PhysicalField(array([[3, 3],
                                   [3, 3],
                                   [3, 3]]), 'm'))

```

Parameters

- **value** (int or float or *array_like*) –
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable
- **array** (*ndarray*, *optional*) – The storage array for the *Variable*
- **name** (*str*) – The user-readable name of the *Variable*
- **cached** (*bool*) – whether to cache or always recalculate the value

__int__ ()**__invert__** ()Returns logical “not” of the *Variable*

```

>>> a = Variable(value=True)
>>> print(~a)
False

```

__iter__ ()**__le__** (*other*)Test if a *Variable* is less than or equal to another quantity

```

>>> a = Variable(value=3)
>>> b = (a <= 4)
>>> b
(Variable(value=array(3)) <= 4)
>>> b()
1

```

(continues on next page)

(continued from previous page)

```
>>> a.value = 4
>>> print(b())
1
>>> a.value = 5
>>> print(b())
0
```

__len__()**__lt__(other)**Test if a *Variable* is less than another quantity

```
>>> a = Variable(value=3)
>>> b = (a < 4)
>>> b
(Variable(value=array(3)) < 4)
>>> b()
1
>>> a.value = 4
>>> print(b())
0
>>> print(1000000000000000000 * Variable(1) < 1.)
0
>>> print(1000 * Variable(1) < 1.)
0
```

Python automatically reverses the arguments when necessary

```
>>> 4 > Variable(value=3)
(Variable(value=array(3)) < 4)
```

__mod__(other)**__module__** = 'fipy.variables.variable'**__mul__(other)****__ne__(other)**Test if a *Variable* is not equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a != 4)
>>> b
(Variable(value=array(3)) != 4)
>>> b()
1
```

__neg__()**static __new__(cls, *args, **kwargs)**

Create and return a new object. See help(type) for accurate signature.

__nonzero__()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is
↪ambiguous. Use a.any() or a.all()
```

__or__ (*other*)

This test case has been added due to a weird bug that was appearing.

```
>>> a = Variable(value=(0, 0, 1, 1))
>>> b = Variable(value=(0, 1, 0, 1))
>>> print(numerix.equal((a == 0) | (b == 1), [True, True, False, True]).
↪all())
True
>>> print(a | b)
[0 1 1 1]
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx=4)
>>> from fipy.variables.cellVariable import CellVariable
>>> a = CellVariable(value=(0, 0, 1, 1), mesh=mesh)
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> print(numerix.allegal((a == 0) | (b == 1), [True, True, False, True]))
True
>>> print(a | b)
[0 1 1 1]
```

__pos__ ()**__pow__** (*other*)

return self**other, or self raised to power other

```
>>> print(Variable(1, "mol/l")**3)
1.0 mol**3/l**3
>>> print((Variable(1, "mol/l")**3).unit)
<PhysicalUnit mol**3/l**3>
```

__radd__ (*other*)**__rdiv__** (*other*)**__repr__** ()

Return repr(self).

__rmul__ (*other*)**__rpow__** (*other*)**__rsub__** (*other*)**__rtruediv__** (*other*)**__setitem__** (*index, value*)**__setstate__** (*dict*)Used internally to create a new *Variable* from pickled persistent storage.**__str__** ()

Return str(self).

__sub__ (*other*)**__truediv__** (*other*)

`__weakref__`

list of weak references to the object (if defined)

`all` (*axis=None*)

```
>>> print(Variable(value=(0, 0, 1, 1)).all())
0
>>> print(Variable(value=(1, 1, 1, 1)).all())
1
```

`allclose` (*other, rtol=1e-05, atol=1e-08*)

```
>>> var = Variable((1, 1))
>>> print(var.allclose((1, 1)))
1
>>> print(var.allclose((1,)))
1
```

The following test is to check that the system does not run out of memory.

```
>>> from fipy.tools import numerix
>>> var = Variable(numerix.ones(10000))
>>> print(var.allclose(numerix.zeros(10000, '1')))
False
```

`allequal` (*other*)

`any` (*axis=None*)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

`cacheMe` (*recursive=False*)

`constrain` (*value, where=None*)

Constrain the *Variable* to have a *value* at an index or mask location specified by *where*.

```
>>> v = Variable((0, 1, 2, 3))
>>> v.constrain(2, numerix.array((True, False, False, False)))
>>> print(v)
[2 1 2 3]
>>> v[:] = 10
>>> print(v)
[ 2 10 10 10]
>>> v.constrain(5, numerix.array((False, False, True, False)))
>>> print(v)
[ 2 10  5 10]
>>> v[:] = 6
>>> print(v)
[2 6 5 6]
>>> v.constrain(8)
>>> print(v)
[8 8 8 8]
```

(continues on next page)

(continued from previous page)

```
>>> v[:] = 10
>>> print(v)
[8 8 8 8]
>>> del v.constraints[2]
>>> print(v)
[ 2 10  5 10]
```

```
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.meshes import Grid2D
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v = CellVariable(mesh=m, rank=1, value=(x, y))
>>> v.constrain(((0.,), (-1.,)), where=m.facesLeft)
>>> print(v.faceValue)
[[ 0.5  1.5  0.5  1.5  0.5  1.5  0.   1.   1.5  0.   1.   1.5]
 [ 0.5  0.5  1.   1.   1.5  1.5 -1.   0.5  0.5 -1.   1.5  1.5]]
```

Parameters

- **value** (float or *array_like*) – The value of the constraint
- **where** (*array_like* of *bool*) – The constraint mask or index specifying the location of the constraint

property constraints

copy()

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0, 1, 2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

dontCacheMe (*recursive=False*)

dot (*other, opShape=None, operatorClass=None, axis=0*)

getscdtype (*default=None*)

Returns the Numpy *scdtype* of the underlying array.

```
>>> Variable(1).getscdtype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getscdtype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getscdtype() == numerix.NUMERIX.obj2sctype(numerix.
↪array((1., 1.)))
True
```

inBaseUnits()

Return the value of the *Variable* with all units reduced to their base SI elements.

```
>>> e = Variable(value="2.7 Hartree*Nav")
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

inUnitsOf(*units)

Returns one or more *Variable* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *Variable*.

```
>>> freeze = Variable('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *Variable* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = Variable(value=314159., unit='s')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d',
↪'h', 'min', 's'),
...                                                         ['3.0 d', '15.0 h
↪', '15.0 min', '59.0 s'])],
...                        True))
1
```

itemset (value)**property itemsize****property mag****max (axis=None)****min (axis=None)****property name****property numericValue****put (indices, value)****ravel()****release (constraint)**

Remove *constraint* from *self*


```

>>> v = Variable((0, 1, 2, 3))
>>> v.constrain(2, numerix.array((True, False, False, False)))
>>> v[:] = 10
>>> from fipy.boundaryConditions.constraint import Constraint
>>> c1 = Constraint(5, numerix.array((False, False, True, False)))
>>> v.constrain(c1)
>>> v[:] = 6
>>> v.constrain(8)
>>> v[:] = 10
>>> del v.constraints[2]
>>> v.release(constraint=c1)
>>> print(v)
[ 2 10 10 10]

```

setValue (*value*, *unit=None*, *where=None*)

Set the value of the Variable. Can take a masked array.

```

>>> a = Variable((1, 2, 3))
>>> a.setValue(5, where=(1, 0, 1))
>>> print(a)
[5 2 5]

```

```

>>> b = Variable((4, 5, 6))
>>> a.setValue(b, where=(1, 0, 1))
>>> print(a)
[4 2 6]
>>> print(b)
[4 5 6]
>>> a.value = 3
>>> print(a)
[3 3 3]

```

```

>>> b = numerix.array((3, 4, 5))
>>> a.value = b
>>> a[:] = 1
>>> print(b)
[3 4 5]

```

```

>>> a.setValue((4, 5, 6), where=(1, 0))
Traceback (most recent call last):
....
ValueError: shape mismatch: objects cannot be broadcast to a single shape

```

property shape

Tuple of array dimensions.

```

>>> Variable(value=3).shape
()
>>> numerix.allegal(Variable(value=(3,)).shape, (1,))
True
>>> numerix.allegal(Variable(value=(3, 4)).shape, (2,))
True

```

```

>>> Variable(value="3 m").shape
()
>>> numerix.allegal(Variable(value=(3,), unit="m").shape, (1,))

```

(continues on next page)

(continued from previous page)

```
True
>>> numerix.allegal(Variable(value=(3, 4), unit="m").shape, (2,))
True
```

std (*axis=None*, ***kwargs*)

property **subscribedVariables**

sum (*axis=None*)

take (*ids*, *axis=0*)

tostring (*max_line_width=75*, *precision=8*, *suppress_small=False*, *separator=' '*)

property **unit**

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

property **value**

“Evaluate” the *Variable* and return its value (longhand)

```
>>> a = Variable(value=3)
>>> print(a.value)
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b.value
7
```

36.41 Module contents

class `fipy.variables.Variable` (*value=0.0*, *unit=None*, *array=None*, *name=""*, *cached=1*)

Bases: `object`

Lazily evaluated quantity with units.

Using a *Variable* in a mathematical expression will create an automatic dependency *Variable*, e.g.,

```
>>> a = Variable(value=3)
>>> b = 4 * a
>>> b
(Variable(value=array(3)) * 4)
>>> b()
12
```

Changes to the value of a *Variable* will automatically trigger changes in any dependent *Variable* objects

```
>>> a.setValue(5)
>>> b
(Variable(value=array(5)) * 4)
>>> print(b())
20
```

Create a *Variable*.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3, 'm'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3, 2), '1'))
Variable(value=PhysicalField(array([[3, 3],
                                   [3, 3],
                                   [3, 3]]), 'm'))
```

Parameters

- **value** (int or float or *array_like*) –
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable
- **array** (*ndarray*, *optional*) – The storage array for the *Variable*
- **name** (*str*) – The user-readable name of the *Variable*
- **cached** (*bool*) – whether to cache or always recalculate the value

`__abs__()`

Following test it to fix a bug with C inline string using *abs()* instead of *fabs()*

```
>>> print(abs(Variable(2.3) - Variable(1.2)))
1.1
```

`__add__(other)`

`__and__(other)`

This test case has been added due to a weird bug that was appearing.

```
>>> a = Variable(value=(0, 0, 1, 1))
>>> b = Variable(value=(0, 1, 0, 1))
>>> print(numerix.equal((a == 0) & (b == 1), [False, True, False, False]).
↪all())
True
>>> print(a & b)
[0 0 0 1]
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx=4)
>>> from fipy.variables.cellVariable import CellVariable
>>> a = CellVariable(value=(0, 0, 1, 1), mesh=mesh)
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> print(numerix.allegal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]
```

`__array__(t=None)`

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

__array_priority__ = 100.0

__array_wrap__ (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples of ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

__bool__ ()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is
↪ambiguous. Use a.any() or a.all()
```

__call__ ()

“Evaluate” the *Variable* and return its value

```
>>> a = Variable(value=3)
>>> print(a())
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b()
7
```

__dict__ = mappingproxy({'__module__': 'fipy.variables.variable', '__doc__': '\n Laz

__div__ (*other*)

__eq__ (*other*)

Test if a *Variable* is equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a == 4)
>>> b
(Variable(value=array(3)) == 4)
>>> b()
0
```

__float__ ()

__ge__ (*other*)

Test if a *Variable* is greater than or equal to another quantity

```

>>> a = Variable(value=3)
>>> b = (a >= 4)
>>> b
(Variable(value=array(3)) >= 4)
>>> b()
0
>>> a.value = 4
>>> print(b())
1
>>> a.value = 5
>>> print(b())
1

```

`__getitem__(index)`

“Evaluate” the *Variable* and return the specified element

```

>>> a = Variable(value=((3., 4.), (5., 6.)), unit="m") + "4 m"
>>> print(a[1, 1])
10.0 m

```

It is an error to slice a *Variable* whose *value* is not sliceable

```

>>> Variable(value=3)[2]
Traceback (most recent call last):
...
IndexError: 0-d arrays can't be indexed

```

`__getstate__()`

Used internally to collect the necessary information to pickle the *Variable* to persistent storage.

`__gt__(other)`

Test if a *Variable* is greater than another quantity

```

>>> a = Variable(value=3)
>>> b = (a > 4)
>>> b
(Variable(value=array(3)) > 4)
>>> print(b())
0
>>> a.value = 5
>>> print(b())
1

```

`__hash__()`

Return hash(self).

`__init__(value=0.0, unit=None, array=None, name="", cached=1)`

Create a *Variable*.

```

>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3, 'm'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3, 2), '1'))
Variable(value=PhysicalField(array([[3, 3],
[3, 3],
[3, 3]]), 'm'))

```

Parameters

- **value** (int or float or *array_like*) –
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable
- **array** (*ndarray*, *optional*) – The storage array for the *Variable*
- **name** (*str*) – The user-readable name of the *Variable*
- **cached** (*bool*) – whether to cache or always recalculate the value

__int__()**__invert__**()Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__iter__()**__le__** (*other*)Test if a *Variable* is less than or equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a <= 4)
>>> b
(Variable(value=array(3)) <= 4)
>>> b()
1
>>> a.value = 4
>>> print(b())
1
>>> a.value = 5
>>> print(b())
0
```

__len__()**__lt__** (*other*)Test if a *Variable* is less than another quantity

```
>>> a = Variable(value=3)
>>> b = (a < 4)
>>> b
(Variable(value=array(3)) < 4)
>>> b()
1
>>> a.value = 4
>>> print(b())
0
>>> print(1000000000000000000 * Variable(1) < 1.)
0
>>> print(1000 * Variable(1) < 1.)
0
```

Python automatically reverses the arguments when necessary

```
>>> 4 > Variable(value=3)
(Variable(value=array(3)) < 4)
```

__mod__ (*other*)

__module__ = 'fipy.variables.variable'

__mul__ (*other*)

__ne__ (*other*)

Test if a *Variable* is not equal to another quantity

```
>>> a = Variable(value=3)
>>> b = (a != 4)
>>> b
(Variable(value=array(3)) != 4)
>>> b()
1
```

__neg__ ()

static __new__ (*cls, *args, **kwds*)

Create and return a new object. See help(type) for accurate signature.

__nonzero__ ()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is
↪ambiguous. Use a.any() or a.all()
```

__or__ (*other*)

This test case has been added due to a weird bug that was appearing.

```
>>> a = Variable(value=(0, 0, 1, 1))
>>> b = Variable(value=(0, 1, 0, 1))
>>> print( numerix.equal((a == 0) | (b == 1), [True, True, False, True]) )
↪all()
True
>>> print(a | b)
[0 1 1 1]
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx=4)
>>> from fipy.variables.cellVariable import CellVariable
>>> a = CellVariable(value=(0, 0, 1, 1), mesh=mesh)
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> print( numerix.allegal((a == 0) | (b == 1), [True, True, False, True]) )
True
>>> print(a | b)
[0 1 1 1]
```

__pos__ ()

__pow__ (*other*)

return self**other, or self raised to power other

```
>>> print(Variable(1, "mol/l")**3)
1.0 mol**3/l**3
>>> print((Variable(1, "mol/l")**3).unit)
<PhysicalUnit mol**3/l**3>
```

__radd__ (*other*)

__rdiv__ (*other*)

__repr__ ()

Return repr(self).

__rmul__ (*other*)

__rpow__ (*other*)

__rsub__ (*other*)

__rtruediv__ (*other*)

__setitem__ (*index, value*)

__setstate__ (*dict*)

Used internally to create a new *Variable* from pickled persistent storage.

__str__ ()

Return str(self).

__sub__ (*other*)

__truediv__ (*other*)

__weakref__

list of weak references to the object (if defined)

all (*axis=None*)

```
>>> print(Variable(value=(0, 0, 1, 1)).all())
0
>>> print(Variable(value=(1, 1, 1, 1)).all())
1
```

allclose (*other, rtol=1e-05, atol=1e-08*)

```
>>> var = Variable((1, 1))
>>> print(var.allclose((1, 1)))
1
>>> print(var.allclose((1,)))
1
```

The following test is to check that the system does not run out of memory.

```
>>> from fipy.tools import numerix
>>> var = Variable(numerix.ones(10000))
>>> print(var.allclose(numerix.zeros(10000, 'l')))
False
```

allequal (*other*)

any (*axis=None*)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

cacheMe (*recursive=False*)

constrain (*value, where=None*)

Constrain the *Variable* to have a *value* at an index or mask location specified by *where*.

```
>>> v = Variable((0, 1, 2, 3))
>>> v.constrain(2, numerix.array((True, False, False, False)))
>>> print(v)
[2 1 2 3]
>>> v[:] = 10
>>> print(v)
[ 2 10 10 10]
>>> v.constrain(5, numerix.array((False, False, True, False)))
>>> print(v)
[ 2 10  5 10]
>>> v[:] = 6
>>> print(v)
[2 6 5 6]
>>> v.constrain(8)
>>> print(v)
[8 8 8 8]
>>> v[:] = 10
>>> print(v)
[8 8 8 8]
>>> del v.constraints[2]
>>> print(v)
[ 2 10  5 10]
```

```
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.meshes import Grid2D
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v = CellVariable(mesh=m, rank=1, value=(x, y))
>>> v.constrain(((0.,), (-1.,)), where=m.facesLeft)
>>> print(v.faceValue)
[[ 0.5  1.5  0.5  1.5  0.5  1.5  0.   1.   1.5  0.   1.   1.5]
 [ 0.5  0.5  1.   1.   1.5  1.5 -1.   0.5  0.5 -1.   1.5  1.5]]
```

Parameters

- **value** (float or *array_like*) – The value of the constraint
- **where** (*array_like* of `bool`) – The constraint mask or index specifying the location of the constraint

property constraints

copy ()

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0, 1, 2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

dontCacheMe (*recursive=False*)

dot (*other, opShape=None, operatorClass=None, axis=0*)

getsctype (*default=None*)

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↪array((1., 1.)))
True
```

inBaseUnits ()

Return the value of the *Variable* with all units reduced to their base SI elements.

```
>>> e = Variable(value="2.7 Hartree*Nav")
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

inUnitsOf (**units*)

Returns one or more *Variable* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *Variable*.

```
>>> freeze = Variable('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *Variable* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```

>>> t = Variable(value=314159., unit='s')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d',
↳ 'h', 'min', 's')),
...                                     ['3.0 d', '15.0 h
↳ ', '15.0 min', '59.0 s']]),
...                                     True))
1

```

itemset (*value*)

property **itemsizes**

property **mag**

max (*axis=None*)

min (*axis=None*)

property **name**

property **numericValue**

put (*indices, value*)

ravel ()

release (*constraint*)

Remove *constraint* from *self*

```

>>> v = Variable((0, 1, 2, 3))
>>> v.constrain(2, numerix.array((True, False, False, False)))
>>> v[:] = 10
>>> from fipy.boundaryConditions.constraint import Constraint
>>> c1 = Constraint(5, numerix.array((False, False, True, False)))
>>> v.constrain(c1)
>>> v[:] = 6
>>> v.constrain(8)
>>> v[:] = 10
>>> del v.constraints[2]
>>> v.release(constraint=c1)
>>> print(v)
[ 2 10 10 10]

```

setValue (*value, unit=None, where=None*)

Set the value of the Variable. Can take a masked array.

```

>>> a = Variable((1, 2, 3))
>>> a.setValue(5, where=(1, 0, 1))
>>> print(a)
[5 2 5]

```

```

>>> b = Variable((4, 5, 6))
>>> a.setValue(b, where=(1, 0, 1))
>>> print(a)
[4 2 6]
>>> print(b)
[4 5 6]
>>> a.value = 3

```

(continues on next page)

(continued from previous page)

```
>>> print(a)
[3 3 3]
```

```
>>> b = numerix.array((3, 4, 5))
>>> a.value = b
>>> a[:] = 1
>>> print(b)
[3 4 5]
```

```
>>> a.setValue((4, 5, 6), where=(1, 0))
Traceback (most recent call last):
....
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

property shape

Tuple of array dimensions.

```
>>> Variable(value=3).shape
()
>>> numerix.allegal(Variable(value=(3,)).shape, (1,))
True
>>> numerix.allegal(Variable(value=(3, 4)).shape, (2,))
True
```

```
>>> Variable(value="3 m").shape
()
>>> numerix.allegal(Variable(value=(3,), unit="m").shape, (1,))
True
>>> numerix.allegal(Variable(value=(3, 4), unit="m").shape, (2,))
True
```

std (*axis=None, **kwargs*)

property subscribedVariables

sum (*axis=None*)

take (*ids, axis=0*)

tostring (*max_line_width=75, precision=8, suppress_small=False, separator=' '*)

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

property value

“Evaluate” the *Variable* and return its value (longhand)

```
>>> a = Variable(value=3)
>>> print(a.value)
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
```

(continues on next page)

(continued from previous page)

```
>>> b.value
7
```

```
class fipy.variables.CellVariable(mesh, name="", value=0.0, rank=None, elementsshape=None, unit=None, hasOld=0)
```

Bases: `fipy.variables.meshVariable._MeshVariable`

Represents the field of values of a variable on a *Mesh*.

A *CellVariable* can be pickled to persistent storage (disk) for later use:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 10, ny = 10)
```

```
>>> var = CellVariable(mesh = mesh, value = 1., hasOld = 1, name = 'test')
>>> x, y = mesh.cellCenters
>>> var.value = (x * y)
```

```
>>> from fipy.tools import dump
>>> (f, filename) = dump.write(var, extension = '.gz')
>>> unPickledVar = dump.read(filename, f)
```

```
>>> print(var.allclose(unPickledVar, atol = 1e-10, rtol = 1e-10))
1
```

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float* or *array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementsshape** (*tuple* of *int*) – the shape of each element of this variable Default: `rank * (mesh.dim,)`
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable

__call__ (*points=None, order=0, nearestCellIDs=None*)

Interpolates the *CellVariable* to a set of points using a method that has a memory requirement on the order of *Ncells* by *Npoints* in general, but uses only *Ncells* when the *CellVariable*'s mesh is a *UniformGrid* object.

Tests

```
>>> from fipy import *
>>> m = Grid2D(nx=3, ny=2)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> print(v(((0., 1.1, 1.2), (0., 1., 1.))))
[ 0.5  1.5  1.5]
>>> print(v(((0., 1.1, 1.2), (0., 1., 1.)), order=1))
[ 0.25  1.1  1.2 ]
>>> m0 = Grid2D(nx=2, ny=2, dx=1., dy=1.)
>>> m1 = Grid2D(nx=4, ny=4, dx=.5, dy=.5)
```

(continues on next page)

(continued from previous page)

```

>>> x, y = m0.cellCenters
>>> v0 = CellVariable(mesh=m0, value=x * y)
>>> print(v0(m1.cellCenters.globalValue))
[ 0.25  0.25  0.75  0.75  0.25  0.25  0.75  0.75  0.75  0.75  2.25  2.25
  0.75  0.75  2.25  2.25]
>>> print(v0(m1.cellCenters.globalValue, order=1))
[ 0.125  0.25  0.5  0.625  0.25  0.375  0.875  1.  0.5  0.875
  1.875  2.25  0.625  1.  2.25  2.625]

```

Parameters

- **points** (tuple or list of tuple) – A point or set of points in the format (X, Y, Z)
- **order** ({0, 1}) – The order of interpolation, default is 0
- **nearestCellIDs** (array_like) – Optional argument if user can calculate own nearest cell IDs array, shape should be same as points

`__getstate__()`

Used internally to collect the necessary information to pickle the *CellVariable* to persistent storage.

`__init__(mesh, name="", value=0.0, rank=None, elementshape=None, unit=None, hasOld=0)`

Parameters

- **mesh** (Mesh) – the mesh that defines the geometry of this *Variable*
- **name** (str) – the user-readable name of the *Variable*
- **value** (float or array_like) – the initial value
- **rank** (int) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (tuple of int) – the shape of each element of this variable Default: $\text{rank} * (\text{mesh.dim},)$
- **unit** (str or PhysicalUnit) – The physical units of the variable

`__module__ = 'fipy.variables.cellVariable'`

`__setstate__(dict)`

Used internally to create a new *CellVariable* from pickled persistent storage.

property arithmeticFaceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```

>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

property cellVolumeAverage

Return the cell-volume-weighted average of the *CellVariable*:

$$\langle \phi \rangle_{\text{vol}} = \frac{\sum_{\text{cells}} \phi_{\text{cell}} V_{\text{cell}}}{\sum_{\text{cells}} V_{\text{cell}}}$$

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .1)
>>> var = CellVariable(value = (1, 2, 6), mesh = mesh)
>>> print(var.cellVolumeAverage)
3.0
```

constrain (value, where=None)

Constrains the *CellVariable* to *value* at a location specified by *where*.

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> v.constrain(0., where=m.facesLeft)
>>> v.faceGrad.constrain([1.], where=m.facesRight)
>>> print(v.faceGrad)
[[ 1.  1.  1.  1.]]
>>> print(v.faceValue)
[ 0.  1.  2.  2.5]
```

Changing the constraint changes the dependencies

```
>>> v.constrain(1., where=m.facesLeft)
>>> print(v.faceGrad)
[[-1.  1.  1.  1.]]
>>> print(v.faceValue)
[ 1.  1.  2.  2.5]
```

Constraints can be *Variable*

```
>>> c = Variable(0.)
>>> v.constrain(c, where=m.facesLeft)
>>> print(v.faceGrad)
[[ 1.  1.  1.  1.]]
>>> print(v.faceValue)
[ 0.  1.  2.  2.5]
```

(continues on next page)

(continued from previous page)

```
>>> c.value = 1.
>>> print(v.faceGrad)
[[-1.  1.  1.  1.]]
>>> print(v.faceValue)
[ 1.  1.  2.  2.5]
```

Constraints can have a *Variable* mask.

```
>>> v = CellVariable(mesh=m)
>>> mask = FaceVariable(mesh=m, value=m.facesLeft)
>>> v.constrain(1., where=mask)
>>> print(v.faceValue)
[ 1.  0.  0.  0.]
>>> mask[:] = mask | m.facesRight
>>> print(v.faceValue)
[ 1.  0.  0.  1.]
```

copy()

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0, 1, 2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

property faceGradAverage

Deprecated since version 3.3: use `grad.arithmeticFaceValue()` instead

Return $\nabla\phi$ as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

property faceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$


```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

property gaussGrad

Return $\frac{1}{V_P} \sum_f \vec{n}_f A_f$ as a rank-1 *CellVariable* (first-order gradient).

property globalValue

Concatenate and return values from all processors

When running on a single processor, the result is identical to *value*.

property grad

Return $\nabla\phi$ as a rank-1 *CellVariable* (first-order gradient).

property harmonicFaceValue

Returns a *FaceVariable* whose value corresponds to the harmonic interpolation of the adjacent cells:

$$\phi_f = \frac{\phi_1 \phi_2}{(\phi_2 - \phi_1) \frac{d_{f2}}{d_{12}} + \phi_1}$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (0.5 / 1.) + L)
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
```

(continues on next page)

(continued from previous page)

```
>>> answer = L * R / ((R - L) * (1.0 / 3.0) + L)
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.harmonicFaceValue[mesh.interiorFaces.value]
>>> answer = L * R / ((R - L) * (5.0 / 55.0) + L)
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

property leastSquaresGrad

Return $\nabla\phi$, which is determined by solving for $\nabla\phi$ in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n} \cdot \nabla\phi)_{AP}$$

The matrix equation is derived by minimizing the following least squares sum,

$$F(\phi_x, \phi_y) = \sqrt{\sum_f (d_{AP} \vec{n}_{AP} \cdot \nabla\phi - d_{AP} (\vec{n}_{AP} \cdot \nabla\phi)_{AP})^2}$$

Tests

```
>>> from fipy import Grid2D
>>> m = Grid2D(nx=2, ny=2, dx=0.1, dy=2.0)
>>> print(numerix.allclose(CellVariable(mesh=m, value=(0, 1, 3, 6)).
↪ leastSquaresGrad.globalValue, \
...                                     [[8.0, 8.0, 24.0, 24.0],
...                                     [1.2, 2.0, 1.2, 2.0]]))
True
```

```
>>> from fipy import Grid1D
>>> print(numerix.allclose(CellVariable(mesh=Grid1D(dx=(2.0, 1.0, 0.5)),
...                                     value=(0, 1, 2)).leastSquaresGrad.
↪ globalValue, [[0.461538461538, 0.8, 1.2]]))
True
```

property minmodFaceValue

Returns a *FaceVariable* with a value that is the minimum of the absolute values of the adjacent cells. If the values are of opposite sign then the result is zero:

$$\phi_f = \begin{cases} \phi_1 & \text{when } |\phi_1| \leq |\phi_2|, \\ \phi_2 & \text{when } |\phi_2| < |\phi_1|, \\ 0 & \text{when } \phi_1 \phi_2 < 0 \end{cases}$$

```
>>> from fipy import *
>>> print(CellVariable(mesh=Grid1D(nx=2), value=(1, 2)).minmodFaceValue)
[1 1 2]
>>> print(CellVariable(mesh=Grid1D(nx=2), value=(-1, -2)).minmodFaceValue)
[-1 -1 -2]
>>> print(CellVariable(mesh=Grid1D(nx=2), value=(-1, 2)).minmodFaceValue)
[-1 0 2]
```

property old

Return the values of the *CellVariable* from the previous solution sweep.

Combinations of *CellVariable*'s should also return old values.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> var1 = CellVariable(mesh = mesh, value = (2, 3), hasOld = 1)
>>> var2 = CellVariable(mesh = mesh, value = (3, 4))
>>> v = var1 * var2
>>> print(v)
[ 6 12]
>>> var1.value = ((3, 2))
>>> print(v)
[9 8]
>>> print(v.old)
[ 6 12]
```

The following small test is to correct for a bug when the operator does not just use variables.

```
>>> v1 = var1 * 3
>>> print(v1)
[9 6]
>>> print(v1.old)
[6 9]
```

release (constraint)

Remove *constraint* from *self*

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> c = Constraint(0., where=m.facesLeft)
>>> v.constrain(c)
>>> print(v.faceValue)
[ 0.  1.  2.  2.5]
>>> v.release(constraint=c)
>>> print(v.faceValue)
[ 0.5  1.  2.  2.5]
```

setValue (value, unit=None, where=None)

Set the value of the Variable. Can take a masked array.

```
>>> a = Variable((1, 2, 3))
>>> a.setValue(5, where=(1, 0, 1))
>>> print(a)
[5 2 5]
```

```
>>> b = Variable((4, 5, 6))
>>> a.setValue(b, where=(1, 0, 1))
>>> print(a)
[4 2 6]
>>> print(b)
[4 5 6]
>>> a.value = 3
>>> print(a)
[3 3 3]
```

```
>>> b = numerix.array((3, 4, 5))
>>> a.value = b
>>> a[:] = 1
>>> print(b)
[3 4 5]
```

```
>>> a.setValue((4, 5, 6), where=(1, 0))
Traceback (most recent call last):
....
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

updateOld()

Set the values of the previous solution sweep to the current values.

```
>>> from fipy import *
>>> v = CellVariable(mesh=Grid1D(), hasOld=False)
>>> v.updateOld()
Traceback (most recent call last):
....
AssertionError: The updateOld method requires the CellVariable to have an old_
↳value. Set hasOld to True when instantiating the CellVariable.
```

```
class fipy.variables.FaceVariable(mesh, name="", value=0.0, rank=None, ele-
                                mentshape=None, unit=None, cached=1)
Bases: fipy.variables.meshVariable._MeshVariable
```

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float* or *array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementsshape** (*tuple* of *int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable

```
__module__ = 'fipy.variables.faceVariable'
```

copy()

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0, 1, 2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

property divergence

the divergence of *self*, \vec{u} ,

$$\nabla \cdot \vec{u} \approx \frac{\sum_f (\vec{u} \cdot \hat{n})_f A_f}{V_P}$$

Returns **divergence** – one rank lower than *self*

Return type *fipy.variables.cellVariable.CellVariable*

Examples

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=3, ny=2)
>>> from builtins import range
>>> var = CellVariable(mesh=mesh, value=list(range(3*2)))
>>> print(var.faceGrad.divergence)
[ 4.  3.  2. -2. -3. -4.]
```

property globalValue

setValue (*value*, *unit=None*, *where=None*)

Set the value of the Variable. Can take a masked array.

```
>>> a = Variable((1, 2, 3))
>>> a.setValue(5, where=(1, 0, 1))
>>> print(a)
[5 2 5]
```

```
>>> b = Variable((4, 5, 6))
>>> a.setValue(b, where=(1, 0, 1))
>>> print(a)
[4 2 6]
>>> print(b)
[4 5 6]
>>> a.value = 3
>>> print(a)
[3 3 3]
```

```
>>> b = numerix.array((3, 4, 5))
>>> a.value = b
>>> a[:] = 1
>>> print(b)
[3 4 5]
```

```
>>> a.setValue((4, 5, 6), where=(1, 0))
Traceback (most recent call last):
....
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

```
class fipy.variables.ScharfetterGummelFaceVariable (var, boundaryConditions=())
    Bases: fipy.variables.cellToFaceVariable._CellToFaceVariable
```

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float* or *array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple* of *int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable

```
__init__ (var, boundaryConditions=())
```

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float* or *array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple* of *int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable

```
__module__ = 'fipy.variables.scharfetterGummelFaceVariable'
```

```
class fipy.variables.ModularVariable (mesh, name="", value=0.0, rank=None, ele-
                                     mentshape=None, unit=None, hasOld=0)
    Bases: fipy.variables.cellVariable.CellVariable
```

The *ModularVariable* defines a variable that exists on the circle between $-\pi$ and π

The following examples show how *ModularVariable* works. When subtracting the answer wraps back around the circle.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 2)
>>> from fipy.tools import numerix
>>> pi = numerix.pi
>>> v1 = ModularVariable(mesh = mesh, value = (2*pi/3, -2*pi/3))
>>> v2 = ModularVariable(mesh = mesh, value = -2*pi/3)
>>> print(numerix.allclose(v2 - v1, (2*pi/3, 0)))
1
```

Obtaining the arithmetic face value.

```
>>> print (numerix.allclose(v1.arithmeticFaceValue, (2*pi/3, pi, -2*pi/3)))
1
```

Obtaining the gradient.

```
>>> print (numerix.allclose(v1.grad, ((pi/3, pi/3),)))
1
```

Obtaining the gradient at the faces.

```
>>> print (numerix.allclose(v1.faceGrad, ((0, 2*pi/3, 0),)))
1
```

Obtaining the gradient at the faces but without modular arithmetic.

```
>>> print (numerix.allclose(v1.faceGradNoMod, ((0, -4*pi/3, 0),)))
1
```

Parameters

- **mesh** (*Mesh*) – the mesh that defines the geometry of this *Variable*
- **name** (*str*) – the user-readable name of the *Variable*
- **value** (*float* or *array_like*) – the initial value
- **rank** (*int*) – the rank (number of dimensions) of each element of this *Variable*. Default: 0
- **elementshape** (*tuple* of *int*) – the shape of each element of this variable Default: *rank * (mesh.dim,)*
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable

```
__module__ = 'fipy.variables.modularVariable'
```

```
__rsub__ (other)
```

```
__sub__ (other)
```

property arithmeticFaceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

Adjusted for a *ModularVariable*

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* (second-order gradient). Adjusted for a *ModularVariable*

property faceGradNoMod

Return $\nabla\phi$ as a rank-1 *FaceVariable* (second-order gradient). Not adjusted for a *ModularVariable*

property grad

Return $\nabla\phi$ as a rank-1 *CellVariable* (first-order gradient). Adjusted for a *ModularVariable*

updateOld()

Set the values of the previous solution sweep to the current values. Test case due to bug.

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> var = ModularVariable(mesh=mesh, value=1., hasOld=1)
>>> var.updateOld()
>>> var[:] = 2
>>> answer = CellVariable(mesh=mesh, value=1.)
>>> print(var.old.allclose(answer))
True
```

class fipy.variables.BetaNoiseVariable (mesh, alpha, beta, name="", hasOld=0)

Bases: *fipy.variables.noiseVariable.NoiseVariable*

Represents a beta distribution of random numbers with the probability distribution

$$x^{\alpha-1} \frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α , a rate parameter β , and $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform Cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = BetaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), alpha = alpha,
↳ beta = beta)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.01, nx = 100)
```

and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> betadist = CellVariable(mesh = histogram.mesh)
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> betadist = ((Gamma(alpha + beta) / (Gamma(alpha) * Gamma(beta)))
...             * x**(alpha - 1) * (1 - x)**(beta - 1))
```

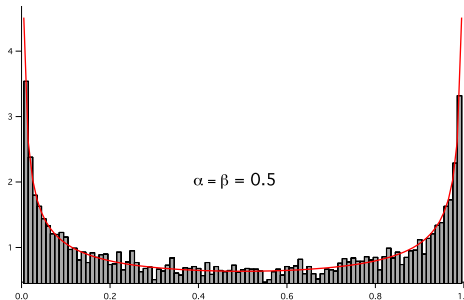
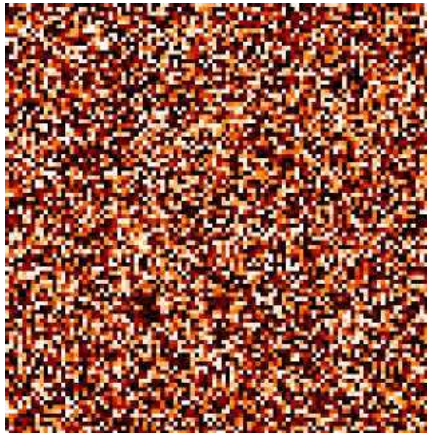
```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=1)
...     histplot = Viewer(vars=(histogram, betadist),
...                          datamin=0, datamax=1.5)
```

```
>>> from fipy.tools.numerix import arange
```



```
>>> for a in arange(0.5, 5, 0.5):
...     alpha.value = a
...     for b in arange(0.5, 5, 0.5):
...         beta.value = b
...         if __name__ == '__main__':
...             import sys
...             print("alpha: %g, beta: %g" % (alpha, beta), file=sys.stderr)
...             viewer.plot()
...             histplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```



Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **alpha** (*float*) – The parameter α .
- **beta** (*float*) – The parameter β .

`__init__(mesh, alpha, beta, name="", hasOld=0)`

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **alpha** (*float*) – The parameter α .
- **beta** (*float*) – The parameter β .

`__module__ = 'fipy.variables.betaNoiseVariable'`

`random()`

class fipy.variables.**ExponentialNoiseVariable** (*mesh, mean=0.0, name="", hasOld=0*)

Bases: *fipy.variables.noiseVariable.NoiseVariable*

Represents an exponential distribution of random numbers with the probability distribution

$$\mu^{-1}e^{-\frac{x}{\mu}}$$

with a mean parameter μ .

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform Cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> mean = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = ExponentialNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), mean =
↳mean)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 100)
```

and compare to a Gaussian distribution

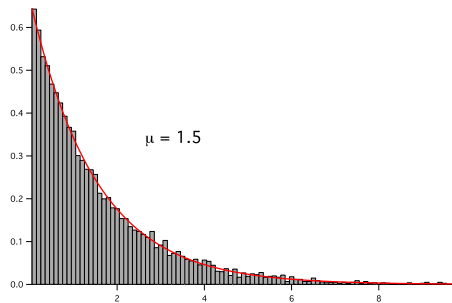
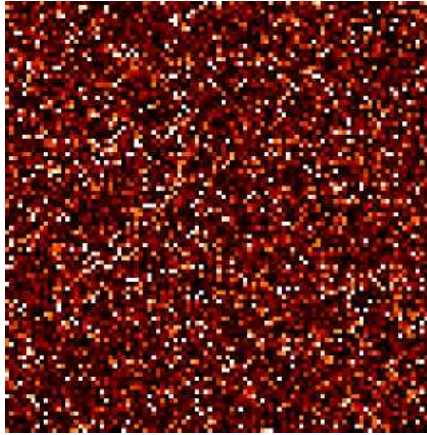
```
>>> from fipy.variables.cellVariable import CellVariable
>>> expdist = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=5)
...     histoplot = Viewer(vars=(histogram, expdist),
...                           datamin=0, datamax=1.5)
```

```
>>> from fipy.tools.numerix import arange, exp
```

```
>>> for mu in arange(0.5, 3, 0.5):
...     mean.value = (mu)
...     expdist.value = ((1/mean)*exp(-x/mean))
...     if __name__ == '__main__':
...         import sys
...         print("mean: %g" % mean, file=sys.stderr)
...         viewer.plot()
...         histoplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```



Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **mean** (*float*) – The mean of the distribution μ .

`__init__(mesh, mean=0.0, name="", hasOld=0)`

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **mean** (*float*) – The mean of the distribution μ .

`__module__ = 'fipy.variables.exponentialNoiseVariable'`

`random()`

class `fipy.variables.GammaNoiseVariable` (*mesh, shape, rate, name="", hasOld=0*)

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a gamma distribution of random numbers with the probability distribution

$$x^{\alpha-1} \frac{\beta^\alpha e^{-\beta x}}{\Gamma(\alpha)}$$

with a shape parameter α , a rate parameter β , and $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$.

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform Cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = GammaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), shape = alpha,
↳ rate = beta)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 300)
```

and compare to a Gaussian distribution

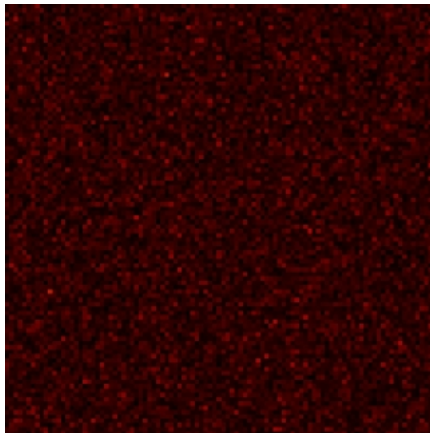
```
>>> from fipy.variables.cellVariable import CellVariable
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> from fipy.tools.numerix import exp
>>> gammadist = (x**(alpha - 1) * (beta**alpha * exp(-beta * x)) / Gamma(alpha))
```

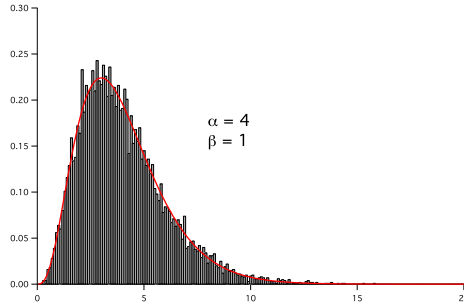
```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=30)
...     histoplot = Viewer(vars=(histogram, gammadist),
...                           datamin=0, datamax=1)
```

```
>>> from fipy.tools.numerix import arange
```

```
>>> for shape in arange(1, 8, 1):
...     alpha.value = shape
...     for rate in arange(0.5, 2.5, 0.5):
...         beta.value = rate
...         if __name__ == '__main__':
...             import sys
...             print("alpha: %g, beta: %g" % (alpha, beta), file=sys.stderr)
...             viewer.plot()
...             histoplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```





Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **shape** (*float*) – The shape parameter, α .
- **rate** (*float*) – The rate or inverse scale parameter, β .

`__init__(mesh, shape, rate, name="", hasOld=0)`

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **shape** (*float*) – The shape parameter, α .
- **rate** (*float*) – The rate or inverse scale parameter, β .

`__module__ = 'fipy.variables.gammaNoiseVariable'`

`random()`

`class fipy.variables.GaussianNoiseVariable (mesh, name="", mean=0.0, variance=1.0, hasOld=0)`

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a normal (Gaussian) distribution of random numbers with mean μ and variance $\langle \eta(\vec{r})\eta(\vec{r}') \rangle = \sigma^2$, which has the probability distribution

$$\frac{1}{\sigma\sqrt{2\pi}} \exp - \frac{(x - \mu)^2}{2\sigma^2}$$

For example, the variance of thermal noise that is uncorrelated in space and time is often expressed as

$$\langle \eta(\vec{r}, t) \eta(\vec{r}', t') \rangle = M k_B T \delta(\vec{r} - \vec{r}') \delta(t - t')$$

which can be obtained with:

```
sigmaSqr = Mobility * kBoltzmann * Temperature / (mesh.cellVolumes * timeStep)
GaussianNoiseVariable(mesh = mesh, variance = sigmaSqr)
```

Note: If the time step will change as the simulation progresses, either through use of an adaptive iterator or by making manual changes at different stages, remember to declare *timeStep* as a *Variable* and to change its value with its *setValue()* method.

```
>>> import sys
>>> from fipy.tools.numerix import *
```

```
>>> mean = 0.
>>> variance = 4.
```

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(3)
```

We generate noise on a non-uniform Cartesian mesh with cell dimensions of x^2 and y^3 .

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = arange(0.1, 5., 0.1)**2, dy = arange(0.1, 3., 0.1)**3)
>>> from fipy.variables.cellVariable import CellVariable
>>> volumes = CellVariable(mesh=mesh, value=mesh.cellVolumes)
>>> noise = GaussianNoiseVariable(mesh = mesh, mean = mean,
...                               variance = variance / volumes)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise * sqrt(volumes),
...                               dx = 0.1, nx = 600, offset = -30)
...                               dx = 0.1, nx = 600, offset = -30)
```

and compare to a Gaussian distribution

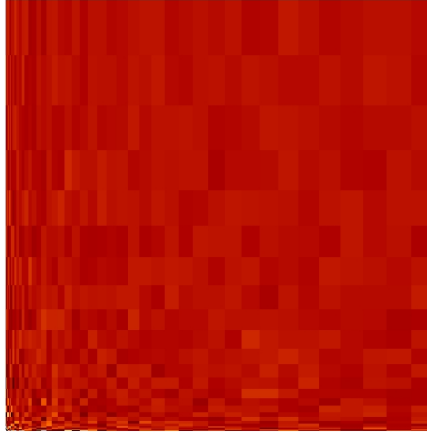
```
>>> gauss = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
>>> gauss.value = ((1/(sqrt(variance * 2 * pi))) * exp(-(x - mean)**2 / (2 *
↪variance)))
```

```
>>> if __name__ == '__main__':
...     from fipy.viewers import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=-5, datamax=5)
...     histoplot = Viewer(vars=(histogram, gauss))
```

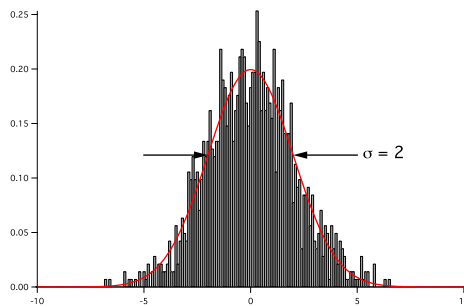
```
>>> from builtins import range
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```

Note that the noise exhibits larger amplitude in the small cells than in the large ones



but that the root-volume-weighted histogram is Gaussian.



Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **mean** (*float*) – The mean of the noise distribution, μ .
- **variance** (*float*) – The variance of the noise distribution, σ^2 .

`__init__(mesh, name="", mean=0.0, variance=1.0, hasOld=0)`

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **mean** (*float*) – The mean of the noise distribution, μ .
- **variance** (*float*) – The variance of the noise distribution, σ^2 .

`__module__ = 'fipy.variables.gaussianNoiseVariable'`

`parallelRandom()`

class `fipy.variables.UniformNoiseVariable` (*mesh*, *name*="", *minimum*=0.0, *maximum*=1.0, *hasOld*=0)

Bases: `fipy.variables.noiseVariable.NoiseVariable`

Represents a uniform distribution of random numbers.

We generate noise on a uniform Cartesian mesh

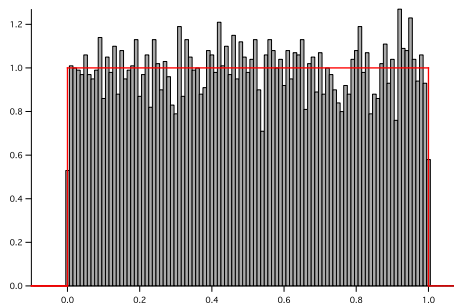
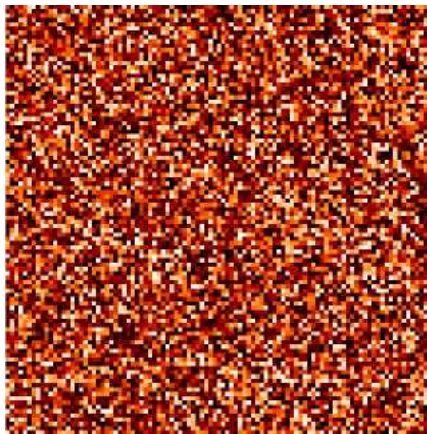
```
>>> from fipy.meshes import Grid2D
>>> noise = UniformNoiseVariable(mesh=Grid2D(nx=100, ny=100))
```

and histogram the noise

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution=noise, dx=0.01, nx=120, offset=-.1)
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=0, datamax=1)
...     histoplot = Viewer(vars=histogram)
```

```
>>> from builtins import range
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()
```



Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **minimum** (*float*) – The minimum (not-inclusive) value of the distribution.
- **maximum** (*float*) – The maximum (not-inclusive) value of the distribution.

`__init__` (*mesh*, *name*="", *minimum*=0.0, *maximum*=1.0, *hasOld*=0)

Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **minimum** (*float*) – The minimum (not-inclusive) value of the distribution.
- **maximum** (*float*) – The maximum (not-inclusive) value of the distribution.


```
__module__ = 'fipy.variables.uniformNoiseVariable'
```

```
random()
```

```
class fipy.variables.HistogramVariable (distribution, dx=1.0, nx=None, offset=0.0)
```

Bases: `fipy.variables.cellVariable.CellVariable`

Produces a histogram of the values of the supplied distribution.

Parameters

- **distribution** (*array_like* or *Variable*) – The collection of values to sample.
- **dx** (*float*) – The bin size
- **nx** (*int*) – The number of bins
- **offset** (*float*) – The position of the first bin

```
__init__ (distribution, dx=1.0, nx=None, offset=0.0)
```

Produces a histogram of the values of the supplied distribution.

Parameters

- **distribution** (*array_like* or *Variable*) – The collection of values to sample.
- **dx** (*float*) – The bin size
- **nx** (*int*) – The number of bins
- **offset** (*float*) – The position of the first bin

```
__module__ = 'fipy.variables.histogramVariable'
```

```
class fipy.variables.SurfactantVariable (value=0.0, distanceVar=None, name='surfactant
                                         variable', hasOld=False)
```

Bases: `fipy.variables.cellVariable.CellVariable`

The *SurfactantVariable* maintains a conserved volumetric concentration on cells adjacent to, but in front of, the interface. The *value* argument corresponds to the initial concentration of surfactant on the interface (moles divided by area). The value held by the *SurfactantVariable* is actually a volume density (moles divided by volume).

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> from fipy.variables.distanceVariable import DistanceVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (-1.5, -0.5, 0.5, 941.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                         distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable, (0, 0., 1., 0)))
1
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (1.5, 0.5, 1.5,
...                                           0.5, -0.5, 0.5,
...                                           1.5, 0.5, 1.5))
```

(continues on next page)

(continued from previous page)

```
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                         distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable, (0, 1, 0, 1, 0, 1, 0, 1, 0)))
1
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (-0.5, 0.5, 0.5, 1.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                         distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable,
...                         (0, numerix.sqrt(2), numerix.sqrt(2), 0)))
1
```

Parameters

- **value** (*float* or *array_like*) – The initial value.
- **distanceVar** (*DistanceVariable*) –
- **name** (*str*) – The name of the variable.

__init__ (value=0.0, distanceVar=None, name='surfactant variable', hasOld=False)

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> from fipy.variables.distanceVariable import DistanceVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (-1.5, -0.5, 0.5, 941.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                         distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable, (0, 0., 1., 0)))
1
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (1.5, 0.5, 1.5,
...                                           0.5, -0.5, 0.5,
...                                           1.5, 0.5, 1.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                         distanceVar = distanceVariable)
>>> print(numerix.allclose(surfactantVariable, (0, 1, 0, 1, 0, 1, 0, 1, 0)))
1
```

Another 2D test case:

```
>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (-0.5, 0.5, 0.5, 1.5))
>>> surfactantVariable = SurfactantVariable(value = 1,
...                                         distanceVar = distanceVariable)
```

(continues on next page)

(continued from previous page)

```
>>> print(numerix.allclose(surfactantVariable,
...                        (0, numerix.sqrt(2), numerix.sqrt(2), 0)))
1
```

Parameters

- **value** (*float or array_like*) – The initial value.
- **distanceVar** (*DistanceVariable*) –
- **name** (*str*) – The name of the variable.

```
__module__ = 'fipy.variables.surfactantVariable'
```

copy()

Make an duplicate of the *Variable*

```
>>> a = Variable(value=3)
>>> b = a.copy()
>>> b
Variable(value=array(3))
```

The duplicate will not reflect changes made to the original

```
>>> a.setValue(5)
>>> b
Variable(value=array(3))
```

Check that this works for arrays.

```
>>> a = Variable(value=numerix.array((0, 1, 2)))
>>> b = a.copy()
>>> b
Variable(value=array([0, 1, 2]))
>>> a[1] = 3
>>> b
Variable(value=array([0, 1, 2]))
```

property interfaceVar

Returns the *SurfactantVariable* rendered as an *_InterfaceSurfactantVariable* which evaluates the surfactant concentration as an area concentration the interface rather than a volumetric concentration.

class fipy.variables.SurfactantConvectionVariable(*distanceVar*)

Bases: *fipy.variables.faceVariable.FaceVariable*

Convection coefficient for the *ConservativeSurfactantEquation*. The coefficient only has a value for a negative *distanceVar*.

Simple one dimensional test:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(nx = 3, ny = 1, dx = 1., dy = 1.)
>>> from fipy.variables.distanceVariable import DistanceVariable
>>> distanceVar = DistanceVariable(mesh, value = (-.5, .5, 1.5))
>>> ## answer = numerix.zeros((2, mesh.numberOfFaces), 'd')
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[0, 7] = -1
```

(continues on next page)

(continued from previous page)

```
>>> print (numerix.allclose(SurfactantConvectionVariable(distanceVar).globalValue,
↳answer))
True
```

Change the dimensions:

```
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .25)
>>> distanceVar = DistanceVariable(mesh, value = (-.25, .25, .75))
>>> answer[0, 7] = -.5
>>> print (numerix.allclose(SurfactantConvectionVariable(distanceVar).globalValue,
↳answer))
True
```

Two dimensional example:

```
>>> mesh = Grid2D(nx = 2, ny = 2, dx = 1., dy = 1.)
>>> distanceVar = DistanceVariable(mesh, value = (-1.5, -.5, -.5, .5))
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[1, 2] = -.5
>>> answer[1, 3] = -1
>>> answer[0, 7] = -.5
>>> answer[0, 10] = -1
>>> print (numerix.allclose(SurfactantConvectionVariable(distanceVar).globalValue,
↳answer))
True
```

Larger grid:

```
>>> mesh = Grid2D(nx = 3, ny = 3, dx = 1., dy = 1.)
>>> distanceVar = DistanceVariable(mesh, value = (1.5, .5, 1.5,
...                                              .5, -.5, .5,
...                                              1.5, .5, 1.5))
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[1, 4] = .25
>>> answer[1, 7] = -.25
>>> answer[0, 17] = .25
>>> answer[0, 18] = -.25
>>> print (numerix.allclose(SurfactantConvectionVariable(distanceVar).globalValue,
↳answer))
True
```

`__init__` (*distanceVar*)

Simple one dimensional test:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(nx = 3, ny = 1, dx = 1., dy = 1.)
>>> from fipy.variables.distanceVariable import DistanceVariable
>>> distanceVar = DistanceVariable(mesh, value = (-.5, .5, 1.5))
>>> ## answer = numerix.zeros((2, mesh.numberOfFaces), 'd')
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[0, 7] = -1
>>> print (numerix.allclose(SurfactantConvectionVariable(distanceVar).
↳globalValue, answer))
True
```

Change the dimensions:

```
>>> mesh = Grid2D(nx = 3, ny = 1, dx = .5, dy = .25)
>>> distanceVar = DistanceVariable(mesh, value = (-.25, .25, .75))
>>> answer[0, 7] = -.5
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).
↪globalValue, answer))
True
```

Two dimensional example:

```
>>> mesh = Grid2D(nx = 2, ny = 2, dx = 1., dy = 1.)
>>> distanceVar = DistanceVariable(mesh, value = (-1.5, -.5, -.5, .5))
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[1, 2] = -.5
>>> answer[1, 3] = -1
>>> answer[0, 7] = -.5
>>> answer[0, 10] = -1
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).
↪globalValue, answer))
True
```

Larger grid:

```
>>> mesh = Grid2D(nx = 3, ny = 3, dx = 1., dy = 1.)
>>> distanceVar = DistanceVariable(mesh, value = (1.5, .5, 1.5,
...                                              .5, -.5, .5,
...                                              1.5, .5, 1.5))
>>> answer = FaceVariable(mesh=mesh, rank=1, value=0.).globalValue
>>> answer[1, 4] = .25
>>> answer[1, 7] = -.25
>>> answer[0, 17] = .25
>>> answer[0, 18] = -.25
>>> print(numerix.allclose(SurfactantConvectionVariable(distanceVar).
↪globalValue, answer))
True
```

```
__module__ = 'fipy.variables.surfactantConvectionVariable'
```

```
class fipy.variables.DistanceVariable (mesh, name="", value=0.0, unit=None, hasOld=0)
```

```
Bases: fipy.variables.cellVariable.CellVariable
```

A *DistanceVariable* object calculates ϕ so it satisfies,

$$|\nabla\phi| = 1$$

using the fast marching method with an initial condition defined by the zero level set. The solution can either be first or second order.

Here we will define a few test cases. Firstly a 1D test case

```
>>> from fipy.meshes import Grid1D
>>> from fipy.tools import serialComm
>>> mesh = Grid1D(dx = .5, nx = 8, communicator=serialComm)
>>> from .distanceVariable import DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = (-1., -1., -1., -1., 1., 1., 1.,
↪1.))
>>> var.calcDistanceFunction()
>>> answer = (-1.75, -1.25, -.75, -0.25, 0.25, 0.75, 1.25, 1.75)
>>> print(var.allclose(answer))
1
```

A 1D test case with very small dimensions.

```
>>> dx = 1e-10
>>> mesh = Grid1D(dx = dx, nx = 8, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., -1., -1., -1., 1., 1., 1., 1.))
>>> var.calcDistanceFunction()
>>> answer = numerix.arange(8) * dx - 3.5 * dx
>>> print(var.allclose(answer))
1
```

A 2D test case to test `_calcTrialValue` for a pathological case.

```
>>> dx = 1.
>>> dy = 2.
>>> from fipy.meshes import Grid2D
>>> mesh = Grid2D(dx = dx, dy = dy, nx = 2, ny = 3, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., 1., 1., 1., -1., 1.))
```

```
>>> var.calcDistanceFunction()
>>> vbl = -dx * dy / numerix.sqrt(dx**2 + dy**2) / 2.
>>> vbr = dx / 2
>>> vml = dy / 2.
>>> crossProd = dx * dy
>>> dsq = dx**2 + dy**2
>>> top = vbr * dx**2 + vml * dy**2
>>> sqrt = crossProd**2 * (dsq - (vbr - vml)**2)
>>> sqrt = numerix.sqrt(max(sqrt, 0))
>>> vmr = (top + sqrt) / dsq
>>> answer = (vbl, vbr, vml, vmr, vbl, vbr)
>>> print(var.allclose(answer))
1
```

The `extendVariable` method solves the following equation for a given *extensionVariable*.

$$\nabla u \cdot \nabla \phi = 0$$

using the fast marching method with an initial condition defined at the zero level set.

```
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 2, ny = 2, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., 1., 1., 1.))
>>> var.calcDistanceFunction()
>>> extensionVar = CellVariable(mesh = mesh, value = (-1, .5, 2, -1))
>>> tmp = 1 / numerix.sqrt(2)
>>> print(var.allclose((-tmp / 2, 0.5, 0.5, 0.5 + tmp)))
1
>>> var.extendVariable(extensionVar, order=1)
>>> print(extensionVar.allclose((1.25, .5, 2, 1.25)))
1
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., 1., 1.,
...                                             1., 1., 1.,
...                                             1., 1., 1.))
>>> var.calcDistanceFunction(order=1)
>>> extensionVar = CellVariable(mesh = mesh, value = (-1., .5, -1.,
...                                                    2., -1., -1.,
...                                                    -1., -1., -1.))
```

```

>>> v1 = 0.5 + tmp
>>> v2 = 1.5
>>> tmp1 = (v1 + v2) / 2 + numerix.sqrt(2. - (v1 - v2)**2) / 2
>>> tmp2 = tmp1 + 1 / numerix.sqrt(2)
>>> print(var.allclose((-tmp / 2, 0.5, 1.5, 0.5, 0.5 + tmp,
...                      tmp1, 1.5, tmp1, tmp2)))
1
>>> answer = (1.25, .5, .5, 2, 1.25, 0.9544, 2, 1.5456, 1.25)
>>> var.extendVariable(extensionVar, order=1)
>>> print(extensionVar.allclose(answer, rtol = 1e-4))
1

```

Test case for a bug that occurs when initializing the distance variable at the interface. Currently it is assumed that adjacent cells that are opposite sign neighbors have perpendicular normal vectors. In fact the two closest cells could have opposite normals.

```

>>> mesh = Grid1D(dx = 1., nx = 3, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., 1., -1.))
>>> var.calcDistanceFunction()
>>> print(var.allclose((-0.5, 0.5, -0.5)))
1

```

Testing second order. This example failed with Scikit-fmm.

```

>>> mesh = Grid2D(dx = 1., dy = 1., nx = 4, ny = 4, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., -1., 1., 1.,
...                                              -1., -1., 1., 1.,
...                                              1., 1., 1., 1.,
...                                              1, 1, 1, 1))
>>> var.calcDistanceFunction(order=2)
>>> answer = [-1.30473785, -0.5, 0.5, 1.49923009,
...          -0.5, -0.35355339, 0.5, 1.45118446,
...          0.5, 0.5, 0.97140452, 1.76215286,
...          1.49923009, 1.45118446, 1.76215286, 2.33721352]
>>> print(numerix.allclose(var, answer, rtol=1e-9))
True

```

**** A test for a bug in both LSMLIB and Scikit-fmm ****

The following test gives different result depending on whether LSMLIB or Scikit-fmm is used. There is a deeper problem that is related to this issue. When a value becomes “known” after previously being a “trial” value it updates its neighbors’ values. In a second order scheme the neighbors one step away also need to be updated (if the in between cell is “known” and the far cell is a “trial” cell), but are not in either package. By luck (due to trial values having the same value), the values calculated in Scikit-fmm for the following example are correct although an example that didn’t work for Scikit-fmm could also be constructed.

```

>>> mesh = Grid2D(dx = 1., dy = 1., nx = 4, ny = 4, communicator=serialComm)
>>> var = DistanceVariable(mesh = mesh, value = (-1., -1., -1., -1.,
...                                              1., 1., -1., -1.,
...                                              1., 1., -1., -1.,
...                                              1., 1., -1., -1.))
>>> var.calcDistanceFunction(order=2)
>>> var.calcDistanceFunction(order=2)
>>> answer = [-0.5,          -0.58578644, -1.08578644, -1.85136395,
...          0.5,          0.29289322, -0.58578644, -1.54389939,
...          1.30473785, 0.5,          -0.5,          -1.5,
...          1.49547948, 0.5,          -0.5,          -1.5]

```

The 3rd and 7th element are different for LSMLIB. This is because the 15th element is not “known” when the “trial” value for the 7th element is calculated. Scikit-fmm calculates the values in a slightly different order so gets a seemingly better answer, but this is just chance.

```
>>> print(numerix.allclose(var, answer, rtol=1e-9))
True
```

Creates a *distanceVariable* object.

Parameters

- **mesh** (*Mesh*) – The mesh that defines the geometry of this variable.
- **name** (*str*) – The name of the variable.
- **value** (*float or array_like*) – The initial value.
- **unit** (*str or PhysicalUnit*) – The physical units of the variable
- **hasOld** (*bool*) – Whether the variable maintains an old value.

__init__ (*mesh, name='', value=0.0, unit=None, hasOld=0*)

Creates a *distanceVariable* object.

Parameters

- **mesh** (*Mesh*) – The mesh that defines the geometry of this variable.
- **name** (*str*) – The name of the variable.
- **value** (*float or array_like*) – The initial value.
- **unit** (*str or PhysicalUnit*) – The physical units of the variable
- **hasOld** (*bool*) – Whether the variable maintains an old value.

__module__ = 'fipy.variables.distanceVariable'

calcDistanceFunction (*order=2*)

Calculates the *distanceVariable* as a distance function.

Parameters **order** (*{1, 2}*) – The order of accuracy for the distance function calculation

property cellInterfaceAreas

Returns the length of the interface that crosses the cell

A simple 1D test:

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(dx = 1., nx = 4)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (-1.5, -0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh, value=(0, 0., 1., 0))
>>> print(numerix.allclose(distanceVariable.cellInterfaceAreas,
...                           answer))
...
True
```

A 2D test case:

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(dx = 1., dy = 1., nx = 3, ny = 3)
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                     value = (1.5, 0.5, 1.5,
```

(continues on next page)

(continued from previous page)

```

...             0.5, -0.5, 0.5,
...             1.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                        value=(0, 1, 0, 1, 0, 1, 0, 1, 0))
>>> print(numerix.allclose(distanceVariable.cellInterfaceAreas, answer))
True

```

Another 2D test case:

```

>>> mesh = Grid2D(dx = .5, dy = .5, nx = 2, ny = 2)
>>> from fipy.variables.cellVariable import CellVariable
>>> distanceVariable = DistanceVariable(mesh = mesh,
...                                   value = (-0.5, 0.5, 0.5, 1.5))
>>> answer = CellVariable(mesh=mesh,
...                       value=(0, numerix.sqrt(2) / 4, numerix.sqrt(2) / 4,
...                               ↪ 0))
>>> print(numerix.allclose(distanceVariable.cellInterfaceAreas,
...                           answer))
...
True

```

Test to check that the circumference of a circle is, in fact, $2\pi r$.

```

>>> mesh = Grid2D(dx = 0.05, dy = 0.05, nx = 20, ny = 20)
>>> r = 0.25
>>> x, y = mesh.cellCenters
>>> rad = numerix.sqrt((x - .5)**2 + (y - .5)**2) - r
>>> distanceVariable = DistanceVariable(mesh = mesh, value = rad)
>>> print(numerix.allclose(distanceVariable.cellInterfaceAreas.sum(), 1.
... ↪57984690073))
1

```

extendVariable (*extensionVariable*, *order=2*)

Calculates the extension of *extensionVariable* from the zero level set.

Parameters **extensionVariable** (*CellVariable*) – The variable to extend from the zero level set.

getLSMshape ()

fipy.viewers package

37.1 Subpackages

37.1.1 fipy.viewers.matplotlibViewer package

Submodules

fipy.viewers.matplotlibViewer.matplotlib1DViewer module

```
class fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer(vars,
                                                                    ti-
                                                                    tle=None,
                                                                    xlog=False,
                                                                    ylog=False,
                                                                    lim-
                                                                    its={},
                                                                    leg-
                                                                    end='upper
                                                                    left',
                                                                    axes=None,
                                                                    **kwlim-
                                                                    its)
```

Bases: *fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer*

Displays a y vs. x plot of one or more 1D *CellVariable* objects using Matplotlib.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar),
↪ numerix.pi)),
...                               limits={'xmin': 10, 'xmax': 90},
...                               datamin=-0.9, datamax=2.0,
...                               title="Matplotlib1DViewer test")
```

(continues on next page)

(continued from previous page)

```
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xlog** (*bool*) – log scaling of x axis if *True*
- **ylog** (*bool*) – log scaling of y axis if *True*
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **legend** (*str*) – place a legend at the specified position, if not *None*
- **axes** (*Axes*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

```
__init__ (vars, title=None, xlog=False, ylog=False, limits={}, legend='upper left', axes=None,
          **kwlimits)
```

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xlog** (*bool*) – log scaling of x axis if *True*
- **ylog** (*bool*) – log scaling of y axis if *True*
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **legend** (*str*) – place a legend at the specified position, if not *None*
- **axes** (*Axes*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlib1DViewer'
```

```
property log
```

```
    logarithmic data scaling
```

fiPy.viewers.matplotlibViewer.matplotlib2DContourViewer module

class fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.**Matplotlib2DContourViewer** (var

Bases: fipy.viewers.matplotlibViewer.matplotlib2DViewer.
AbstractMatplotlib2DViewer

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DContourViewer* plots a 2D *CellVariable* using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DContourViewer(vars=numerix.sin(k * xyVar),
...                                     limits={'ymin': 0.1, 'ymax': 0.9},
...                                     datamin=-0.9, datamax=2.0,
...                                     title="Matplotlib2DContourViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DContourViewer*.

Parameters

- **vars** (*CellVariable*) – Variable to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this *Matplotlib Axes* object

- **number** (*int*, *optional*) – contour *number* automatically-chosen levels
- **levels** (*list* of *float*, *optional*) – A list of numbers indicating the level curves to draw; e.g. to draw just the zero contour pass `levels=[0]`
- **figaspect** (*float*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the Variable’s mesh.

`__init__` (*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *number=10*, *levels=None*, *figaspect='auto'*, ***kwlimits*)
Creates a *Matplotlib2DContourViewer*.

Parameters

- **vars** (*CellVariable*) – Variable to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the Colormap. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **number** (*int*, *optional*) – contour *number* automatically-chosen levels
- **levels** (*list* of *float*, *optional*) – A list of numbers indicating the level curves to draw; e.g. to draw just the zero contour pass `levels=[0]`
- **figaspect** (*float*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the Variable’s mesh.

`__module__` = `'fipy.viewers.matplotlibViewer.matplotlib2DContourViewer'`

`fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer` module

`class` `fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer`.**Matplotlib2DGridContourV**

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DGridContourViewer* plots a 2D *CellVariable* using [Matplotlib](#).

```

>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer(vars=numerix.sin(k * xyVar),
...                                     limits={'ymin': 0.1, 'ymax': 0.9},
...                                     datamin=-0.9, datamax=2.0,
...                                     title="Matplotlib2DGridContourViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this *Matplotlib Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If a number, use that aspect ratio. If *auto*, the aspect ratio will be determined from the *vars*'s mesh.

__init__ (*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this *Matplotlib Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If a number, use that aspect ratio. If *auto*, the aspect ratio will be determined from the *vars*'s mesh.

__module__ = *'fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer'*

fipy.viewers.matplotlibViewer.matplotlib2DGridViewer module

```
class fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer(vars,
ti-
tle=None,
lim-
its={},
cmap=None,
col-
or-
bar='vertical',
axes=None,
fi-
gaspect='auto',
**kwargs)
its)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`.
`AbstractMatplotlib2DViewer`

Displays an image plot of a 2D *CellVariable* object using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=numerix.sin(k * xyVar),
...                               limits={'ymin': 0.1, 'ymax': 0.9},
...                               datamin=-0.9, datamax=2.0,
...                               title="Matplotlib2DGridViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DGridViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this *Matplotlib Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.


```
__init__ (vars, title=None, limits={}, cmap=None, colorbar='vertical', axes=None, figaspect='auto',
          **kwlimits)
    Creates a Matplotlib2DGridViewer.
```

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this *Matplotlib Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlib2DGridViewer'
```

fipy.viewers.matplotlibViewer.matplotlib2DViewer module

```
class fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer (vars,
                                                                           ti-
                                                                           tle=None,
                                                                           lim-
                                                                           its={},
                                                                           cmap=None,
                                                                           col-
                                                                           or-
                                                                           bar='vertical',
                                                                           axes=None,
                                                                           fi-
                                                                           gaspect='auto',
                                                                           **kwlim-
                                                                           its)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DViewer* plots a 2D *CellVariable* using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5, ), (0.2, ))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=numerix.sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
```

(continues on next page)

(continued from previous page)

```

...         datamin=-0.9, datamax=2.0,
...         title="Matplotlib2DViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

```
__init__(vars, title=None, limits={}, cmap=None, colorbar='vertical', axes=None, figaspect='auto',
        **kwlimits)
```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlib2DViewer'
```

fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer module

```

class fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.MatplotlibSparseMatrixViewer
    Bases: object

    __dict__ = mappingproxy({'__module__': 'fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer'})
    __init__(title='Sparsity')
        Initialize self. See help(type(self)) for accurate signature.
    __module__ = 'fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer'
    __weakref__
        list of weak references to the object (if defined)
    plot(matrix, RHSvector, log='auto')

```

fipy.viewers.matplotlibViewer.matplotlibStreamViewer module

```

class fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer(vars,
    title=None,
    log=False,
    limits={},
    axes=None,
    figure=None,
    aspect='auto',
    density=1,
    linewidth=None,
    color=None,
    cmap=None,
    norm=None,
    arrowsize=1,
    arrowstyle='>',
    minlength=0.1,
    **kwargs)

```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer.AbstractMatplotlib2DViewer`

Displays a stream plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using [Matplotlib](#)

One issue is that this *Viewer* relies on `scipy.interpolate.griddata`, which interpolates on the convex hull of the data. The results is that streams are plotted across any concavities in the mesh.

Another issue is that it does not seem possible to remove the streams without calling `cla()`, which means that different set of streams cannot be overlaid.

```

>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters

```

(continues on next page)

(continued from previous page)

```
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).grad,
...                               title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                               title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).grad,
...                               title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                               title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *MatplotlibStreamViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

- **density** (*float* or (*float*, *float*), *optional*) – Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (`density_x`, `density_y`).
- **linewidth** (*array_like* or *CellVariable* or *FaceVariable*, *optional*) – The width of the stream lines. With a rank-0 *CellVariable* or *FaceVariable* the line width can be varied across the grid. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **color** (*str* or *CellVariable* or *FaceVariable*, *optional*) – The streamline color as a matplotlib color code or a field of numbers. If given a rank-0 *CellVariable* or *FaceVariable*, its values are converted to colors using *cmap* and *norm*. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **cmap** (*Colormap*, *optional*) – Colormap used to plot streamlines and arrows. This is only used if *color* is a *MeshVariable*.
- **norm** (*Normalize*, *optional*) – Normalize object used to scale luminance data to 0, 1. If *None*, stretch (min, max) to (0, 1). Only necessary when *color* is a *MeshVariable*.
- **arrowsize** (*float*, *optional*) – Scaling factor for the arrow size.
- **arrowstyle** (*str*, *optional*) – Arrow style specification. See `~matplotlib.patches.FancyArrowPatch`.
- **minlength** (*float*, *optional*) – Minimum length of streamline in axes coordinates.

`__init__`(*vars*, *title=None*, *log=False*, *limits={}*, *axes=None*, *figaspect='auto'*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='|>'*, *minlength=0.1*, ***kwlimits*)
Creates a *MatplotlibStreamViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.
- **density** (*float* or (*float*, *float*), *optional*) – Controls the closeness of streamlines. When `density = 1`, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (`density_x`, `density_y`).
- **linewidth** (*array_like* or *CellVariable* or *FaceVariable*, *optional*) – The width of the stream lines. With a rank-0 *CellVariable* or *FaceVariable* the line width can be varied across the grid. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.

- **color** (*str* or *CellVariable* or *FaceVariable*, *optional*) – The streamline color as a matplotlib color code or a field of numbers. If given a rank-0 *CellVariable* or *FaceVariable*, its values are converted to colors using *cmap* and *norm*. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **cmap** (*Colormap*, *optional*) – Colormap used to plot streamlines and arrows. This is only used if *color* is a *MeshVariable*.
- **norm** (*Normalize*, *optional*) – Normalize object used to scale luminance data to 0, 1. If *None*, stretch (min, max) to (0, 1). Only necessary when *color* is a *MeshVariable*.
- **arrowsize** (*float*, *optional*) – Scaling factor for the arrow size.
- **arrowstyle** (*str*, *optional*) – Arrow style specification. See *~matplotlib.patches.FancyArrowPatch*.
- **minlength** (*float*, *optional*) – Minimum length of streamline in axes coordinates.

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlibStreamViewer'
```

fipy.viewers.matplotlibViewer.matplotlibVectorViewer module

```
class fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer (vars,
ti-
tle=None,
scale=None,
spar-
sity=None,
log=False,
lim-
its={},
axes=None,
fi-
gaspect='au
**kwlim-
its)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using *Matplotlib*

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
```

(continues on next page)

(continued from previous page)

```
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> for sparsity in numerix.arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **scale** (*float*, *optional*) – if not *None*, scale all arrow lengths by this value
- **sparsity** (*int*, *optional*) – if not *None*, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*’s mesh.

```
__init__(vars, title=None, scale=None, sparsity=None, log=False, limits={}, axes=None, fi-
        gaspect='auto', **kwlimits)
    Creates a Matplotlib2DViewer.
```

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **scale** (*float*, *optional*) – if not *None*, scale all arrow lengths by this value
- **sparsity** (*int*, *optional*) – if not *None*, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this *Matplotlib Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlibVectorViewer'
```

```
quiver (sparsity=None, scale=None)
```

fipy.viewers.matplotlibViewer.matplotlibViewer module

```
class fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer (vars,
ti-
tle=None,
fi-
gaspect=1.0,
cmap=None,
col-
or-
bar=None,
axes=None,
log=False,
**kwlim-
its)
```

Bases: *fipy.viewers.viewer.AbstractViewer*

Attention: This class is abstract. Always create one of its subclasses.

The *AbstractMatplotlibViewer* is the base class for the viewers that use the *Matplotlib* python plotting package. Create a *AbstractMatplotlibViewer*.

Parameters

- **vars** (*CellVariable* or *list*) – the *Variable* objects to display.

- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the Variable's mesh.
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **log** (*bool*, *optional*) – whether to logarithmically scale the data

__init__ (*vars*, *title=None*, *figaspect=1.0*, *cmap=None*, *colorbar=None*, *axes=None*, *log=False*, ***kwlimits*)
Create a *AbstractMatplotlibViewer*.

Parameters

- **vars** (*CellVariable* or *list*) – the *Variable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the Variable's mesh.
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **log** (*bool*, *optional*) – whether to logarithmically scale the data

__module__ = 'fipy.viewers.matplotlibViewer.matplotlibViewer'

figaspect (*figaspect*)

property log

logarithmic data scaling

plot (*filename=None*)

Update the display of the viewed variables.

Parameters filename (*str*) – If not *None*, the name of a file to save the image into.

fipy.viewers.matplotlibViewer.test module

Test numeric implementation of the mesh

Module contents

`fipy.viewers.matplotlibViewer.MatplotlibViewer` (*vars*, *title=None*, *limits={}*,
cmap=None, *colorbar='vertical'*,
axes=None, ***kwlimits*)

Generic function for creating a *MatplotlibViewer*.

The *MatplotlibViewer* factory will search the module tree and return an instance of the first *MatplotlibViewer* it finds of the correct dimension and rank.

It is possible to view different *Variables* against different *Matplotlib Axes*

```
>>> from matplotlib import pyplot as plt
>>> from fipy import *
```

```
>>> plt.ion()
>>> fig = plt.figure()
```

```
>>> ax1 = plt.subplot((221))
>>> ax2 = plt.subplot((223))
>>> ax3 = plt.subplot((224))
```

```
>>> k = Variable(name="k", value=0.)
```

```
>>> mesh1 = Grid1D(nx=100)
>>> x, = mesh1.cellCenters
>>> xVar = CellVariable(mesh=mesh1, name="x", value=x)
>>> viewer1 = MatplotlibViewer(vars=(numerix.sin(0.1 * k * xVar), numerix.cos(0.1 *
↪ k * xVar / numerix.pi)),
...                               limits={'xmin': 10, 'xmax': 90},
...                               datamin=-0.9, datamax=2.0,
...                               title="Grid1D test",
...                               axes=ax1,
...                               legend=None)
```

```
>>> mesh2 = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh2.cellCenters
>>> xyVar = CellVariable(mesh=mesh2, name="x y", value=x * y)
>>> viewer2 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
...                               limits={'ymin': 0.1, 'ymax': 0.9},
...                               datamin=-0.9, datamax=2.0,
...                               title="Grid2D test",
...                               axes=ax2,
...                               colorbar=None)
```

```
>>> mesh3 = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...          + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...             + ((0.5,), (0.2,))))
>>> x, y = mesh3.cellCenters
>>> xyVar = CellVariable(mesh=mesh3, name="x y", value=x * y)
>>> viewer3 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
```

(continues on next page)

(continued from previous page)

```

...             limits={'ymin': 0.1, 'ymax': 0.9},
...             datamin=-0.9, datamax=2.0,
...             title="Irregular 2D test",
...             axes=ax3,
...             cmap = plt.cm.OrRd)

```

```

>>> viewer = MultiViewer(viewers=(viewer1, viewer2, viewer3))
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()

```

```

>>> viewer._promptForOpinion()

```

Parameters

- **vars** (*CellVariable* or *list*) – the *Variable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

```

class fipy.viewers.matplotlibViewer.Matplotlib1DViewer (vars, title=None,
                                                         xlog=False, ylog=False,
                                                         limits={}, legend='upper
                                                         left', axes=None, **kwlim-
                                                         its)

```

Bases: *fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer*

Displays a y vs. x plot of one or more 1D *CellVariable* objects using *Matplotlib*.

```

>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar),
↪ numerix.pi)),
...             limits={'xmin': 10, 'xmax': 90},
...             datamin=-0.9, datamax=2.0,
...             title="Matplotlib1DViewer test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xlog** (*bool*) – log scaling of x axis if *True*
- **ylog** (*bool*) – log scaling of y axis if *True*
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **legend** (*str*) – place a legend at the specified position, if not *None*
- **axes** (*Axes*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

```
__init__(vars, title=None, xlog=False, ylog=False, limits={}, legend='upper left', axes=None,
        **kwlimits)
```

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xlog** (*bool*) – log scaling of x axis if *True*
- **ylog** (*bool*) – log scaling of y axis if *True*
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **legend** (*str*) – place a legend at the specified position, if not *None*
- **axes** (*Axes*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlib1DViewer'
```

property log

logarithmic data scaling

```
class fipy.viewers.matplotlibViewer.Matplotlib2DGridViewer (vars, title=None, limits={}, cmap=None,
                                                            colorbar='vertical',
                                                            axes=None,          fi-
                                                            gaspect='auto',
                                                            **kwlimits)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays an image plot of a 2D *CellVariable* object using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=numerix.sin(k * xyVar),
```

(continues on next page)

(continued from previous page)

```

...         limits={'ymin': 0.1, 'ymax': 0.9},
...         datamin=-0.9, datamax=2.0,
...         title="Matplotlib2DGridViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

Creates a *Matplotlib2DGridViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

__init__ (*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Creates a *Matplotlib2DGridViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

__module__ = 'fipy.viewers.matplotlibViewer.matplotlib2DGridViewer'

```
class fipy.viewers.matplotlibViewer.Matplotlib2DGridContourViewer (vars,      ti-
                                                                    tle=None,
                                                                    limits={},
                                                                    cmap=None,
                                                                    color-
                                                                    bar='vertical',
                                                                    axes=None,
                                                                    fi-
                                                                    gaspect='auto',
                                                                    **kwlim-
                                                                    its)

Bases:
    fipy.viewers.matplotlibViewer.matplotlib2DViewer.
    AbstractMatplotlib2DViewer
```

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DGridContourViewer* plots a 2D *CellVariable* using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer(vars=numerix.sin(k * xyVar),
...                                     limits={'ymin': 0.1, 'ymax': 0.9},
...                                     datamin=-0.9, datamax=2.0,
...                                     title="Matplotlib2DGridContourViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this *Matplotlib Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If a number, use that aspect ratio. If *auto*, the aspect ratio will be determined from the *vars*'s mesh.

```
__init__ (vars, title=None, limits={}, cmap=None, colorbar='vertical', axes=None, figaspect='auto',
          **kwlimits)
```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.

- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If a number, use that aspect ratio. If *auto*, the aspect ratio will be determined from the *vars*’s mesh.

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer'
class fipy.viewers.matplotlibViewer.Matplotlib2DViewer (vars, title=None, limits={}, cmap=None, colorbar='vertical', axes=None, figaspect='auto', **kwlimits)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DViewer* plots a 2D *CellVariable* using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...         + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=numerix.sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Matplotlib2DViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*

- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the Variable's mesh.

__init__ (*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)
Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the Variable's mesh.

__module__ = 'fipy.viewers.matplotlibViewer.matplotlib2DViewer'

```
class fipy.viewers.matplotlibViewer.MatplotlibVectorViewer (vars, title=None,
                                                            scale=None, spar-
                                                            sity=None, log=False,
                                                            limits={}, axes=None,
                                                            figaspect='auto',
                                                            **kwlimits)
```

Bases: *fipy.viewers.matplotlibViewer.matplotlib2DViewer*, *AbstractMatplotlib2DViewer*

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using *Matplotlib*

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
```

(continues on next page)

(continued from previous page)

```
... viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> for sparsity in numerix.arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5, ), (0.2, ))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                                 title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **scale** (*float*, *optional*) – if not *None*, scale all arrow lengths by this value
- **sparsity** (*int*, *optional*) – if not *None*, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*’s mesh.

__init__ (*vars*, *title=None*, *scale=None*, *sparsity=None*, *log=False*, *limits={}*, *axes=None*, *figaspect='auto'*, ***kwlimits*)
Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **scale** (*float*, *optional*) – if not *None*, scale all arrow lengths by this value
- **sparsity** (*int*, *optional*) – if not *None*, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlibVectorViewer'
```

```
quiver (sparsity=None, scale=None)
```

```
class fipy.viewers.matplotlibViewer.MatplotlibStreamViewer (vars, title=None,
                                                            log=False, limits={}, axes=None,
                                                            figaspect='auto',
                                                            density=1,
                                                            linewidth=None,
                                                            color=None,
                                                            cmap=None,
                                                            norm=None, arrow-
                                                            size=1, arrowstyle='->',
                                                            minlength=0.1,
                                                            **kwargs)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays a stream plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using [Matplotlib](#)

One issue is that this *Viewer* relies on `scipy.interpolate.griddata`, which interpolates on the convex hull of the data. The results is that streams are plotted across any concavities in the mesh.

Another issue is that it does not seem possible to remove the streams without calling `cla()`, which means that different set of streams cannot be overlaid.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).grad,
...                               title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
```

(continues on next page)

(continued from previous page)

```
... viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                                  title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).grad,
...                                  title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                                  title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *MatplotlibStreamViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.
- **density** (*float* or (*float*, *float*), *optional*) – Controls the closeness of streamlines. When *density* = 1, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (*density_x*, *density_y*).

- **linewidth** (*array_like* or *CellVariable* or *FaceVariable*, *optional*) – The width of the stream lines. With a rank-0 *CellVariable* or *FaceVariable* the line width can be varied across the grid. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **color** (*str* or *CellVariable* or *FaceVariable*, *optional*) – The streamline color as a matplotlib color code or a field of numbers. If given a rank-0 *CellVariable* or *FaceVariable*, its values are converted to colors using *cmap* and *norm*. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **cmap** (*Colormap*, *optional*) – Colormap used to plot streamlines and arrows. This is only used if *color* is a *MeshVariable*.
- **norm** (*Normalize*, *optional*) – Normalize object used to scale luminance data to 0, 1. If *None*, stretch (min, max) to (0, 1). Only necessary when *color* is a *MeshVariable*.
- **arrowsize** (*float*, *optional*) – Scaling factor for the arrow size.
- **arrowstyle** (*str*, *optional*) – Arrow style specification. See `~matplotlib.patches.FancyArrowPatch`.
- **minlength** (*float*, *optional*) – Minimum length of streamline in axes coordinates.

`__init__`(*vars*, *title=None*, *log=False*, *limits={}*, *axes=None*, *figaspect='auto'*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='|>'*, *minlength=0.1*, ***kwlimits*)
Creates a *MatplotlibStreamViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.
- **density** (*float* or (*float*, *float*), *optional*) – Controls the closeness of streamlines. When *density* = 1, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (*density_x*, *density_y*).
- **linewidth** (*array_like* or *CellVariable* or *FaceVariable*, *optional*) – The width of the stream lines. With a rank-0 *CellVariable* or *FaceVariable* the line width can be varied across the grid. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **color** (*str* or *CellVariable* or *FaceVariable*, *optional*) – The streamline color as a matplotlib color code or a field of numbers. If given a rank-0 *CellVariable* or *FaceVariable*, its values are converted to colors using *cmap* and *norm*. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.

- **cmap** (*Colormap*, *optional*) – Colormap used to plot streamlines and arrows. This is only used if *color* is a MeshVariable.
- **norm** (*Normalize*, *optional*) – Normalize object used to scale luminance data to 0, 1. If None, stretch (min, max) to (0, 1). Only necessary when *color* is a MeshVariable.
- **arrowsize** (*float*, *optional*) – Scaling factor for the arrow size.
- **arrowstyle** (*str*, *optional*) – Arrow style specification. See `~matplotlib.patches.FancyArrowPatch`.
- **minlength** (*float*, *optional*) – Minimum length of streamline in axes coordinates.

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlibStreamViewer'
```

37.1.2 fipy.viewers.mayaviViewer package

Submodules

fipy.viewers.mayaviViewer.mayaviClient module

```
class fipy.viewers.mayaviViewer.mayaviClient.MayaviClient (vars, title=None,
                                                           daemon_file=None,
                                                           fps=1.0, **kwlimits)
```

Bases: `fipy.viewers.viewer.AbstractViewer`

The *MayaviClient* uses the *Mayavi* python plotting package.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar /
↳numerix.pi)),
...                       limits={'xmin': 10, 'xmax': 90},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5, ), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

```

>>> from fipy import *
>>> mesh = Grid3D(nx=50, ny=100, nz=10, dx=0.1, dy=0.01, dz=0.1)
>>> x, y, z = mesh.cellCenters
>>> xyzVar = CellVariable(mesh=mesh, name="x y z", value=x * y * z)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyzVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

Create a *MayaviClient*.

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **daemon_file** (*str*, *optional*) – the path to the script to run the separate Mayavi viewer process. Defaults to *fipy/viewers/mayaviViewer/mayaviDaemon.py*
- **fps** (*float*, *optional*) – frames per second to attempt to display

`__del__()`

`__init__(vars, title=None, daemon_file=None, fps=1.0, **kwlimits)`

Create a *MayaviClient*.

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window

- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D Viewer will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **daemon_file** (*str*, *optional*) – the path to the script to run the separate Mayavi viewer process. Defaults to *fipy/viewers/mayaviViewer/mayaviDaemon.py*
- **fps** (*float*, *optional*) – frames per second to attempt to display

__module__ = 'fipy.viewers.mayaviViewer.mayaviClient'

plot (*filename=None*)

Update the display of the viewed variables.

Parameters filename (*str*) – If not *None*, the name of a file to save the image into.

fipy.viewers.mayaviViewer.mayaviDaemon module

A simple script that polls a data file for changes and then updates the Mayavi pipeline automatically.

This script is based heavily on the *poll_file.py* example in the Mayavi distribution.

This script is to be run like so:

```
$ mayavi2 -x mayaviDaemon.py <options>
```

Or:

```
$ python mayaviDaemon.py <options>
```

Run:

```
$ python mayaviDaemon.py --help
```

to see available options.

class fipy.viewers.mayaviViewer.mayaviDaemon.**MayaviDaemon**

Bases: *mayavi.plugins.app.Mayavi*

Given a file name and a mayavi2 data reader object, this class polls the file for any changes and automatically updates the mayavi pipeline.

__base_traits__ = {'application': <traits.ctrait.CTrait object>, 'log_mode': <traits

__class_traits__ = {'application': <traits.ctrait.CTrait object>, 'log_mode': <trait

__del__ ()

__instance_traits__ = {}

__listener_traits__ = {'_on_application_gui_started': ('method', {'pattern': 'applic

__module__ = 'fipy.viewers.mayaviViewer.mayaviDaemon'

__prefix_traits__ = {'': <traits.ctrait.CTrait object>, '*': ['_traits_cache_', ''],

__view_traits__ = {}

clip_data (*src*)

parse_command_line (*argv*)

Parse command line options.

Parameters **argv** (list of str) – The command line arguments

poll_file()

run()

This function is called after the GUI has started. Override this to do whatever you want to do as a MayaVi script. If this is not overridden then an empty MayaVi application will be started.

Make sure all other MayaVi specific imports are made here! If you import MayaVi related code earlier you will run into difficulties. Use 'self.script' to script the mayavi engine.

setup_source(fname)

Given a VTK file name *fname*, this creates a mayavi2 reader for it and adds it to the pipeline. It returns the reader created.

update_pipeline(source)

Override this to do something else if needed.

view_data()

Sets up the mayavi pipeline for the visualization.

fipy.viewers.mayaviViewer.test module

Test numeric implementation of the mesh

Module contents

class fipy.viewers.mayaviViewer.**MayaviClient** (vars, title=None, daemon_file=None, fps=1.0, **kwlimits)

Bases: *fipy.viewers.viewer.AbstractViewer*

The *MayaviClient* uses the *Mayavi* python plotting package.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar /
↳numerix.pi)),
...                       limits={'xmin': 10, 'xmax': 90},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
```

(continues on next page)

(continued from previous page)

```
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = Grid3D(nx=50, ny=100, nz=10, dx=0.1, dy=0.01, dz=0.1)
>>> x, y, z = mesh.cellCenters
>>> xyzVar = CellVariable(mesh=mesh, name="x y z", value=x * y * z)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyzVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Create a *MayaviClient*.

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, optional) – displayed at the top of the *Viewer* window
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **daemon_file** (*str*, optional) – the path to the script to run the separate Mayavi viewer process. Defaults to *fipy/viewers/mayaviViewer/mayaviDaemon.py*
- **fps** (*float*, optional) – frames per second to attempt to display

`__del__()`

`__init__` (*vars*, *title=None*, *daemon_file=None*, *fps=1.0*, ***kwlimits*)
Create a *MayaviClient*.

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **daemon_file** (*str*, *optional*) – the path to the script to run the separate Mayavi viewer process. Defaults to *fipy/viewers/mayaviViewer/mayaviDaemon.py*
- **fps** (*float*, *optional*) – frames per second to attempt to display

`__module__` = `'fipy.viewers.mayaviViewer.mayaviClient'`

`plot` (*filename=None*)
Update the display of the viewed variables.

Parameters **filename** (*str*) – If not *None*, the name of a file to save the image into.

37.1.3 fipy.viewers.vtkViewer package

Submodules

`fipy.viewers.vtkViewer.test` module

Test numeric implementation of the mesh

`fipy.viewers.vtkViewer.vtkCellViewer` module

class `fipy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer` (*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer.VTKViewer`

Renders *CellVariable* data in VTK format

Creates a *VTKViewer*

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

`__module__` = `'fipy.viewers.vtkViewer.vtkCellViewer'`

fipy.viewers.vtkViewer.vtkFaceViewer module

class fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer(*vars*, *title=None*, *limits={}, **kwlimits*)

Bases: *fipy.viewers.vtkViewer.vtkViewer.VTKViewer*

Renders *_MeshVariable* data in VTK format

Creates a *VTKViewer*

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

__module__ = 'fipy.viewers.vtkViewer.vtkFaceViewer'

fipy.viewers.vtkViewer.vtkViewer module

class fipy.viewers.vtkViewer.vtkViewer.VTKViewer(*vars*, *title=None*, *limits={}, **kwlimits*)

Bases: *fipy.viewers.viewer.AbstractViewer*

Renders *_MeshVariable* data in VTK format

Creates a *VTKViewer*

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

__init__ (*vars*, *title=None*, *limits={}, **kwlimits*)

Creates a *VTKViewer*

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

__module__ = 'fipy.viewers.vtkViewer.vtkViewer'

plot (*filename=None*)

Update the display of the viewed variables.

Parameters **filename** (*str*) – If not *None*, the name of a file to save the image into.

Module contents

`fipy.viewers.vtkViewer.VTKViewer` (*vars*, *title=None*, *limits={}*, ***kwlimits*)

Generic function for creating a *VTKViewer*.

The *VTKViewer* factory will search the module tree and return an instance of the first *VTKViewer* it finds of the correct dimension and rank.

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

class `fipy.viewers.vtkViewer.VTKCellViewer` (*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer.VTKViewer`

Renders *CellVariable* data in VTK format

Creates a *VTKViewer*

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

`__module__` = `'fipy.viewers.vtkViewer.vtkCellViewer'`

class `fipy.viewers.vtkViewer.VTKFaceViewer` (*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer.VTKViewer`

Renders *_MeshVariable* data in VTK format

Creates a *VTKViewer*

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

`__module__` = `'fipy.viewers.vtkViewer.vtkFaceViewer'`

37.2 Submodules

37.3 fipy.viewers.multiViewer module

class fipy.viewers.multiViewer.**MultiViewer**(viewers)

Bases: *fipy.viewers.viewer.AbstractViewer*

Treat a collection of different viewers (such for different 2D plots or 1D plots with different axes) as a single viewer that will *plot()* all subviewers simultaneously.

Parameters **viewers** (list of ~fipy.viewers.viewer.Viewer) – the viewers to bind together

__init__ (viewers)

Parameters **viewers** (list of ~fipy.viewers.viewer.Viewer) – the viewers to bind together

__module__ = 'fipy.viewers.multiViewer'

plot ()

Update the display of the viewed variables.

Parameters **filename** (*str*) – If not *None*, the name of a file to save the image into.

setLimits (limits={}, **kwlimits)

Update the limits.

Parameters

- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.

37.4 fipy.viewers.test module

Test implementation of the viewers

37.5 fipy.viewers.testinteractive module

Interactively test the viewers

37.6 fipy.viewers.tsvViewer module

class fipy.viewers.tsvViewer.**TSVViewer**(vars, title=None, limits={}, **kwlimits)

Bases: *fipy.viewers.viewer.AbstractViewer*

“Views” one or more variables in tab-separated-value format.

Output is a list of coordinates and variable values at each cell center.

File contents will be, e.g.:

```

title
x      y      ...      var0      var2      ...
0.0    0.0    ...      3.14      1.41      ...
1.0    0.0    ...      2.72      0.866     ...
:
:

```

Creates a *TSVViewer*.

Any cell centers that lie outside the limits provided will not be included. Any values that lie outside the *datamin* or *datamax* will be replaced with *nan*.

All variables must have the same mesh.

It tries to do something reasonable with rank-1 *CellVariable* and *FaceVariable* objects.

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

__init__ (*vars*, *title=None*, *limits={}*, ***kwlimits*)

Creates a *TSVViewer*.

Any cell centers that lie outside the limits provided will not be included. Any values that lie outside the *datamin* or *datamax* will be replaced with *nan*.

All variables must have the same mesh.

It tries to do something reasonable with rank-1 *CellVariable* and *FaceVariable* objects.

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

__module__ = `'fipy.viewers.tsvViewer'`

plot (*filename=None*)

“plot” the coordinates and values of the variables to *filename*. If *filename* is not provided, “plots” to *stdout*.

```

>>> from fipy.meshes import Grid1D
>>> m = Grid1D(nx = 3, dx = 0.4)
>>> from fipy.variables.cellVariable import CellVariable
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, 5))
>>> TSVViewer(vars = (v, v.grad)).plot()
x      var      var_gauss_grad_x
0.2    0         2.5
0.6    2         6.25
1      5         3.75

```

```
>>> from fipy.meshes import Grid2D
>>> m = Grid2D(nx = 2, dx = .1, ny = 2, dy = 0.3)
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, -2, 5))
>>> TSVViewer(vars = (v, v.grad)).plot()
x          y          var          var_gauss_grad_x          var_gauss_grad_y
0.05      0.15         0          10          -3.333333333333333
0.15      0.15         2          10           5
0.05      0.45        -2          35          -3.333333333333333
0.15      0.45         5          35           5
```

Parameters `filename` (*str*) – If not *None*, the name of a file to save the image into.

37.7 fipy.viewers.viewer module

class `fipy.viewers.viewer.AbstractViewer` (*vars*, *title=None*, ***kwlimits*)
 Bases: `object`

Attention: This class is abstract. Always create one of its subclasses.

Create a *AbstractViewer* object.

Parameters

- **vars** (*CellVariable* or *list*) – the *CellVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

`__dict__` = `mappingproxy({'__module__': 'fipy.viewers.viewer', '__doc__': '\n .. atte`

`__init__` (*vars*, *title=None*, ***kwlimits*)

Create a *AbstractViewer* object.

Parameters

- **vars** (*CellVariable* or *list*) – the *CellVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

`__module__` = `'fipy.viewers.viewer'`

`__weakref__`

list of weak references to the object (if defined)

plot (*filename=None*)

Update the display of the viewed variables.

Parameters `filename` (*str*) – If not *None*, the name of a file to save the image into.

plotMesh (*filename=None*)

Display a representation of the mesh

Parameters `filename` (*str*) – If not *None*, the name of a file to save the image into.

setLimits (*limits*={}, ***kwlimits*)

Update the limits.

Parameters

- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.

37.8 Module contents

`fipy.viewers.MatplotlibViewer` (*vars*, *title*=None, *limits*={}, *cmap*=None, *colorbar*='vertical', *axes*=None, ***kwlimits*)

Generic function for creating a *MatplotlibViewer*.

The *MatplotlibViewer* factory will search the module tree and return an instance of the first *MatplotlibViewer* it finds of the correct dimension and rank.

It is possible to view different *Variables* against different *Matplotlib Axes*

```
>>> from matplotlib import pyplot as plt
>>> from fipy import *
```

```
>>> plt.ion()
>>> fig = plt.figure()
```

```
>>> ax1 = plt.subplot((221))
>>> ax2 = plt.subplot((223))
>>> ax3 = plt.subplot((224))
```

```
>>> k = Variable(name="k", value=0.)
```

```
>>> mesh1 = Grid1D(nx=100)
>>> x, = mesh1.cellCenters
>>> xVar = CellVariable(mesh=mesh1, name="x", value=x)
>>> viewer1 = MatplotlibViewer(vars=(numerix.sin(0.1 * k * xVar), numerix.cos(0.1 *
↪ * k * xVar / numerix.pi)),
...                               limits={'xmin': 10, 'xmax': 90},
...                               datamin=-0.9, datamax=2.0,
...                               title="Grid1D test",
...                               axes=ax1,
...                               legend=None)
```

```
>>> mesh2 = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh2.cellCenters
>>> xyVar = CellVariable(mesh=mesh2, name="x y", value=x * y)
>>> viewer2 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Grid2D test",
...                             axes=ax2,
...                             colorbar=None)
```



```
>>> mesh3 = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...          + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...            + ((0.5,), (0.2,))))
>>> x, y = mesh3.cellCenters
>>> xyVar = CellVariable(mesh=mesh3, name="x y", value=x * y)
>>> viewer3 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Irregular 2D test",
...                             axes=ax3,
...                             cmap = plt.cm.OrRd)
```

```
>>> viewer = MultiViewer(viewers=(viewer1, viewer2, viewer3))
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
```

```
>>> viewer._promptForOpinion()
```

Parameters

- **vars** (*CellVariable* or *list*) – the *Variable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

class fipy.viewers.**Matplotlib1DViewer** (*vars*, *title=None*, *xlog=False*, *ylog=False*, *limits={}*,
legend='upper left', *axes=None*, ***kwlimits*)

Bases: *fipy.viewers.matplotlibViewer.matplotlibViewer*,
AbstractMatplotlibViewer

Displays a y vs. x plot of one or more 1D *CellVariable* objects using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar),
↪ numerix.pi)),
...                             limits={'xmin': 10, 'xmax': 90},
...                             datamin=-0.9, datamax=2.0,
...                             title="Matplotlib1DViewer test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
```

(continues on next page)

(continued from previous page)

```
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xlog** (*bool*) – log scaling of x axis if *True*
- **ylog** (*bool*) – log scaling of y axis if *True*
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **legend** (*str*) – place a legend at the specified position, if not *None*
- **axes** (*Axes*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

```
__init__(vars, title=None, xlog=False, ylog=False, limits={}, legend='upper left', axes=None,
        **kwlimits)
```

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xlog** (*bool*) – log scaling of x axis if *True*
- **ylog** (*bool*) – log scaling of y axis if *True*
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **legend** (*str*) – place a legend at the specified position, if not *None*
- **axes** (*Axes*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlib1DViewer'
```

property log

logarithmic data scaling

```
class fipy.viewers.Matplotlib2DGridViewer (vars, title=None, limits={}, cmap=None, col-
                                         orbar='vertical', axes=None, figaspect='auto',
                                         **kwlimits)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays an image plot of a 2D *CellVariable* object using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
```

(continues on next page)

(continued from previous page)

```

>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=numerix.sin(k * xyVar),
...                               limits={'ymin': 0.1, 'ymax': 0.9},
...                               datamin=-0.9, datamax=2.0,
...                               title="Matplotlib2DGridViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

Creates a *Matplotlib2DGridViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

__init__ (*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Creates a *Matplotlib2DGridViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

__module__ = `'fipy.viewers.matplotlibViewer.matplotlib2DGridViewer'`

```
class fipy.viewers.Matplotlib2DGridContourViewer (vars, title=None, limits={},
                                                  cmap=None, colorbar='vertical',
                                                  axes=None, figaspect='auto',
                                                  **kwlimits)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DGridContourViewer* plots a 2D *CellVariable* using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer (vars=numerix.sin(k * xyVar),
...                                         limits={'ymin': 0.1, 'ymax': 0.9},
...                                         datamin=-0.9, datamax=2.0,
...                                         title="Matplotlib2DGridContourViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **colorbar** (*bool*, *optional*) – plot a color bar if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this *Matplotlib Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If a number, use that aspect ratio. If *auto*, the aspect ratio will be determined from the *vars*'s mesh.

```
__init__ (vars, title=None, limits={}, cmap=None, colorbar='vertical', axes=None, figaspect='auto',
          **kwlimits)
```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*

- **colorbar** (*bool*, *optional*) – plot a color bar if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If a number, use that aspect ratio. If *auto*, the aspect ratio will be determined from the *vars*'s mesh.

`__module__ = 'fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer'`

class `fipy.viewers.Matplotlib2DViewer` (*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DViewer* plots a 2D *CellVariable* using *Matplotlib*.

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...         + ((0.5, ), (0.2, ))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=numerix.sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Matplotlib2DViewer test")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

`__init__` (*vars*, *title=None*, *limits={}*, *cmap=None*, *colorbar='vertical'*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable*) – the *Variable* to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **cmap** (*Colormap*, *optional*) – the *Colormap*. Defaults to *matplotlib.cm.jet*
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **colorbar** (*bool*, *optional*) – plot a color bar in specified orientation if not *None*
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlib2DViewer'
```

```
class fipy.viewers.MatplotlibVectorViewer (vars, title=None, scale=None, sparsity=None,
                                           log=False, limits={}, axes=None, fi-
                                           gaspect='auto', **kwlimits)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays a vector plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using *Matplotlib*

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> for sparsity in numerix.arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5,), (0.2,))))
```

(continues on next page)

(continued from previous page)

```

>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).grad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

```

>>> viewer = MatplotlibVectorViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **scale** (*float*, *optional*) – if not *None*, scale all arrow lengths by this value
- **sparsity** (*int*, *optional*) – if not *None*, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.

__init__ (*vars*, *title=None*, *scale=None*, *sparsity=None*, *log=False*, *limits={}*, *axes=None*, *figaspect='auto'*, ***kwlimits*)

Creates a *Matplotlib2DViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **scale** (*float*, *optional*) – if not *None*, scale all arrow lengths by this value
- **sparsity** (*int*, *optional*) – if not *None*, then this number of arrows will be randomly chosen (weighted by the cell volume or face area)
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude

- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the Variable's mesh.

```
__module__ = 'fipy.viewers.matplotlibViewer.matplotlibVectorViewer'
```

```
quiver (sparsity=None, scale=None)
```

```
class fipy.viewers.MatplotlibStreamViewer (vars, title=None, log=False, limits={},
                                             axes=None, figaspect='auto', density=1,
                                             linewidth=None, color=None, cmap=None,
                                             norm=None, arrowsize=1, arrowstyle='->',
                                             minlength=0.1, **kwlimits)
```

Bases: `fipy.viewers.matplotlibViewer.matplotlib2DViewer`, `AbstractMatplotlib2DViewer`

Displays a stream plot of a 2D rank-1 *CellVariable* or *FaceVariable* object using [Matplotlib](#)

One issue is that this *Viewer* relies on `scipy.interpolate.griddata`, which interpolates on the convex hull of the data. The results is that streams are plotted across any concavities in the mesh.

Another issue is that it does not seem possible to remove the streams without calling `cla()`, which means that different set of streams cannot be overlaid.

```
>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).grad,
...                                title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                                title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=1.)
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).grad,
...                                title="MatplotlibStreamViewer test")
```

(continues on next page)

(continued from previous page)

```
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibStreamViewer(vars=numerix.sin(k * xyVar).faceGrad,
...                                 title="MatplotlibStreamViewer test")
>>> for kval in numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Creates a *MatplotlibStreamViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.
- **density** (*float* or (*float*, *float*), *optional*) – Controls the closeness of streamlines. When *density* = 1, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (*density_x*, *density_y*).
- **linewidth** (*array_like* or *CellVariable* or *FaceVariable*, *optional*) – The width of the stream lines. With a rank-0 *CellVariable* or *FaceVariable* the line width can be varied across the grid. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **color** (*str* or *CellVariable* or *FaceVariable*, *optional*) – The streamline color as a matplotlib color code or a field of numbers. If given a rank-0 *CellVariable* or *FaceVariable*, its values are converted to colors using *cmap* and *norm*. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **cmap** (*Colormap*, *optional*) – Colormap used to plot streamlines and arrows. This is only used if *color* is a *MeshVariable*.
- **norm** (*Normalize*, *optional*) – Normalize object used to scale luminance data to 0, 1. If *None*, stretch (min, max) to (0, 1). Only necessary when *color* is a *MeshVariable*.
- **arrowsize** (*float*, *optional*) – Scaling factor for the arrow size.
- **arrowstyle** (*str*, *optional*) – Arrow style specification. See *~matplotlib.patches.FancyArrowPatch*.

- **minlength** (*float*, *optional*) – Minimum length of streamline in axes coordinates.

__init__ (*vars*, *title=None*, *log=False*, *limits={}*, *axes=None*, *figaspect='auto'*, *density=1*, *linewidth=None*, *color=None*, *cmap=None*, *norm=None*, *arrowsize=1*, *arrowstyle='->'*, *minlength=0.1*, ***kwlimits*)
Creates a *MatplotlibStreamViewer*.

Parameters

- **vars** (*CellVariable* or *FaceVariable*) – rank-1 *Variable* to display
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **log** (*bool*, *optional*) – if *True*, arrow length goes at the base-10 logarithm of the magnitude
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin**, **xmax**, **ymin**, **ymax**, **datamin**, **datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **axes** (*Axes*, *optional*) – if not *None*, *vars* will be plotted into this Matplotlib *Axes* object
- **figaspect** (*float*, *optional*) – desired aspect ratio of figure. If arg is a number, use that aspect ratio. If arg is *auto*, the aspect ratio will be determined from the *Variable*'s mesh.
- **density** (*float* or (*float*, *float*), *optional*) – Controls the closeness of streamlines. When *density* = 1, the domain is divided into a 30x30 grid. *density* linearly scales this grid. Each cell in the grid can have, at most, one traversing streamline. For different densities in each direction, use a tuple (*density_x*, *density_y*).
- **linewidth** (*array_like* or *CellVariable* or *FaceVariable*, *optional*) – The width of the stream lines. With a rank-0 *CellVariable* or *FaceVariable* the line width can be varied across the grid. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **color** (*str* or *CellVariable* or *FaceVariable*, *optional*) – The streamline color as a matplotlib color code or a field of numbers. If given a rank-0 *CellVariable* or *FaceVariable*, its values are converted to colors using *cmap* and *norm*. The *MeshVariable* must have the same type and be defined on the same *Mesh* as *vars*.
- **cmap** (*Colormap*, *optional*) – Colormap used to plot streamlines and arrows. This is only used if *color* is a *MeshVariable*.
- **norm** (*Normalize*, *optional*) – Normalize object used to scale luminance data to 0, 1. If *None*, stretch (min, max) to (0, 1). Only necessary when *color* is a *MeshVariable*.
- **arrowsize** (*float*, *optional*) – Scaling factor for the arrow size.
- **arrowstyle** (*str*, *optional*) – Arrow style specification. See *~matplotlib.patches.FancyArrowPatch*.
- **minlength** (*float*, *optional*) – Minimum length of streamline in axes coordinates.

__module__ = 'fipy.viewers.matplotlibViewer.matplotlibStreamViewer'

class *fipy.viewers.MayaviClient* (*vars*, *title=None*, *daemon_file=None*, *fps=1.0*, ***kwlimits*)
Bases: *fipy.viewers.viewer.AbstractViewer*

The *MayaviClient* uses the *Mayavi* python plotting package.

```

>>> from fipy import *
>>> mesh = Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = CellVariable(mesh=mesh, name="x", value=x)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=(numerix.sin(k * xVar), numerix.cos(k * xVar /
↳numerix.pi)),
...                       limits={'xmin': 10, 'xmax': 90},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> for kval in numerix.arange(0, 0.3, 0.03):
...     k.set_value(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

```

>>> from fipy import *
>>> mesh = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> from builtins import range
>>> for kval in range(10):
...     k.set_value(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

```

>>> from fipy import *
>>> mesh = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...        + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...        + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")
>>> from builtins import range
>>> for kval in range(10):
...     k.set_value(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

```

>>> from fipy import *
>>> mesh = Grid3D(nx=50, ny=100, nz=10, dx=0.1, dy=0.01, dz=0.1)
>>> x, y, z = mesh.cellCenters
>>> xyzVar = CellVariable(mesh=mesh, name="x y z", value=x * y * z)
>>> k = Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=numerix.sin(k * xyzVar),
...                       limits={'ymin': 0.1, 'ymax': 0.9},
...                       datamin=-0.9, datamax=2.0,
...                       title="MayaviClient test")

```

(continues on next page)

(continued from previous page)

```
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

Create a *MayaviClient*.

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **daemon_file** (*str*, *optional*) – the path to the script to run the separate Mayavi viewer process. Defaults to *fipy/viewers/mayaviViewer/mayaviDaemon.py*
- **fps** (*float*, *optional*) – frames per second to attempt to display

`__del__()`

`__init__(vars, title=None, daemon_file=None, fps=1.0, **kwlimits)`

Create a *MayaviClient*.

Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.
- **daemon_file** (*str*, *optional*) – the path to the script to run the separate Mayavi viewer process. Defaults to *fipy/viewers/mayaviViewer/mayaviDaemon.py*
- **fps** (*float*, *optional*) – frames per second to attempt to display

`__module__ = 'fipy.viewers.mayaviViewer.mayaviClient'`

`plot(filename=None)`

Update the display of the viewed variables.

Parameters `filename` (*str*) – If not *None*, the name of a file to save the image into.

class `fipy.viewers.MultiViewer` (*viewers*)

Bases: *fipy.viewers.viewer.AbstractViewer*

Treat a collection of different viewers (such for different 2D plots or 1D plots with different axes) as a single viewer that will *plot()* all subviewers simultaneously.

Parameters `viewers` (*list* of *~fipy.viewers.viewer.Viewer*) – the viewers to bind together

`__init__(viewers)`

Parameters `viewers` (*list* of *~fipy.viewers.viewer.Viewer*) – the viewers to bind together

`__module__ = 'fipy.viewers.multiViewer'`

plot()

Update the display of the viewed variables.

Parameters **filename** (*str*) – If not *None*, the name of a file to save the image into.

setLimits (*limits*=*{}*, ***kwlimits*)

Update the limits.

Parameters

- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.

class `fiPy.viewers.TSVViewer` (*vars*, *title*=*None*, *limits*=*{}*, ***kwlimits*)

Bases: `fiPy.viewers.viewer.AbstractViewer`

“Views” one or more variables in tab-separated-value format.

Output is a list of coordinates and variable values at each cell center.

File contents will be, e.g.:

```
title
x      y      ...    var0    var2    ...
0.0    0.0    ...    3.14    1.41    ...
1.0    0.0    ...    2.72    0.866   ...
:
:
```

Creates a *TSVViewer*.

Any cell centers that lie outside the limits provided will not be included. Any values that lie outside the *datamin* or *datamax* will be replaced with *nan*.

All variables must have the same mesh.

It tries to do something reasonable with rank-1 *CellVariable* and *FaceVariable* objects.

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

__init__ (*vars*, *title*=*None*, *limits*=*{}*, ***kwlimits*)

Creates a *TSVViewer*.

Any cell centers that lie outside the limits provided will not be included. Any values that lie outside the *datamin* or *datamax* will be replaced with *nan*.

All variables must have the same mesh.

It tries to do something reasonable with rank-1 *CellVariable* and *FaceVariable* objects.

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

`__module__ = 'fipy.viewers.tsviewer'`

plot (*filename=None*)

“plot” the coordinates and values of the variables to *filename*. If *filename* is not provided, “plots” to *stdout*.

```
>>> from fipy.meshes import Grid1D
>>> m = Grid1D(nx = 3, dx = 0.4)
>>> from fipy.variables.cellVariable import CellVariable
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, 5))
>>> TSVViewer(vars = (v, v.grad)).plot()
x      var      var_gauss_grad_x
0.2      0          2.5
0.6      2          6.25
1        5          3.75
```

```
>>> from fipy.meshes import Grid2D
>>> m = Grid2D(nx = 2, dx = .1, ny = 2, dy = 0.3)
>>> v = CellVariable(mesh = m, name = "var", value = (0, 2, -2, 5))
>>> TSVViewer(vars = (v, v.grad)).plot()
x      y      var      var_gauss_grad_x      var_gauss_grad_y
0.05    0.15    0          10      -3.3333333333333333
0.15    0.15    2          10          5
0.05    0.45   -2          35      -3.3333333333333333
0.15    0.45    5          35          5
```

Parameters filename (*str*) – If not *None*, the name of a file to save the image into.

`fipy.viewers.VTKViewer` (*vars*, *title=None*, *limits={}, **kwlimits*)

Generic function for creating a *VTKViewer*.

The *VTKViewer* factory will search the module tree and return an instance of the first *VTKViewer* it finds of the correct dimension and rank.

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

class `fipy.viewers.VTKCellViewer` (*vars*, *title=None*, *limits={}, **kwlimits*)

Bases: `fipy.viewers.vtkViewer.vtkViewer.VTKViewer`

Renders *CellVariable* data in VTK format

Creates a *VTKViewer*

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

```
__module__ = 'fipy.viewers.vtkViewer.vtkCellViewer'
```

```
class fipy.viewers.VTKFaceViewer (vars, title=None, limits={}, **kwlimits)
```

Bases: *fipy.viewers.vtkViewer.vtkViewer.VTKViewer*

Renders *_MeshVariable* data in VTK format

Creates a *VTKViewer*

Parameters

- **vars** (*CellVariable* or *FaceVariable* or *list*) – the *MeshVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*, *optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

```
__module__ = 'fipy.viewers.vtkViewer.vtkFaceViewer'
```

```
exception fipy.viewers.MeshDimensionError
```

Bases: *IndexError*

```
__module__ = 'fipy.viewers'
```

```
__weakref__
```

list of weak references to the object (if defined)

```
class fipy.viewers.DummyViewer (vars, title=None, **kwlimits)
```

Bases: *fipy.viewers.viewer.AbstractViewer*

Create a *AbstractViewer* object.

Parameters

- **vars** (*CellVariable* or *list*) – the *CellVariable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

```
__module__ = 'fipy.viewers'
```

```
plot (filename=None)
```

Update the display of the viewed variables.

Parameters filename (*str*) – If not *None*, the name of a file to save the image into.

```
fipy.viewers.Viewer (vars, title=None, limits={}, FIPY_VIEWER=None, **kwlimits)
```

Generic function for creating a *Viewer*.

The *Viewer* factory will search the module tree and return an instance of the first *Viewer* it finds that supports the dimensions of *vars*. Setting the *FIPY_VIEWER* environment variable to either *matplotlib*, *mayavi*, *tsv*, or *vtk* will specify the viewer.

The *kwlimits* or *limits* parameters can be used to constrain the view. For example:

```
Viewer(vars=some1Dvar, xmin=0.5, xmax=None, datamax=3)
```

or:

```
Viewer(vars=some1Dvar,
       limits={'xmin': 0.5, 'xmax': None, 'datamax': 3})
```

will return a viewer that displays a line plot from an *x* value of 0.5 up to the largest *x* value in the dataset. The data values will be truncated at an upper value of 3, but will have no lower limit.

Parameters

- **vars** (*CellVariable* or *list*) – the *Variable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **FIPY_VIEWER** – a specific viewer to attempt (possibly multiple times for multiple variables)
- **xmin, xmax, ymin, ymax, zmin, zmax, datamin, datamax** (*float*) – displayed range of data. A 1D *Viewer* will only use *xmin* and *xmax*, a 2D viewer will also use *ymin* and *ymax*, and so on. All viewers will use *datamin* and *datamax*. Any limit set to a (default) value of *None* will autoscale.

Bibliography

- [1] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996.
- [2] Scott Chacon. *Pro Git*. Apress, 2009. URL: <http://git-scm.com/book>.
- [3] W. J. Boettinger, J. A. Warren, C. Beckermann, and A. Karma. Phase-field simulation of solidification. *Annual Review of Materials Research*, 32:163–194, 2002. doi:10.1146/annurev.matsci.32.101901.155803.
- [4] L. Q. Chen. Phase-field models for microstructure evolution. *Annual Review of Materials Research*, 32:113–140, 2002. doi:10.1146/annurev.matsci.32.112001.132041.
- [5] G. B. McFadden. Phase-field models of solidification. *Contemporary Mathematics*, 306:107–145, 2002.
- [6] David M Saylor, Jonathan E Guyer, Daniel Wheeler, and James A Warren. Predicting microstructure development during casting of drug-eluting coatings. *Acta Biomaterialia*, 7(2):604–613, Jan 2011. doi:10.1016/j.actbio.2010.09.019.
- [7] Daniel Wheeler, James A. Warren, and William J. Boettinger. Modeling the early stages of reactive wetting. *Physical Review E*, 82(5):051601, Nov 2010. doi:10.1103/PhysRevE.82.051601.
- [8] C. M Hangarter, B. H Hamadani, J. E Guyer, H Xu, R Need, and D Josell. Three dimensionally structured interdigitated back contact thin film heterojunction solar cells. *Journal of Applied Physics*, 109(7):073514, Jan 2011. doi:10.1063/1.3561487.
- [9] D. Josell, D. Wheeler, W. H. Huber, and T. P. Moffat. Superconformal electrodeposition in submicron features. *Physical Review Letters*, 87(1):016102, 2001. doi:10.1103/PhysRevLett.87.016102.
- [10] James A. Warren, Ryo Kobayashi, Alexander E. Lobkovsky, and W. Craig Carter. Extending phase field models of solidification to polycrystalline materials. *Acta Materialia*, 51(20):6035–6058, 2003. doi:10.1016/S1359-6454(03)00388-4.
- [11] Guido van Rossum. *Python Tutorial*. URL: <http://docs.python.org/tut/>.
- [12] Mark Pilgrim. *Dive Into Python*. Apress, 2004. ISBN 1590593561. URL: <http://diveintopython.org>.
- [13] Guido van Rossum. *Python Reference Manual*. URL: <http://docs.python.org/ref/>.
- [14] Greg Ward. *Installing Python Modules*. URL: <http://docs.python.org/inst/>.
- [15] T. N. Croft. *Unstructured Mesh - Finite Volume Algorithms for Swirling, Turbulent Reacting Flows*. PhD thesis, University of Greenwich, 1998. URL: <http://gala.gre.ac.uk/id/eprint/6371/>.
- [16] John W. Cahn and John E. Hilliard. Free energy of a nonuniform system. I. Interfacial free energy. *Journal of Chemical Physics*, 28(2):258–267, 1958.
- [17] John W. Cahn. Free energy of a nonuniform system. II. Thermodynamic basis. *Journal of Chemical Physics*, 30(5):1121–1124, 1959.

- [18] John W. Cahn and John E. Hilliard. Free energy of a nonuniform system. III. Nucleation in a two-component incompressible fluid. *Journal of Chemical Physics*, 31(3):688–699, 1959.
- [19] K. R Elder, K Thornton, and J. J Hoyt. The kirkendall effect in the phase field crystal model. *Philosophical Magazine*, 91(1):151–164, Jan 2011. doi:10.1080/14786435.2010.506427.
- [20] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow*. Taylor and Francis, 1980.
- [21] H. K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics*. Longman Scientific and Technical, 1995.
- [22] C. Mattiussi. An analysis of finite volume, finite element, and finite difference methods using some concepts from algebraic topology. *Journal of Computational Physics*, 133:289–309, 1997. URL: <http://lis.epfl.ch/publications/JCP1997.pdf>.
- [23] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: the Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1999.
- [24] D. Wheeler, D. Josell, and T. P. Moffat. Modeling superconformal electrodeposition using the level set method. *Journal of The Electrochemical Society*, 150(5):C302–C310, 2003. doi:10.1149/1.1562598.
- [25] D. Josell, D. Wheeler, and T. P. Moffat. Gold superfill in submicrometer trenches: experiment and prediction. *Journal of The Electrochemical Society*, 153(1):C11–C18, 2006. doi:10.1149/1.2128765.
- [26] T. P. Moffat, D. Wheeler, S. K. Kim, and D. Josell. Curvature enhanced adsorbate coverage model for electrodeposition. *Journal of The Electrochemical Society*, 153(2):C127–C132, 2006. doi:10.1149/1.2165580.
- [27] A. A. Wheeler, W. J. Boettinger, and G. B. McFadden. Phase-field model for isothermal phase transitions in binary alloys. *Physical Review A*, 45(10):7424–7439, 1992.
- [28] J. A. Warren and W. J. Boettinger. Prediction of dendritic growth and microsegregation in a binary alloy using the phase field method. *Acta Metallurgica et Materialia*, 43(2):689–703, 1995.
- [29] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden. Phase field modeling of electrochemistry I: Equilibrium. *Physical Review E*, 69:021603, 2004. arXiv:cond-mat/0308173, doi:10.1103/PhysRevE.69.021603.
- [30] J. E. Guyer, W. J. Boettinger, J. A. Warren, and G. B. McFadden. Phase field modeling of electrochemistry II: Kinetics. *Physical Review E*, 69:021604, 2004. arXiv:cond-mat/0308179, doi:10.1103/PhysRevE.69.021604.
- [31] T. P. Moffat, D. Wheeler, and D. Josell. Superfilling and the curvature enhanced accelerator coverage mechanism. *The Electrochemical Society, Interface*, 13(4):46–52, 2004. URL: <http://www.electrochem.org/publications/interface/winter2004/IF12-04-Pg46.pdf>.
- [32] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, 1996.
- [33] C.-C. Rossow. A blended pressure/density based method for the computation of incompressible and compressible flows. *Journal of Computational Physics*, 185(2):375–398, 2003. doi:10.1016/S0021-9991(02)00059-1.

Python Module Index

e

examples.cahnHilliard.mesh2DCoupled, 203
examples.cahnHilliard.sphere, 206
examples.convection.exponential1D.mesh1D, 143
examples.convection.exponential1DSource.mesh1D, 145
examples.convection.robin, 146
examples.convection.source, 148
examples.diffusion.anisotropy, 141
examples.diffusion.circle, 130
examples.diffusion.coupled, 125
examples.diffusion.electrostatics, 134
examples.diffusion.mesh1D, 105
examples.diffusion.mesh20x20, 127
examples.diffusion.nthOrder.input4thOrder1D, 139
examples.flow.stokesCavity, 209
examples.levelSet.advection.circle, 201
examples.levelSet.advection.mesh1D, 199
examples.levelSet.distanceFunction.circle, 198
examples.levelSet.distanceFunction.mesh1D, 197
examples.phase.anisotropy, 174
examples.phase.binaryCoupled, 158
examples.phase.impingement.mesh20x20, 180
examples.phase.impingement.mesh40x1, 177
examples.phase.polyxtal, 184
examples.phase.polyxtalCoupled, 190
examples.phase.quaternary, 167
examples.phase.simple, 149
examples.reactiveWetting.liquidVapor1D, 215
examples.updating.update0_1to1_0, 226
examples.updating.update1_0to2_0, 222
examples.updating.update2_0to3_0, 221

f

fipy.boundaryConditions, 240

fipy.boundaryConditions.boundaryCondition, 237
fipy.boundaryConditions.constraint, 238
fipy.boundaryConditions.fixedFlux, 238
fipy.boundaryConditions.fixedValue, 238
fipy.boundaryConditions.nthOrderBoundaryCondition, 239
fipy.boundaryConditions.test, 239
fipy.matrices, 243
fipy.matrices.offsetSparseMatrix, 243
fipy.matrices.petscMatrix, 243
fipy.matrices.scipyMatrix, 243
fipy.matrices.sparseMatrix, 243
fipy.matrices.test, 243
fipy.meshes, 280
fipy.meshes.abstractMesh, 246
fipy.meshes.builders, 245
fipy.meshes.builders.abstractGridBuilder, 245
fipy.meshes.builders.grid1DBuilder, 245
fipy.meshes.builders.grid2DBuilder, 245
fipy.meshes.builders.grid3DBuilder, 245
fipy.meshes.builders.periodicGrid1DBuilder, 245
fipy.meshes.builders.utilityClasses, 245
fipy.meshes.cylindricalGrid1D, 253
fipy.meshes.cylindricalGrid2D, 253
fipy.meshes.cylindricalNonUniformGrid1D, 253
fipy.meshes.cylindricalNonUniformGrid2D, 255
fipy.meshes.cylindricalUniformGrid1D, 256
fipy.meshes.cylindricalUniformGrid2D, 256
fipy.meshes.factoryMeshes, 257
fipy.meshes.gmshMesh, 259
fipy.meshes.grid1D, 265
fipy.meshes.grid2D, 265
fipy.meshes.grid3D, 265
fipy.meshes.mesh, 265

fipy.meshes.mesh1D, 266
 fipy.meshes.mesh2D, 267
 fipy.meshes.nonUniformGrid1D, 269
 fipy.meshes.nonUniformGrid2D, 269
 fipy.meshes.nonUniformGrid3D, 270
 fipy.meshes.periodicGrid1D, 271
 fipy.meshes.periodicGrid2D, 272
 fipy.meshes.periodicGrid3D, 273
 fipy.meshes.representations, 245
 fipy.meshes.representations.abstractRepresentation, 245
 fipy.meshes.representations.gridRepresentation, 245
 fipy.meshes.representations.meshRepresentation, 245
 fipy.meshes.skewedGrid2D, 276
 fipy.meshes.test, 276
 fipy.meshes.topologies, 246
 fipy.meshes.topologies.abstractTopology, 246
 fipy.meshes.topologies.gridTopology, 246
 fipy.meshes.topologies.meshTopology, 246
 fipy.meshes.tri2D, 276
 fipy.meshes.uniformGrid, 277
 fipy.meshes.uniformGrid1D, 278
 fipy.meshes.uniformGrid2D, 278
 fipy.meshes.uniformGrid3D, 279
 fipy.solvers, 321
 fipy.solvers.petsc, 298
 fipy.solvers.petsc.comms, 294
 fipy.solvers.petsc.comms.parallelPETScCommWrapper, 293
 fipy.solvers.petsc.comms.petscCommWrapper, 294
 fipy.solvers.petsc.comms.serialPETScCommWrapper, 294
 fipy.solvers.petsc.dummySolver, 294
 fipy.solvers.petsc.linearBicgSolver, 295
 fipy.solvers.petsc.linearCGSSolver, 295
 fipy.solvers.petsc.linearGMRESSolver, 295
 fipy.solvers.petsc.linearLUSolver, 296
 fipy.solvers.petsc.linearPCGSolver, 296
 fipy.solvers.petsc.petscKrylovSolver, 297
 fipy.solvers.petsc.petscSolver, 297
 fipy.solvers.pyAMG, 303
 fipy.solvers.pyAMG.linearCGSSolver, 300
 fipy.solvers.pyAMG.linearGeneralSolver, 301
 fipy.solvers.pyAMG.linearGMRESSolver, 301
 fipy.solvers.pyAMG.linearLUSolver, 302
 fipy.solvers.pyAMG.linearPCGSolver, 302
 fipy.solvers.pyAMG.preconditioners, 300
 fipy.solvers.pyAMG.preconditioners.smoothedAggregation, 300
 fipy.solvers.pyamgx, 310
 fipy.solvers.pyamgx.aggregationAMGSolver, 305
 fipy.solvers.pyamgx.classicalAMGSolver, 306
 fipy.solvers.pyamgx.linearBiCGStabSolver, 307
 fipy.solvers.pyamgx.linearCGSolver, 307
 fipy.solvers.pyamgx.linearFGMRESSolver, 308
 fipy.solvers.pyamgx.linearGMRESSolver, 309
 fipy.solvers.pyamgx.linearLUSolver, 309
 fipy.solvers.pyamgx.preconditioners, 305
 fipy.solvers.pyamgx.preconditioners.preconditioners, 305
 fipy.solvers.pyamgx.pyAMGXSolver, 310
 fipy.solvers.pyamgx.smoothers, 305
 fipy.solvers.pyamgx.smoothers.smoothers, 305
 fipy.solvers.scipy, 316
 fipy.solvers.scipy.linearBicgstabSolver, 313
 fipy.solvers.scipy.linearCGSSolver, 314
 fipy.solvers.scipy.linearGMRESSolver, 314
 fipy.solvers.scipy.linearLUSolver, 315
 fipy.solvers.scipy.linearPCGSolver, 315
 fipy.solvers.scipy.scipyKrylovSolver, 316
 fipy.solvers.scipy.scipySolver, 316
 fipy.solvers.solver, 319
 fipy.solvers.test, 321
 fipy.steppers, 328
 fipy.steppers.pidStepper, 327
 fipy.steppers.pseudoRKQSStepper, 327
 fipy.steppers.stepper, 328
 fipy.terms, 354
 fipy.terms.abstractBinaryTerm, 331
 fipy.terms.abstractConvectionTerm, 331
 fipy.terms.abstractDiffusionTerm, 331
 fipy.terms.abstractUpwindConvectionTerm, 331
 fipy.terms.advectionTerm, 331
 fipy.terms.asymmetricConvectionTerm, 334
 fipy.terms.binaryTerm, 334
 fipy.terms.cellTerm, 334
 fipy.terms.centralDiffConvectionTerm, 334
 fipy.terms.coupledBinaryTerm, 336
 fipy.terms.diffusionTerm, 336

Python Module Index	577
----------------------------	------------

[519](#)
 fipy.viewers.matplotlibViewer.matplotlib2DContourViewer,
[521](#)
 fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer,
[522](#)
 fipy.viewers.matplotlibViewer.matplotlib2DGridViewer,
[524](#)
 fipy.viewers.matplotlibViewer.matplotlib2DViewer,
[525](#)
 fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer,
[527](#)
 fipy.viewers.matplotlibViewer.matplotlibStreamViewer,
[527](#)
 fipy.viewers.matplotlibViewer.matplotlibVectorViewer,
[530](#)
 fipy.viewers.matplotlibViewer.matplotlibViewer,
[532](#)
 fipy.viewers.matplotlibViewer.test, [534](#)
 fipy.viewers.mayaviViewer, [548](#)
 fipy.viewers.mayaviViewer.mayaviClient,
[545](#)
 fipy.viewers.mayaviViewer.mayaviDaemon,
[547](#)
 fipy.viewers.mayaviViewer.test, [548](#)
 fipy.viewers.multiViewer, [553](#)
 fipy.viewers.test, [553](#)
 fipy.viewers.testinteractive, [553](#)
 fipy.viewers.tsvViewer, [553](#)
 fipy.viewers.viewer, [555](#)
 fipy.viewers.vtkViewer, [552](#)
 fipy.viewers.vtkViewer.test, [550](#)
 fipy.viewers.vtkViewer.vtkCellViewer,
[550](#)
 fipy.viewers.vtkViewer.vtkFaceViewer,
[551](#)
 fipy.viewers.vtkViewer.vtkViewer, [551](#)

p

package.subpackage, [236](#)
 package.subpackage.base, [233](#)
 package.subpackage.object, [235](#)

Index

Symbols

`:math:\pi`, 175, 180, 181

`__abs__` (*fipy.tools.PhysicalField* method), 418

`__abs__` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 385

`__abs__` (*fipy.variables.Variable* method), 479

`__abs__` (*fipy.variables.variable.Variable* method), 468

`__add__` (*fipy.meshes.abstractMesh.AbstractMesh* method), 246

`__add__` (*fipy.terms.term.Term* method), 347

`__add__` (*fipy.tools.PhysicalField* method), 418

`__add__` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 386

`__add__` (*fipy.variables.Variable* method), 479

`__add__` (*fipy.variables.variable.Variable* method), 468

`__and__` (*fipy.terms.term.Term* method), 347

`__and__` (*fipy.variables.Variable* method), 479

`__and__` (*fipy.variables.variable.Variable* method), 468

`__array__` (*fipy.tools.PhysicalField* method), 418

`__array__` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 386

`__array__` (*fipy.variables.Variable* method), 479

`__array__` (*fipy.variables.variable.Variable* method), 469

`__array_priority__` (*fipy.tools.PhysicalField* attribute), 419

`__array_priority__` (*fipy.tools.dimensions.physicalField.PhysicalField* attribute), 386

`__array_priority__` (*fipy.variables.Variable* attribute), 480

`__array_priority__` (*fipy.variables.variable.Variable* attribute), 469

`__array_wrap__` (*fipy.tools.PhysicalField* method), 419

`__array_wrap__` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 386

`__array_wrap__` (*fipy.variables.Variable* method), 480

`__array_wrap__` (*fipy.variables.variable.Variable* method), 469

`__base_traits__` (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* attribute), 547

`__bool__` (*fipy.tools.PhysicalField* method), 419

`__bool__` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 386

`__bool__` (*fipy.variables.Variable* method), 480

`__bool__` (*fipy.variables.variable.Variable* method), 469

`__call__` (*fipy.variables.CellVariable* method), 489

`__call__` (*fipy.variables.Variable* method), 480

`__call__` (*fipy.variables.cellVariable.CellVariable* method), 438

`__call__` (*fipy.variables.variable.Variable* method), 469

`__class_traits__` (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* attribute), 547

`__del__` (*fipy.viewers.MayaviClient* method), 568

`__del__` (*fipy.viewers.mayaviViewer.MayaviClient* method), 549

`__del__` (*fipy.viewers.mayaviViewer.mayaviClient.MayaviClient* method), 546

`__del__` (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 547

`__dict__` (*fipy.boundaryConditions.Constraint* attribute), 240

`__dict__` (*fipy.boundaryConditions.boundaryCondition.BoundaryCondition* attribute), 237

`__dict__` (*fipy.boundaryConditions.constraint.Constraint* attribute), 238

`__dict__` (*fipy.meshes.abstractMesh.AbstractMesh* attribute), 248

`__dict__` (*fipy.solvers.Solver* attribute), 322

`__dict__` (*fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner* attribute), 300

`__dict__` (*fipy.solvers.solver.Solver* attribute), 321

`__dict__` (*fipy.steppers stepper.Stepper* attribute), 328

`__dict__` (*fipy.terms.term.Term* attribute), 347

`__dict__` (*fipy.tools.PhysicalField* attribute), 419

`__dict__` (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* attribute), 381
`__dict__` (*fipy.tools.dimensions.physicalField.PhysicalField* attribute), 387
`__dict__` (*fipy.tools.dimensions.physicalField.PhysicalUnit* attribute), 401
`__dict__` (*fipy.variables.Variable* attribute), 480
`__dict__` (*fipy.variables.variable.Variable* attribute), 470
`__dict__` (*fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer* attribute), 527
`__dict__` (*fipy.viewers.viewer.AbstractViewer* attribute), 555
`__dict__` (*package.subpackage.base.Base* attribute), 234
`__div__` () (*fipy.meshes.abstractMesh.AbstractMesh* method), 248
`__div__` () (*fipy.terms.term.Term* method), 347
`__div__` () (*fipy.tools.PhysicalField* method), 419
`__div__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 387
`__div__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 401
`__div__` () (*fipy.variables.Variable* method), 480
`__div__` () (*fipy.variables.variable.Variable* method), 470
`__enter__` () (*fipy.solvers.Solver* method), 322
`__enter__` () (*fipy.solvers.solver.Solver* method), 321
`__eq__` () (*fipy.terms.term.Term* method), 347
`__eq__` () (*fipy.tools.PhysicalField* method), 419
`__eq__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 387
`__eq__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 402
`__eq__` () (*fipy.variables.Variable* method), 480
`__eq__` () (*fipy.variables.variable.Variable* method), 470
`__exit__` () (*fipy.solvers.Solver* method), 322
`__exit__` () (*fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver* method), 240
`__exit__` () (*fipy.solvers.solver.Solver* method), 321
`__float__` () (*fipy.tools.PhysicalField* method), 419
`__float__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 387
`__float__` () (*fipy.variables.Variable* method), 480
`__float__` () (*fipy.variables.variable.Variable* method), 470
`__ge__` () (*fipy.tools.PhysicalField* method), 420
`__ge__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 387
`__ge__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 402
`__ge__` () (*fipy.variables.Variable* method), 480
`__ge__` () (*fipy.variables.variable.Variable* method),
`__getitem__` () (*fipy.tools.PhysicalField* method),
`__getitem__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 387
`__getitem__` () (*fipy.variables.Variable* method), 481
`__getitem__` () (*fipy.variables.variable.Variable* method), 470
`__getstate__` () (*fipy.meshes.abstractMesh.AbstractMesh* method), 248
`__getstate__` () (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 381
`__getstate__` () (*fipy.variables.CellVariable* method), 490
`__getstate__` () (*fipy.variables.Variable* method), 481
`__getstate__` () (*fipy.variables.cellVariable.CellVariable* method), 438
`__getstate__` () (*fipy.variables.variable.Variable* method), 470
`__gt__` () (*fipy.tools.PhysicalField* method), 420
`__gt__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 388
`__gt__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 402
`__gt__` () (*fipy.variables.Variable* method), 481
`__gt__` () (*fipy.variables.variable.Variable* method), 470
`__hash__` (*fipy.tools.dimensions.physicalField.PhysicalUnit* attribute), 402
`__hash__` () (*fipy.terms.term.Term* method), 347
`__hash__` () (*fipy.tools.PhysicalField* method), 421
`__hash__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 388
`__hash__` () (*fipy.variables.Variable* method), 481
`__hash__` () (*fipy.variables.variable.Variable* method), 471
`__init__` () (*fipy.boundaryConditions.Constraint* method), 240
`__init__` () (*fipy.boundaryConditions.FixedFlux* method), 240
`__init__` () (*fipy.boundaryConditions.NthOrderBoundaryCondition* method), 241
`__init__` () (*fipy.boundaryConditions.boundaryCondition.BoundaryCondition* method), 237
`__init__` () (*fipy.boundaryConditions.constraint.Constraint* method), 238
`__init__` () (*fipy.boundaryConditions.fixedFlux.FixedFlux* method), 238
`__init__` () (*fipy.boundaryConditions.nthOrderBoundaryCondition.NthOrderBoundaryCondition* method), 239
`__init__` () (*fipy.meshes.Gmsh2D* method), 291
`__init__` () (*fipy.meshes.Gmsh2DIn3DSpace* method), 291

`__init__()` (*fipy.meshes.Gmsh3D method*), 291
`__init__()` (*fipy.meshes.GmshGrid2D method*), 292
`__init__()` (*fipy.meshes.GmshGrid3D method*), 292
`__init__()` (*fipy.meshes.PeriodicGrid1D method*), 282
`__init__()` (*fipy.meshes.SkewedGrid2D method*), 286
`__init__()` (*fipy.meshes.Tri2D method*), 286
`__init__()` (*fipy.meshes.abstractMesh.AbstractMesh method*), 248
`__init__()` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D method*), 254
`__init__()` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D method*), 255
`__init__()` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D method*), 256
`__init__()` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D method*), 257
`__init__()` (*fipy.meshes.gmshMesh.Gmsh2D method*), 263
`__init__()` (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace method*), 263
`__init__()` (*fipy.meshes.gmshMesh.Gmsh3D method*), 264
`__init__()` (*fipy.meshes.gmshMesh.GmshGrid2D method*), 264
`__init__()` (*fipy.meshes.gmshMesh.GmshGrid3D method*), 264
`__init__()` (*fipy.meshes.mesh.Mesh method*), 265
`__init__()` (*fipy.meshes.mesh1D.Mesh1D method*), 266
`__init__()` (*fipy.meshes.mesh2D.Mesh2D method*), 267
`__init__()` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D method*), 269
`__init__()` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D method*), 270
`__init__()` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D method*), 270
`__init__()` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D method*), 271
`__init__()` (*fipy.meshes.skewedGrid2D.SkewedGrid2D method*), 276
`__init__()` (*fipy.meshes.tri2D.Tri2D method*), 277
`__init__()` (*fipy.meshes.uniformGrid1D.UniformGrid1D method*), 278
`__init__()` (*fipy.meshes.uniformGrid2D.UniformGrid2D method*), 278
`__init__()` (*fipy.meshes.uniformGrid3D.UniformGrid3D method*), 279
`__init__()` (*fipy.solvers.LinearLUSolver method*), 323
`__init__()` (*fipy.solvers.Solver method*), 322
`__init__()` (*fipy.solvers.SolverConvergenceWarning method*), 321
`__init__()` (*fipy.solvers.petsc.LinearLUSolver method*), 298
`__init__()` (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper method*), 293
`__init__()` (*fipy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper method*), 294
`__init__()` (*fipy.solvers.petsc.comms.serialPETScCommWrapper.SerialPETScCommWrapper method*), 294
`__init__()` (*fipy.solvers.petsc.linearLUSolver.LinearLUSolver method*), 296
`__init__()` (*fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver method*), 297
`__init__()` (*fipy.solvers.petsc.petscSolver.PETScSolver method*), 297
`__init__()` (*fipy.solvers.pyAMG.LinearCGSSolver method*), 303
`__init__()` (*fipy.solvers.pyAMG.LinearGMRESSolver method*), 303
`__init__()` (*fipy.solvers.pyAMG.LinearPCGSolver method*), 304
`__init__()` (*fipy.solvers.pyAMG.linearCGSSolver.LinearCGSSolver method*), 300
`__init__()` (*fipy.solvers.pyAMG.linearGMRESSolver.LinearGMRESSolver method*), 301
`__init__()` (*fipy.solvers.pyAMG.linearPCGSolver.LinearPCGSolver method*), 302
`__init__()` (*fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner method*), 300
`__init__()` (*fipy.solvers.pyamgx.AggregationAMGSolver method*), 312
`__init__()` (*fipy.solvers.pyamgx.LinearBiCGStabSolver method*), 312
`__init__()` (*fipy.solvers.pyamgx.LinearCGSolver method*), 311
`__init__()` (*fipy.solvers.pyamgx.LinearFGMRESSolver method*), 311
`__init__()` (*fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver method*), 305
`__init__()` (*fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver method*), 306
`__init__()` (*fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver method*), 307
`__init__()` (*fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver method*), 307
`__init__()` (*fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver method*), 308
`__init__()` (*fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver method*), 309
`__init__()` (*fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver method*), 310
`__init__()` (*fipy.solvers.scipy.LinearBicgstabSolver method*), 317
`__init__()` (*fipy.solvers.scipy.LinearCGSSolver method*), 316

`__init__()` (*fipy.solvers.scipy.LinearGMRESSolver* method), 388
`__init__()` (*fipy.solvers.scipy.LinearPCGSolver* method), 317
`__init__()` (*fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver* method), 314
`__init__()` (*fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver* method), 314
`__init__()` (*fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver* method), 314
`__init__()` (*fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver* method), 315
`__init__()` (*fipy.solvers.solver.Solver* method), 321
`__init__()` (*fipy.solvers.solver.SolverConvergenceWarning* method), 319
`__init__()` (*fipy.steps.pidStepper.PIDStepper* method), 327
`__init__()` (*fipy.steps.pseudoRKQSSolver.PseudoRKQSSolver* method), 328
`__init__()` (*fipy.steps.stepper.Stepper* method), 328
`__init__()` (*fipy.terms.AbstractBaseClassError* method), 354
`__init__()` (*fipy.terms.ExplicitVariableError* method), 354
`__init__()` (*fipy.terms.FirstOrderAdvectionTerm* method), 356, 372
`__init__()` (*fipy.terms.IncorrectSolutionVariable* method), 355
`__init__()` (*fipy.terms.ResidualTerm* method), 362
`__init__()` (*fipy.terms.SolutionVariableNumberError* method), 354
`__init__()` (*fipy.terms.SolutionVariableRequiredError* method), 355
`__init__()` (*fipy.terms.TermMultiplyError* method), 354
`__init__()` (*fipy.terms.VectorCoeffError* method), 354
`__init__()` (*fipy.terms.cellTerm.CellTerm* method), 334
`__init__()` (*fipy.terms.faceTerm.FaceTerm* method), 340
`__init__()` (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 342
`__init__()` (*fipy.terms.residualTerm.ResidualTerm* method), 346
`__init__()` (*fipy.terms.sourceTerm.SourceTerm* method), 346
`__init__()` (*fipy.terms.term.Term* method), 347
`__init__()` (*fipy.tools.PhysicalField* method), 421
`__init__()` (*fipy.tools.Vitals* method), 433
`__init__()` (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 381
`__init__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 558
`__init__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 402
`__init__()` (*fipy.tools.vitals.Vitals* method), 417
`__init__()` (*fipy.variables.BetaNoiseVariable* method), 501
`__init__()` (*fipy.variables.CellVariable* method), 490
`__init__()` (*fipy.variables.DistanceVariable* method),
`__init__()` (*fipy.variables.ExponentialNoiseVariable* method),
`__init__()` (*fipy.variables.GammaNoiseVariable* method), 505
`__init__()` (*fipy.variables.GaussianNoiseVariable* method), 507
`__init__()` (*fipy.variables.HistogramVariable* method), 509
`__init__()` (*fipy.variables.ScharfetterGummelFaceVariable* method), 498
`__init__()` (*fipy.variables.SurfactantConvectionVariable* method), 512
`__init__()` (*fipy.variables.SurfactantVariable* method), 510
`__init__()` (*fipy.variables.UniformNoiseVariable* method), 508
`__init__()` (*fipy.variables.Variable* method), 481
`__init__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 437
`__init__()` (*fipy.variables.cellVariable.CellVariable* method), 438
`__init__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 447
`__init__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 450
`__init__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 454
`__init__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 457
`__init__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 457
`__init__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 460
`__init__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 461
`__init__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 462
`__init__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 464
`__init__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 467
`__init__()` (*fipy.variables.variable.Variable* method),
`__init__()` (*fipy.viewers.Matplotlib1DViewer* method), 558

`__init__()` (*fipy.viewers.Matplotlib2DGridContourViewer* method), 551
`__init__()` (*fipy.viewers.Matplotlib2DGridViewer* method), 559
`__init__()` (*fipy.viewers.Matplotlib2DViewer* method), 561
`__init__()` (*fipy.viewers.MatplotlibStreamViewer* method), 566
`__init__()` (*fipy.viewers.MatplotlibVectorViewer* method), 563
`__init__()` (*fipy.viewers.MayaviClient* method), 568
`__init__()` (*fipy.viewers.MultiViewer* method), 568
`__init__()` (*fipy.viewers.TSVViewer* method), 569
`__init__()` (*fipy.viewers.matplotlibViewer.Matplotlib1DViewer* method), 536
`__init__()` (*fipy.viewers.matplotlibViewer.Matplotlib2DGridContourViewer* method), 538
`__init__()` (*fipy.viewers.matplotlibViewer.Matplotlib2DGridViewer* method), 537
`__init__()` (*fipy.viewers.matplotlibViewer.Matplotlib2DViewer* method), 540
`__init__()` (*fipy.viewers.matplotlibViewer.MatplotlibStreamViewer* method), 544
`__init__()` (*fipy.viewers.matplotlibViewer.MatplotlibVectorViewer* method), 541
`__init__()` (*fipy.viewers.matplotlibViewer.matplotlib1DViewer* method), 520
`__init__()` (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer* method), 522
`__init__()` (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer* method), 523
`__init__()` (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer* method), 524
`__init__()` (*fipy.viewers.matplotlibViewer.matplotlib2DViewer* method), 526
`__init__()` (*fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer* method), 527
`__init__()` (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer* method), 529
`__init__()` (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer* method), 531
`__init__()` (*fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer* method), 533
`__init__()` (*fipy.viewers.mayaviViewer.MayaviClient* method), 549
`__init__()` (*fipy.viewers.mayaviViewer.mayaviClient.MayaviClient* method), 546
`__init__()` (*fipy.viewers.multiViewer.MultiViewer* method), 553
`__init__()` (*fipy.viewers.tsvViewer.TSVViewer* method), 554
`__init__()` (*fipy.viewers.viewer.AbstractViewer* method), 555
`__init__()` (*fipy.viewers.vtkViewer.vtkViewer.VTKViewer* method), 551
`__init__()` (*package.subpackage.base.Base* method), 234
`__init__()` (*package.subpackage.object.Object* method), 235
`__instance_traits__` (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* attribute), 547
`__int__()` (*fipy.variables.Variable* method), 482
`__int__()` (*fipy.variables.variable.Variable* method), 471
`__invert__()` (*fipy.variables.Variable* method), 482
`__invert__()` (*fipy.variables.variable.Variable* method), 471
`__iter__()` (*fipy.variables.Variable* method), 482
`__iter__()` (*fipy.variables.variable.Variable* method), 471
`__le__()` (*fipy.tools.PhysicalField* method), 421
`__le__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 389
`__le__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 402
`__le__()` (*fipy.variables.Variable* method), 482
`__le__()` (*fipy.variables.variable.Variable* method), 471
`__len__()` (*fipy.tools.PhysicalField* method), 421
`__len__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 389
`__len__()` (*fipy.variables.Variable* method), 482
`__len__()` (*fipy.variables.variable.Variable* method), 472
`__lt__()` (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* attribute), 547
`__lt__()` (*fipy.tools.PhysicalField* method), 421
`__lt__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 389
`__lt__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 402
`__lt__()` (*fipy.variables.Variable* method), 482
`__lt__()` (*fipy.variables.variable.Variable* method), 471
`__mod__()` (*fipy.tools.PhysicalField* method), 422
`__mod__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 389
`__mod__()` (*fipy.variables.Variable* method), 483
`__mod__()` (*fipy.variables.variable.Variable* method), 472
`__module__` (*fipy.boundaryConditions.Constraint* attribute), 240
`__module__` (*fipy.boundaryConditions.FixedFlux* attribute), 240
`__module__` (*fipy.boundaryConditions.FixedValue* attribute), 240

<code>__module__</code> (<code>fipy.boundaryConditions.NthOrderBoundaryCondition</code> attribute), 263	<code>__module__</code> (<code>fipy.meshes.gmshMesh.Gmsh2DIn3DSpace</code> attribute), 263
<code>__module__</code> (<code>fipy.boundaryConditions.boundaryCondition.BoundaryCondition</code> attribute), 237	<code>__module__</code> (<code>fipy.meshes.gmshMesh.Gmsh3D</code> attribute), 264
<code>__module__</code> (<code>fipy.boundaryConditions.constraint.Constraint</code> attribute), 238	<code>__module__</code> (<code>fipy.meshes.gmshMesh.GmshGrid2D</code> attribute), 264
<code>__module__</code> (<code>fipy.boundaryConditions.fixedFlux.FixedFlux</code> attribute), 238	<code>__module__</code> (<code>fipy.meshes.gmshMesh.GmshGrid3D</code> attribute), 264
<code>__module__</code> (<code>fipy.boundaryConditions.fixedValue.FixedValue</code> attribute), 239	<code>__module__</code> (<code>fipy.meshes.mesh.Mesh</code> attribute), 265
<code>__module__</code> (<code>fipy.boundaryConditions.nthOrderBoundaryCondition.NthOrderBoundaryCondition</code> attribute), 239	<code>__module__</code> (<code>fipy.meshes.mesh1D.Mesh1D</code> attribute), 267
<code>__module__</code> (<code>fipy.meshes.Gmsh2D</code> attribute), 291	<code>__module__</code> (<code>fipy.meshes.mesh2D.Mesh2D</code> attribute), 267
<code>__module__</code> (<code>fipy.meshes.Gmsh2DIn3DSpace</code> attribute), 291	<code>__module__</code> (<code>fipy.meshes.nonUniformGrid1D.NonUniformGrid1D</code> attribute), 269
<code>__module__</code> (<code>fipy.meshes.Gmsh3D</code> attribute), 291	<code>__module__</code> (<code>fipy.meshes.nonUniformGrid2D.NonUniformGrid2D</code> attribute), 270
<code>__module__</code> (<code>fipy.meshes.GmshGrid2D</code> attribute), 292	<code>__module__</code> (<code>fipy.meshes.nonUniformGrid3D.NonUniformGrid3D</code> attribute), 270
<code>__module__</code> (<code>fipy.meshes.GmshGrid3D</code> attribute), 292	<code>__module__</code> (<code>fipy.meshes.periodicGrid1D.PeriodicGrid1D</code> attribute), 271
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid1D</code> attribute), 282	<code>__module__</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2D</code> attribute), 273
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid2D</code> attribute), 283	<code>__module__</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight</code> attribute), 273
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid2DLeftRight</code> attribute), 283	<code>__module__</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom</code> attribute), 273
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid2DTopBottom</code> attribute), 284	<code>__module__</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3D</code> attribute), 274
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid3D</code> attribute), 285	<code>__module__</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack</code> attribute), 275
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid3DFrontBack</code> attribute), 285	<code>__module__</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight</code> attribute), 274
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid3DLeftRight</code> attribute), 285	<code>__module__</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack</code> attribute), 275
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid3DLeftRightFrontBack</code> attribute), 285	<code>__module__</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom</code> attribute), 275
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid3DLeftRightTopBottom</code> attribute), 285	<code>__module__</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom</code> attribute), 275
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid3DTopBottom</code> attribute), 285	<code>__module__</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack</code> attribute), 276
<code>__module__</code> (<code>fipy.meshes.PeriodicGrid3DTopBottomFrontBack</code> attribute), 286	<code>__module__</code> (<code>fipy.meshes.skewedGrid2D.SkewedGrid2D</code> attribute), 276
<code>__module__</code> (<code>fipy.meshes.SkewedGrid2D</code> attribute), 286	<code>__module__</code> (<code>fipy.meshes.skewedGrid2D.SkewedGrid2DLeftRight</code> attribute), 276
<code>__module__</code> (<code>fipy.meshes.Tri2D</code> attribute), 287	<code>__module__</code> (<code>fipy.meshes.tri2D.Tri2D</code> attribute), 277
<code>__module__</code> (<code>fipy.meshes.abstractMesh.AbstractMesh</code> attribute), 248	<code>__module__</code> (<code>fipy.meshes.uniformGrid1D.UniformGrid1D</code> attribute), 278
<code>__module__</code> (<code>fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D</code> attribute), 254	<code>__module__</code> (<code>fipy.meshes.uniformGrid2D.UniformGrid2D</code> attribute), 278
<code>__module__</code> (<code>fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D</code> attribute), 255	<code>__module__</code> (<code>fipy.meshes.uniformGrid3D.UniformGrid3D</code> attribute), 278
<code>__module__</code> (<code>fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D</code> attribute), 256	
<code>__module__</code> (<code>fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D</code> attribute), 257	
<code>__module__</code> (<code>fipy.meshes.gmshMesh.Gmsh2D</code> attribute), 291	

- [attribute\), 279](#)
- [__module__ \(fipy.solvers.DummySolver attribute\), 323](#)
- [__module__ \(fipy.solvers.IllConditionedPreconditionerWarning attribute\), 322](#)
- [__module__ \(fipy.solvers.LinearBicgSolver attribute\), 324](#)
- [__module__ \(fipy.solvers.LinearCGSSolver attribute\), 324](#)
- [__module__ \(fipy.solvers.LinearGMRESSolver attribute\), 324](#)
- [__module__ \(fipy.solvers.LinearLUSolver attribute\), 323](#)
- [__module__ \(fipy.solvers.LinearPCGSolver attribute\), 324](#)
- [__module__ \(fipy.solvers.MatrixIllConditionedWarning attribute\), 322](#)
- [__module__ \(fipy.solvers.MaximumIterationWarning attribute\), 321](#)
- [__module__ \(fipy.solvers.PreconditionerNotPositiveDefiniteWarning attribute\), 322](#)
- [__module__ \(fipy.solvers.PreconditionerWarning attribute\), 321](#)
- [__module__ \(fipy.solvers.ScalarQuantityOutOfRangeWarning attribute\), 322](#)
- [__module__ \(fipy.solvers.Solver attribute\), 323](#)
- [__module__ \(fipy.solvers.SolverConvergenceWarning attribute\), 321](#)
- [__module__ \(fipy.solvers.StagnatedSolverWarning attribute\), 322](#)
- [__module__ \(fipy.solvers.petsc.DummySolver attribute\), 298](#)
- [__module__ \(fipy.solvers.petsc.LinearBicgSolver attribute\), 299](#)
- [__module__ \(fipy.solvers.petsc.LinearCGSSolver attribute\), 299](#)
- [__module__ \(fipy.solvers.petsc.LinearGMRESSolver attribute\), 299](#)
- [__module__ \(fipy.solvers.petsc.LinearLUSolver attribute\), 298](#)
- [__module__ \(fipy.solvers.petsc.LinearPCGSolver attribute\), 298](#)
- [__module__ \(fipy.solvers.petsc.comms.parallelPETScCommWrapper attribute\), 293](#)
- [__module__ \(fipy.solvers.petsc.comms.petscCommWrapper attribute\), 294](#)
- [__module__ \(fipy.solvers.petsc.comms.serialPETScCommWrapper attribute\), 294](#)
- [__module__ \(fipy.solvers.petsc.dummySolver.DummySolver attribute\), 295](#)
- [__module__ \(fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver attribute\), 295](#)
- [__module__ \(fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver attribute\), 295](#)
- [__module__ \(fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver attribute\), 295](#)
- [__module__ \(fipy.solvers.petsc.linearLUSolver.LinearLUSolver attribute\), 296](#)
- [__module__ \(fipy.solvers.petsc.linearPCGSolver.LinearPCGSolver attribute\), 296](#)
- [__module__ \(fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver attribute\), 297](#)
- [__module__ \(fipy.solvers.petsc.petscSolver.PETScSolver attribute\), 297](#)
- [__module__ \(fipy.solvers.pyAMG.LinearCGSSolver attribute\), 304](#)
- [__module__ \(fipy.solvers.pyAMG.LinearGMRESSolver attribute\), 303](#)
- [__module__ \(fipy.solvers.pyAMG.LinearGeneralSolver attribute\), 304](#)
- [__module__ \(fipy.solvers.pyAMG.LinearLUSolver attribute\), 304](#)
- [__module__ \(fipy.solvers.pyAMG.LinearPCGSolver attribute\), 304](#)
- [__module__ \(fipy.solvers.pyAMG.linearCGSSolver.LinearCGSSolver attribute\), 300](#)
- [__module__ \(fipy.solvers.pyAMG.linearGMRESSolver.LinearGMRESSolver attribute\), 301](#)
- [__module__ \(fipy.solvers.pyAMG.linearGeneralSolver.LinearGeneralSolver attribute\), 301](#)
- [__module__ \(fipy.solvers.pyAMG.linearLUSolver.LinearLUSolver attribute\), 302](#)
- [__module__ \(fipy.solvers.pyAMG.linearPCGSolver.LinearPCGSolver attribute\), 302](#)
- [__module__ \(fipy.solvers.pyAMG.preconditioners.smoothedAggregationAMGSolver attribute\), 300](#)
- [__module__ \(fipy.solvers.pyamgx.AggregationAMGSolver attribute\), 312](#)
- [__module__ \(fipy.solvers.pyamgx.LinearBiCGStabSolver attribute\), 312](#)
- [__module__ \(fipy.solvers.pyamgx.LinearCGSolver attribute\), 311](#)
- [__module__ \(fipy.solvers.pyamgx.LinearFGMRESSolver attribute\), 311](#)
- [__module__ \(fipy.solvers.pyamgx.LinearLUSolver attribute\), 312](#)
- [__module__ \(fipy.solvers.pyamgx.parallelPETScCommWrapper attribute\), 306](#)
- [__module__ \(fipy.solvers.pyamgx.petscCommWrapper attribute\), 306](#)
- [__module__ \(fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver attribute\), 306](#)
- [__module__ \(fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver attribute\), 307](#)
- [__module__ \(fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver attribute\), 308](#)
- [__module__ \(fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver attribute\), 308](#)
- [__module__ \(fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver attribute\), 309](#)
- [__module__ \(fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver attribute\), 309](#)

__module__ (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver attribute), 309	__module__ (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver attribute), 309
__module__ (fipy.solvers.scipy.LinearBicgstabSolver attribute), 310	__module__ (fipy.solvers.scipy.LinearBicgstabSolver attribute), 310
__module__ (fipy.solvers.scipy.LinearCGSSolver attribute), 316	__module__ (fipy.solvers.scipy.LinearCGSSolver attribute), 316
__module__ (fipy.solvers.scipy.LinearGMRESSolver attribute), 316	__module__ (fipy.solvers.scipy.LinearGMRESSolver attribute), 316
__module__ (fipy.solvers.scipy.LinearLUSolver attribute), 317	__module__ (fipy.solvers.scipy.LinearLUSolver attribute), 317
__module__ (fipy.solvers.scipy.LinearPCGSolver attribute), 317	__module__ (fipy.solvers.scipy.LinearPCGSolver attribute), 317
__module__ (fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver attribute), 314	__module__ (fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver attribute), 314
__module__ (fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver attribute), 314	__module__ (fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver attribute), 314
__module__ (fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver attribute), 315	__module__ (fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver attribute), 315
__module__ (fipy.solvers.scipy.linearLUSolver.LinearLUSolver attribute), 315	__module__ (fipy.solvers.scipy.linearLUSolver.LinearLUSolver attribute), 315
__module__ (fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver attribute), 315	__module__ (fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver attribute), 315
__module__ (fipy.solvers.solver.IllConditionedPreconditionerWarning attribute), 320	__module__ (fipy.solvers.solver.IllConditionedPreconditionerWarning attribute), 320
__module__ (fipy.solvers.solver.MatrixIllConditionedWarning attribute), 320	__module__ (fipy.solvers.solver.MatrixIllConditionedWarning attribute), 320
__module__ (fipy.solvers.solver.MaximumIterationWarning attribute), 319	__module__ (fipy.solvers.solver.MaximumIterationWarning attribute), 319
__module__ (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning attribute), 320	__module__ (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning attribute), 320
__module__ (fipy.solvers.solver.PreconditionerWarning attribute), 320	__module__ (fipy.solvers.solver.PreconditionerWarning attribute), 320
__module__ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning attribute), 320	__module__ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning attribute), 320
__module__ (fipy.solvers.solver.Solver attribute), 321	__module__ (fipy.solvers.solver.Solver attribute), 321
__module__ (fipy.solvers.solver.SolverConvergenceWarning attribute), 319	__module__ (fipy.solvers.solver.SolverConvergenceWarning attribute), 319
__module__ (fipy.solvers.solver.StagnatedSolverWarning attribute), 320	__module__ (fipy.solvers.solver.StagnatedSolverWarning attribute), 320
__module__ (fipy.steppers.pidStepper.PIDStepper attribute), 327	__module__ (fipy.steppers.pidStepper.PIDStepper attribute), 327
__module__ (fipy.steppers.pseudoRKQSStepper.PseudoRKQSStepper attribute), 328	__module__ (fipy.steppers.pseudoRKQSStepper.PseudoRKQSStepper attribute), 328
__module__ (fipy.steppers.stepper.Stepper attribute), 328	__module__ (fipy.steppers.stepper.Stepper attribute), 328
__module__ (fipy.terms.AbstractBaseClassError attribute), 354	__module__ (fipy.terms.AbstractBaseClassError attribute), 354
__module__ (fipy.terms.AdvectionTerm attribute), 359, 375	__module__ (fipy.terms.AdvectionTerm attribute), 359, 375
__module__ (fipy.terms.CentralDifferenceConvectionTerm attribute), 363	__module__ (fipy.terms.CentralDifferenceConvectionTerm attribute), 363
__module__ (fipy.terms.DiffusionTerm attribute), 360	__module__ (fipy.terms.DiffusionTerm attribute), 360
__module__ (fipy.terms.DiffusionTermCorrection attribute), 361	__module__ (fipy.terms.DiffusionTermCorrection attribute), 361
__module__ (fipy.terms.ExplicitDiffusionTerm attribute), 361	__module__ (fipy.terms.ExplicitDiffusionTerm attribute), 361
__module__ (fipy.terms.ExplicitUpwindConvectionTerm attribute), 365	__module__ (fipy.terms.ExplicitUpwindConvectionTerm attribute), 365
__module__ (fipy.terms.ExplicitVariableError attribute), 354	__module__ (fipy.terms.ExplicitVariableError attribute), 354
__module__ (fipy.terms.ExponentialConvectionTerm attribute), 366	__module__ (fipy.terms.ExponentialConvectionTerm attribute), 366
__module__ (fipy.terms.FirstOrderAdvectionTerm attribute), 356, 372	__module__ (fipy.terms.FirstOrderAdvectionTerm attribute), 356, 372
__module__ (fipy.terms.HybridConvectionTerm attribute), 367	__module__ (fipy.terms.HybridConvectionTerm attribute), 367
__module__ (fipy.terms.ImplicitSourceTerm attribute), 362	__module__ (fipy.terms.ImplicitSourceTerm attribute), 362
__module__ (fipy.terms.IncorrectSolutionVariable attribute), 355	__module__ (fipy.terms.IncorrectSolutionVariable attribute), 355
__module__ (fipy.terms.PowerLawConvectionTerm attribute), 368	__module__ (fipy.terms.PowerLawConvectionTerm attribute), 368
__module__ (fipy.terms.ResidualTerm attribute), 362	__module__ (fipy.terms.ResidualTerm attribute), 362
__module__ (fipy.terms.SolutionVariableNumberError attribute), 355	__module__ (fipy.terms.SolutionVariableNumberError attribute), 355
__module__ (fipy.terms.SolutionVariableRequiredError attribute), 355	__module__ (fipy.terms.SolutionVariableRequiredError attribute), 355
__module__ (fipy.terms.TermMultiplyError attribute), 354	__module__ (fipy.terms.TermMultiplyError attribute), 354
__module__ (fipy.terms.TransientTerm attribute), 360	__module__ (fipy.terms.TransientTerm attribute), 360
__module__ (fipy.terms.UpwindConvectionTerm attribute), 370	__module__ (fipy.terms.UpwindConvectionTerm attribute), 370
__module__ (fipy.terms.VanLeerConvectionTerm attribute), 371	__module__ (fipy.terms.VanLeerConvectionTerm attribute), 371
__module__ (fipy.terms.VectorCoeffError attribute), 354	__module__ (fipy.terms.VectorCoeffError attribute), 354
__module__ (fipy.terms.advectionTerm.AdvectionTerm attribute), 334	__module__ (fipy.terms.advectionTerm.AdvectionTerm attribute), 334
__module__ (fipy.terms.cellTerm.CellTerm attribute), 334	__module__ (fipy.terms.cellTerm.CellTerm attribute), 334
__module__ (fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm attribute), 336	__module__ (fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm attribute), 336
__module__ (fipy.terms.diffusionTerm.DiffusionTerm attribute), 336	__module__ (fipy.terms.diffusionTerm.DiffusionTerm attribute), 336
__module__ (fipy.terms.diffusionTerm.DiffusionTermCorrection attribute), 336	__module__ (fipy.terms.diffusionTerm.DiffusionTermCorrection attribute), 336
__module__ (fipy.terms.diffusionTerm.DiffusionTermNoCorrection attribute), 336	__module__ (fipy.terms.diffusionTerm.DiffusionTermNoCorrection attribute), 336
__module__ (fipy.terms.diffusionTermCorrection.DiffusionTermCorrection attribute), 337	__module__ (fipy.terms.diffusionTermCorrection.DiffusionTermCorrection attribute), 337
__module__ (fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection attribute), 337	__module__ (fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection attribute), 337
__module__ (fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm attribute), 337	__module__ (fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm attribute), 337
__module__ (fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm attribute), 365	__module__ (fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm attribute), 365

[attribute\), 339](#)
[__module__ \(fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm attribute\), 340](#)
[__module__ \(fipy.terms.faceTerm.FaceTerm attribute\), 341](#)
[__module__ \(fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm attribute\), 342](#)
[__module__ \(fipy.terms.hybridConvectionTerm.HybridConvectionTerm attribute\), 343](#)
[__module__ \(fipy.terms.implicitSourceTerm.ImplicitSourceTerm attribute\), 344](#)
[__module__ \(fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm attribute\), 345](#)
[__module__ \(fipy.terms.residualTerm.ResidualTerm attribute\), 346](#)
[__module__ \(fipy.terms.sourceTerm.SourceTerm attribute\), 346](#)
[__module__ \(fipy.terms.term.Term attribute\), 347](#)
[__module__ \(fipy.terms.transientTerm.TransientTerm attribute\), 351](#)
[__module__ \(fipy.terms.upwindConvectionTerm.UpwindConvectionTerm attribute\), 352](#)
[__module__ \(fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm attribute\), 354](#)
[__module__ \(fipy.tests.test.test attribute\), 378](#)
[__module__ \(fipy.tools.PhysicalField attribute\), 422](#)
[__module__ \(fipy.tools.Vitals attribute\), 433](#)
[__module__ \(fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper attribute\), 381](#)
[__module__ \(fipy.tools.comms.dummyComm.DummyComm attribute\), 382](#)
[__module__ \(fipy.tools.dimensions.physicalField.PhysicalField attribute\), 389](#)
[__module__ \(fipy.tools.dimensions.physicalField.PhysicalUnit attribute\), 402](#)
[__module__ \(fipy.tools.vitals.Vitals attribute\), 417](#)
[__module__ \(fipy.variables.BetaNoiseVariable attribute\), 501](#)
[__module__ \(fipy.variables.CellVariable attribute\), 490](#)
[__module__ \(fipy.variables.DistanceVariable attribute\), 516](#)
[__module__ \(fipy.variables.ExponentialNoiseVariable attribute\), 503](#)
[__module__ \(fipy.variables.FaceVariable attribute\), 496](#)
[__module__ \(fipy.variables.GammaNoiseVariable attribute\), 505](#)
[__module__ \(fipy.variables.GaussianNoiseVariable attribute\), 507](#)
[__module__ \(fipy.variables.HistogramVariable attribute\), 509](#)
[__module__ \(fipy.variables.ModularVariable attribute\), 499](#)
[__module__ \(fipy.variables.ScharfetterGummelFaceVariable attribute\), 513](#)
[__module__ \(fipy.variables.SurfactantConvectionVariable attribute\), 511](#)
[__module__ \(fipy.variables.SurfactantVariable attribute\), 511](#)
[__module__ \(fipy.variables.UniformNoiseVariable attribute\), 508](#)
[__module__ \(fipy.variables.Variable attribute\), 483](#)
[__module__ \(fipy.variables.betaNoiseVariable.BetaNoiseVariable attribute\), 437](#)
[__module__ \(fipy.variables.cellVariable.CellVariable attribute\), 439](#)
[__module__ \(fipy.variables.distanceVariable.DistanceVariable attribute\), 448](#)
[__module__ \(fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable attribute\), 451](#)
[__module__ \(fipy.variables.faceVariable.FaceVariable attribute\), 451](#)
[__module__ \(fipy.variables.gammaNoiseVariable.GammaNoiseVariable attribute\), 454](#)
[__module__ \(fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable attribute\), 457](#)
[__module__ \(fipy.variables.histogramVariable.HistogramVariable attribute\), 457](#)
[__module__ \(fipy.variables.modularVariable.ModularVariable attribute\), 459](#)
[__module__ \(fipy.variables.noiseVariable.NoiseVariable attribute\), 460](#)
[__module__ \(fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable attribute\), 461](#)
[__module__ \(fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable attribute\), 463](#)
[__module__ \(fipy.variables.surfactantVariable.SurfactantVariable attribute\), 465](#)
[__module__ \(fipy.variables.uniformNoiseVariable.UniformNoiseVariable attribute\), 467](#)
[__module__ \(fipy.variables.variable.Variable attribute\), 472](#)
[__module__ \(fipy.viewers.DummyViewer attribute\), 571](#)
[__module__ \(fipy.viewers.Matplotlib1DViewer attribute\), 558](#)
[__module__ \(fipy.viewers.Matplotlib2DGridContourViewer attribute\), 561](#)
[__module__ \(fipy.viewers.Matplotlib2DGridViewer attribute\), 559](#)
[__module__ \(fipy.viewers.Matplotlib2DViewer attribute\), 562](#)
[__module__ \(fipy.viewers.MatplotlibStreamViewer attribute\), 566](#)
[__module__ \(fipy.viewers.MatplotlibVectorViewer attribute\), 564](#)
[__module__ \(fipy.viewers.MayaviClient attribute\), 568](#)

<code>__module__</code> (<i>fipy.viewers.MeshDimensionError</i> attribute), 571	<code>__module__</code> (<i>fipy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer</i> attribute), 550
<code>__module__</code> (<i>fipy.viewers.MultiViewer</i> attribute), 568	<code>__module__</code> (<i>fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer</i> attribute), 551
<code>__module__</code> (<i>fipy.viewers.TSVViewer</i> attribute), 570	<code>__module__</code> (<i>fipy.viewers.vtkViewer.vtkViewer.VTKViewer</i> attribute), 551
<code>__module__</code> (<i>fipy.viewers.VTKCellViewer</i> attribute), 571	<code>__module__</code> (<i>package.subpackage.base.Base</i> attribute), 234
<code>__module__</code> (<i>fipy.viewers.VTKFaceViewer</i> attribute), 571	<code>__module__</code> (<i>package.subpackage.object.Object</i> attribute), 235
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.Matplotlib1DViewer</i> attribute), 536	<code>__module__</code> (<i>fipy.meshes.abstractMesh.AbstractMesh</i> method), 248
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.Matplotlib2DGridContourViewer</i> attribute), 539	<code>__module__</code> (<i>fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D</i> method), 254
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.Matplotlib2DGridViewer</i> attribute), 537	<code>__module__</code> (<i>fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D</i> method), 255
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.Matplotlib2DViewer</i> attribute), 540	<code>__module__</code> (<i>fipy.meshes.mesh.Mesh</i> method), 265
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.MatplotlibStreamViewer</i> attribute), 545	<code>__mul__</code> () (<i>fipy.meshes.mesh1D.Mesh1D</i> method), 268
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.MatplotlibVectorViewer</i> attribute), 542	<code>__mul__</code> () (<i>fipy.meshes.mesh2D.Mesh2D</i> method), 268
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer</i> attribute), 520	<code>__mul__</code> () (<i>fipy.meshes.uniformGrid1D.UniformGrid1D</i> method), 278
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer</i> attribute), 522	<code>__mul__</code> () (<i>fipy.meshes.uniformGrid2D.UniformGrid2D</i> method), 278
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer</i> attribute), 523	<code>__mul__</code> () (<i>fipy.meshes.uniformGrid3D.UniformGrid3D</i> method), 278
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer</i> attribute), 525	<code>__mul__</code> () (<i>fipy.terms.term.Term</i> method), 347
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer</i> attribute), 526	<code>__mul__</code> () (<i>fipy.tools.physicalField.PhysicalField</i> method), 422
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.MatplotlibSparseMatrixViewer</i> attribute), 527	<code>__mul__</code> () (<i>fipy.tools.dimensions.physicalField.PhysicalField</i> method), 439
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer</i> attribute), 530	<code>__mul__</code> () (<i>fipy.tools.dimensions.physicalField.PhysicalUnit</i> method), 439
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer</i> attribute), 532	<code>__mul__</code> () (<i>fipy.variables.Variable</i> method), 483
<code>__module__</code> (<i>fipy.viewers.matplotlibViewer.matplotlibViewerAbstract.MatplotlibViewerAbstract</i> attribute), 533	<code>__mul__</code> () (<i>fipy.variables.variable.Variable</i> method), 472
<code>__module__</code> (<i>fipy.viewers.mayaviViewer.MayaviClient</i> attribute), 550	<code>__ne__</code> () (<i>fipy.tools.dimensions.physicalField.PhysicalField</i> method), 389
<code>__module__</code> (<i>fipy.viewers.mayaviViewer.mayaviClient.MayaviClient</i> attribute), 547	<code>__ne__</code> () (<i>fipy.tools.dimensions.physicalField.PhysicalUnit</i> method), 403
<code>__module__</code> (<i>fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon</i> attribute), 547	<code>__ne__</code> () (<i>fipy.variables.Variable</i> method), 483
<code>__module__</code> (<i>fipy.viewers.multiViewer.MultiViewer</i> attribute), 553	<code>__ne__</code> () (<i>fipy.variables.variable.Variable</i> method), 472
<code>__module__</code> (<i>fipy.viewers.tsvViewer.TSVViewer</i> attribute), 554	<code>__neg__</code> () (<i>fipy.terms.term.Term</i> method), 347
<code>__module__</code> (<i>fipy.viewers.viewer.AbstractViewer</i> attribute), 555	<code>__neg__</code> () (<i>fipy.tools.PhysicalField</i> method), 422
<code>__module__</code> (<i>fipy.viewers.vtkViewer.VTKCellViewer</i> attribute), 552	<code>__neg__</code> () (<i>fipy.tools.dimensions.physicalField.PhysicalField</i> method), 390
<code>__module__</code> (<i>fipy.viewers.vtkViewer.VTKFaceViewer</i> attribute), 552	<code>__neg__</code> () (<i>fipy.variables.Variable</i> method), 483
	<code>__neg__</code> () (<i>fipy.variables.variable.Variable</i> method), 472
	<code>__new__</code> () (<i>fipy.variables.Variable</i> static method), 483
	<code>__new__</code> () (<i>fipy.variables.variable.Variable</i> static method), 472

`__nonzero__()` (*fipy.tools.PhysicalField method*), 422
`__nonzero__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__nonzero__()` (*fipy.variables.Variable method*), 483
`__nonzero__()` (*fipy.variables.variable.Variable method*), 472
`__or__()` (*fipy.variables.Variable method*), 483
`__or__()` (*fipy.variables.variable.Variable method*), 473
`__pos__()` (*fipy.terms.term.Term method*), 347
`__pos__()` (*fipy.tools.PhysicalField method*), 422
`__pos__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__pos__()` (*fipy.variables.Variable method*), 483
`__pos__()` (*fipy.variables.variable.Variable method*), 473
`__pow__()` (*fipy.tools.PhysicalField method*), 422
`__pow__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__pow__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 403
`__pow__()` (*fipy.variables.Variable method*), 483
`__pow__()` (*fipy.variables.variable.Variable method*), 473
`__prefix_traits__` (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon attribute*), 547
`__radd__()` (*fipy.meshes.abstractMesh.AbstractMesh method*), 248
`__radd__()` (*fipy.terms.term.Term method*), 347
`__radd__()` (*fipy.tools.PhysicalField method*), 422
`__radd__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__radd__()` (*fipy.variables.Variable method*), 484
`__radd__()` (*fipy.variables.variable.Variable method*), 473
`__rand__()` (*fipy.terms.term.Term method*), 347
`__rdiv__()` (*fipy.tools.PhysicalField method*), 422
`__rdiv__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__rdiv__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 403
`__rdiv__()` (*fipy.variables.Variable method*), 484
`__rdiv__()` (*fipy.variables.variable.Variable method*), 473
`__repr__()` (*fipy.boundaryConditions.Constraint method*), 240
`__repr__()` (*fipy.boundaryConditions.boundaryCondition.BoundaryCondition method*), 237
`__repr__()` (*fipy.boundaryConditions.constraint.Constraint method*), 238
`__repr__()` (*fipy.meshes.abstractMesh.AbstractMesh method*), 249
`__repr__()` (*fipy.solvers.Solver method*), 323
`__repr__()` (*fipy.solvers.solver.Solver method*), 321
`__repr__()` (*fipy.terms.ResidualTerm method*), 362
`__repr__()` (*fipy.terms.residualTerm.ResidualTerm method*), 346
`__repr__()` (*fipy.terms.term.Term method*), 347
`__repr__()` (*fipy.tools.PhysicalField method*), 423
`__repr__()` (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper method*), 381
`__repr__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__repr__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 404
`__repr__()` (*fipy.variables.Variable method*), 484
`__repr__()` (*fipy.variables.variable.Variable method*), 473
`__rmul__()` (*fipy.meshes.abstractMesh.AbstractMesh method*), 249
`__rmul__()` (*fipy.meshes.mesh.Mesh method*), 266
`__rmul__()` (*fipy.terms.term.Term method*), 347
`__rmul__()` (*fipy.tools.PhysicalField method*), 423
`__rmul__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__rmul__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 404
`__rmul__()` (*fipy.variables.Variable method*), 484
`__rmul__()` (*fipy.variables.variable.Variable method*), 473
`__rpow__()` (*fipy.tools.PhysicalField method*), 423
`__rpow__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__rpow__()` (*fipy.variables.Variable method*), 484
`__rpow__()` (*fipy.variables.variable.Variable method*), 473
`__rsub__()` (*fipy.terms.term.Term method*), 347
`__rsub__()` (*fipy.tools.PhysicalField method*), 423
`__rsub__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__rsub__()` (*fipy.variables.ModularVariable method*), 499
`__rsub__()` (*fipy.variables.Variable method*), 484
`__rsub__()` (*fipy.variables.modularVariable.ModularVariable method*), 459
`__rsub__()` (*fipy.variables.variable.Variable method*), 473
`__rtruediv__()` (*fipy.tools.PhysicalField method*), 423
`__rtruediv__()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 390
`__rtruediv__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 404
`__rtruediv__()` (*fipy.variables.Variable method*), 484
`__rtruediv__()` (*fipy.variables.variable.Variable method*), 473

- `method`), 473
- `__setitem__` () (*fipy.tools.PhysicalField* method), 423
- `__setitem__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 391
- `method`), 391
- `__setitem__` () (*fipy.variables.Variable* method), 484
- `__setitem__` () (*fipy.variables.variable.Variable* method), 473
- `__setstate__` () (*fipy.meshes.Gmsh2D* method), 291
- `__setstate__` () (*fipy.meshes.Gmsh3D* method), 292
- `__setstate__` () (*fipy.meshes.abstractMesh.AbstractMesh* method), 249
- `__setstate__` () (*fipy.meshes.gmshMesh.Gmsh2D* method), 263
- `__setstate__` () (*fipy.meshes.gmshMesh.Gmsh3D* method), 264
- `__setstate__` () (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 381
- `__setstate__` () (*fipy.variables.CellVariable* method), 490
- `__setstate__` () (*fipy.variables.Variable* method), 484
- `__setstate__` () (*fipy.variables.cellVariable.CellVariable* method), 439
- `__setstate__` () (*fipy.variables.variable.Variable* method), 473
- `__str__` () (*fipy.solvers.IllConditionedPreconditionerWarning* method), 322
- `__str__` () (*fipy.solvers.MatrixIllConditionedWarning* method), 322
- `__str__` () (*fipy.solvers.MaximumIterationWarning* method), 321
- `__str__` () (*fipy.solvers.PreconditionerNotPositiveDefiniteWarning* method), 322
- `__str__` () (*fipy.solvers.ScalarQuantityOutOfRangeWarning* method), 322
- `__str__` () (*fipy.solvers.SolverConvergenceWarning* method), 321
- `__str__` () (*fipy.solvers.StagnatedSolverWarning* method), 322
- `__str__` () (*fipy.solvers.solver.IllConditionedPreconditionerWarning* method), 320
- `__str__` () (*fipy.solvers.solver.MatrixIllConditionedWarning* method), 320
- `__str__` () (*fipy.solvers.solver.MaximumIterationWarning* method), 319
- `__str__` () (*fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning* method), 320
- `__str__` () (*fipy.solvers.solver.ScalarQuantityOutOfRangeWarning* method), 320
- `__str__` () (*fipy.solvers.solver.SolverConvergenceWarning* method), 319
- `__str__` () (*fipy.solvers.solver.StagnatedSolverWarning* method), 320
- `__str__` () (*fipy.tools.PhysicalField* method), 423
- `__str__` () (*fipy.tools.Vitals* method), 433
- `__str__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 391
- `__str__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 405
- `__str__` () (*fipy.tools.vitals.Vitals* method), 417
- `__str__` () (*fipy.variables.Variable* method), 484
- `__str__` () (*fipy.variables.variable.Variable* method), 473
- `__sub__` () (*fipy.meshes.abstractMesh.AbstractMesh* method), 249
- `__sub__` () (*fipy.terms.term.Term* method), 347
- `__sub__` () (*fipy.tools.PhysicalField* method), 423
- `__sub__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 391
- `__sub__` () (*fipy.variables.ModularVariable* method), 499
- `__sub__` () (*fipy.variables.Variable* method), 484
- `__sub__` () (*fipy.variables.modularVariable.ModularVariable* method), 459
- `__sub__` () (*fipy.variables.variable.Variable* method), 473
- `__truediv__` () (*fipy.meshes.abstractMesh.AbstractMesh* method), 250
- `__truediv__` () (*fipy.terms.term.Term* method), 347
- `__truediv__` () (*fipy.tools.PhysicalField* method), 423
- `__truediv__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 391
- `__truediv__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 405
- `__truediv__` () (*fipy.variables.Variable* method), 484
- `__truediv__` () (*fipy.variables.variable.Variable* method), 473
- `__view_traits__` (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* attribute), 547
- `__weakref__` (*fipy.boundaryConditions.Constraint* attribute), 240
- `__weakref__` (*fipy.boundaryConditions.boundaryCondition.BoundaryCondition* attribute), 237
- `__weakref__` (*fipy.boundaryConditions.constraint.Constraint* attribute), 238
- `__weakref__` (*fipy.meshes.abstractMesh.AbstractMesh* attribute), 250
- `__weakref__` (*fipy.meshes.mesh.MeshAdditionError* attribute), 265
- `__weakref__` (*fipy.solvers.Solver* attribute), 323
- `__weakref__` (*fipy.solvers.SolverConvergenceWarning* attribute), 321
- `__weakref__` (*fipy.solvers.pyAMG.preconditioners.smoothedAggregation* attribute), 300
- `__weakref__` (*fipy.solvers.solver.Solver* attribute), 321
- `__weakref__` (*fipy.solvers.solver.SolverConvergenceWarning* attribute), 321

- attribute*), 319
 - `__weakref__` (*fipy.steps.stepper.Stepper* attribute), 328
 - `__weakref__` (*fipy.terms.AbstractBaseClassError* attribute), 354
 - `__weakref__` (*fipy.terms.ExplicitVariableError* attribute), 354
 - `__weakref__` (*fipy.terms.IncorrectSolutionVariable* attribute), 355
 - `__weakref__` (*fipy.terms.SolutionVariableNumberError* attribute), 355
 - `__weakref__` (*fipy.terms.SolutionVariableRequiredError* attribute), 355
 - `__weakref__` (*fipy.terms.TermMultiplyError* attribute), 354
 - `__weakref__` (*fipy.terms.VectorCoeffError* attribute), 354
 - `__weakref__` (*fipy.terms.term.Term* attribute), 347
 - `__weakref__` (*fipy.tools.PhysicalField* attribute), 424
 - `__weakref__` (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* attribute), 381
 - `__weakref__` (*fipy.tools.dimensions.physicalField.PhysicalField* attribute), 391
 - `__weakref__` (*fipy.tools.dimensions.physicalField.PhysicalUnit* attribute), 405
 - `__weakref__` (*fipy.variables.Variable* attribute), 484
 - `__weakref__` (*fipy.variables.variable.Variable* attribute), 473
 - `__weakref__` (*fipy.viewers.MeshDimensionError* attribute), 571
 - `__weakref__` (*fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer* attribute), 527
 - `__weakref__` (*fipy.viewers.viewer.AbstractViewer* attribute), 555
 - `__weakref__` (*package.subpackage.base.Base* attribute), 234
 - `--cache`
 - command line option, 31
 - `--inline`
 - command line option, 31
 - `--lsmlib`
 - command line option, 31
 - `--no-cache`
 - command line option, 31
 - `--no-pysparse`
 - command line option, 31
 - `--pyamg`
 - command line option, 31
 - `--pyamgx`
 - command line option, 31
 - `--pysparse`
 - command line option, 31
 - `--scipy`
 - command line option, 31
 - `--skfmm`
 - command line option, 31
 - `--trilinos`
 - command line option, 31
- ## A
- AbstractBaseClassError*, 354
 - AbstractCommWrapper* (class in *fipy.tools.comms.abstractCommWrapper*), 381
 - AbstractMatplotlibViewer* (class in *fipy.viewers.matplotlibViewer.matplotlibViewer*), 532
 - AbstractMesh* (class in *fipy.meshes.abstractMesh*), 246
 - AbstractViewer* (class in *fipy.viewers.viewer*), 555
 - add()* (*fipy.tools.dimensions.physicalField.PhysicalField* method), 391
 - add()* (*fipy.tools.PhysicalField* method), 424
 - AbstractCommWrapper* (class in *fipy.terms*), 356, 372
 - AdvectionTerm* (class in *fipy.terms.advectionTerm*), 331
 - AggregationAMGSolver* (class in *fipy.solvers.pyamgx*), 312
 - AggregationAMGSolver* (class in *fipy.solvers.pyamgx.aggregationAMGSolver*), 305
 - all()* (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 293
 - all()* (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 382
 - all()* (*fipy.variables.Variable* method), 484
 - all()* (*fipy.variables.variable.Variable* method), 474
 - all()* (in module *fipy.tools.numerix*), 410–412
 - allclose*, 173
 - allclose()* (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 293
 - allclose()* (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 382
 - allclose()* (*fipy.tools.dimensions.physicalField.PhysicalField* method), 392
 - allclose()* (*fipy.tools.PhysicalField* method), 424
 - allclose()* (*fipy.variables.Variable* method), 484
 - allclose()* (*fipy.variables.variable.Variable* method), 474
 - allclose()* (in module *fipy.tools.numerix*), 410, 412
 - allequal()* (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 293
 - allequal()* (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 382
 - allequal()* (*fipy.tools.dimensions.physicalField.PhysicalField* method), 392
 - allequal()* (*fipy.tools.PhysicalField* method), 424
 - allequal()* (*fipy.variables.Variable* method), 484

- `allequal()` (*fipy.variables.variable.Variable* method), 474
`allequal()` (in module *fipy.tools.numerix*), 412
`allgather()` (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 293
`allgather()` (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 382
`any()` (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 293
`any()` (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 382
`any()` (*fipy.variables.Variable* method), 484
`any()` (*fipy.variables.variable.Variable* method), 474
`appendChild()` (*fipy.tools.Vitals* method), 433
`appendChild()` (*fipy.tools.vitals.Vitals* method), 417
`appendInfo()` (*fipy.tools.Vitals* method), 433
`appendInfo()` (*fipy.tools.vitals.Vitals* method), 417
AppVeyor, 99
`arccos()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 392
`arccos()` (*fipy.tools.PhysicalField* method), 424
`arccosh()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 392
`arccosh()` (*fipy.tools.PhysicalField* method), 424
`arcsin()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 392
`arcsin()` (*fipy.tools.PhysicalField* method), 425
`arctan`, 175
`arctan()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 392
`arctan()` (*fipy.tools.PhysicalField* method), 425
`arctan2`, 175
`arctan2()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 393
`arctan2()` (*fipy.tools.PhysicalField* method), 425
`arctanh()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 393
`arctanh()` (*fipy.tools.PhysicalField* method), 425
`arithmeticFaceValue()` (*fipy.variables.CellVariable* property), 490
`arithmeticFaceValue()` (*fipy.variables.cellVariable.CellVariable* property), 439
`arithmeticFaceValue()` (*fipy.variables.ModularVariable* property), 499
`arithmeticFaceValue()` (*fipy.variables.modularVariable.ModularVariable* property), 459
`array`, 163
`aspect2D()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 250
- ## B
- `Barrier()` (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 293
`Barrier()` (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 381
`Base` (class in package *fipy.tools.comms.abstractCommWrapper*), 233
`bcast()` (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 294
`bcast()` (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 382
`BetaNoiseVariable` (class in *fipy.variables*), 500
`BetaNoiseVariable` (class in *fipy.variables.betaNoiseVariable*), 435
`BoundaryCondition` (class in *fipy.boundaryConditions.boundaryCondition*), 237
Buildbot, 99
`cacheMatrix`, 212
`cacheMatrix()` (*fipy.terms.term.Term* method), 347
`cacheMe()` (*fipy.variables.Variable* method), 485
`cacheMe()` (*fipy.variables.variable.Variable* method), 474
`cacheRHSvector`, 212
`cacheRHSvector()` (*fipy.terms.term.Term* method), 348
`calcDistanceFunction()` (*fipy.variables.DistanceVariable* method), 516
`calcDistanceFunction()` (*fipy.variables.distanceVariable.DistanceVariable* method), 448
`ceil()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 393
`Cell()` (*fipy.tools.PhysicalField* method), 426
`cellCenters()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 250
`cellCenters()` (*fipy.meshes.PeriodicGrid1D* property), 282
`cellCenters()` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 271
`cellDistanceVectors()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 250
`cellFaceIDs()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 250
`cellInterfaceAreas()` (*fipy.variables.DistanceVariable* property), 516
`cellInterfaceAreas()` (*fipy.variables.distanceVariable.DistanceVariable* property), 448
`CellTerm` (class in *fipy.terms.cellTerm*), 334

- `cellToFaceDistanceVectors()`
(fipy.meshes.abstractMesh.AbstractMesh property), 250
- `CellVariable`, 139, 145, 150, 158, 167, 178, 181, 210, 227, 229
- `CellVariable` (class in *fipy.variables*), 489
- `CellVariable` (class in *fipy.variables.cellVariable*), 437
- `cellVolumeAverage()` (*fipy.variables.CellVariable property*), 491
- `cellVolumeAverage()`
(fipy.variables.cellVariable.CellVariable property), 439
- `cellVolumes()` (*fipy.meshes.abstractMesh.AbstractMesh property*), 250
- `cellVolumes()` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property*), 256
- `cellVolumes()` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property*), 257
- `CentralDifferenceConvectionTerm` (class in *fipy.terms*), 362
- `CentralDifferenceConvectionTerm` (class in *fipy.terms.centralDiffConvectionTerm*), 334
- `childNodes` (*fipy.tools.Vitals attribute*), 433
- `childNodes` (*fipy.tools.vitals.Vitals attribute*), 417
- `CircleCI`, 99
- `ClassicalAMGSolver` (class in *fipy.solvers.pyamgx.classicalAMGSolver*), 306
- `clip_data()` (*fipy.viewers.mayaviViewer.mayaviDaemonViewer.mayaviDaemonViewer method*), 547
- command line option
 - cache, 31
 - inline, 31
 - lsmlib, 31
 - no-cache, 31
 - no-pysparse, 31
 - pyamg, 31
 - pyamgx, 31
 - pysparse, 31
 - scipy, 31
 - skfmm, 31
 - trilinos, 31
- `conda`, 99
- `conjugate()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 393
- `conjugate()` (*fipy.tools.PhysicalField method*), 426
- `constrain()` (*fipy.variables.CellVariable method*), 491
- `constrain()` (*fipy.variables.cellVariable.CellVariable method*), 440
- `constrain()` (*fipy.variables.Variable method*), 485
- `constrain()` (*fipy.variables.variable.Variable method*), 474
- `Constraint` (class in *fipy.boundaryConditions*), 240
- `Constraint` (class in *fipy.boundaryConditions.constraint*), 238
- `constraints()` (*fipy.variables.Variable property*), 485
- `constraints()` (*fipy.variables.variable.Variable property*), 475
- Continuous Integration, 99
- `ConvectionTerm` (in module *fipy.terms*), 355
- `conversionFactorTo()`
(fipy.tools.dimensions.physicalField.PhysicalUnit method), 405
- `conversionTupleTo()`
(fipy.tools.dimensions.physicalField.PhysicalUnit method), 406
- `copy()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 393
- `copy()` (*fipy.terms.term.Term method*), 348
- `copy()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 394
- `copy()` (*fipy.tools.PhysicalField method*), 426
- `copy()` (*fipy.variables.CellVariable method*), 492
- `copy()` (*fipy.variables.cellVariable.CellVariable method*), 440
- `copy()` (*fipy.variables.FaceVariable method*), 496
- `copy()` (*fipy.variables.faceVariable.FaceVariable method*), 451
- `copy()` (*fipy.variables.noiseVariable.NoiseVariable method*), 460
- `copy()` (*fipy.variables.SurfactantVariable method*), 511
- `copy()` (*fipy.variables.surfactantVariable.SurfactantVariable method*), 465
- `copy()` (*fipy.variables.Variable method*), 485
- `copy()` (*fipy.variables.variable.Variable method*), 475
- `cos()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 394
- `cos()` (*fipy.tools.PhysicalField method*), 426
- `cosh()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 394
- `cosh()` (*fipy.tools.PhysicalField method*), 427
- `CylindricalGrid1D()` (in module *fipy.meshes*), 281
- `CylindricalGrid1D()` (in module *fipy.meshes.factoryMeshes*), 258
- `CylindricalGrid2D()` (in module *fipy.meshes*), 281
- `CylindricalGrid2D()` (in module *fipy.meshes.factoryMeshes*), 258
- `CylindricalNonUniformGrid1D` (class in *fipy.meshes.cylindricalNonUniformGrid1D*), 253
- `CylindricalNonUniformGrid2D` (class in *fipy.meshes.cylindricalNonUniformGrid2D*), 255

CylindricalUniformGrid1D (class *fipy.meshes.cylindricalUniformGrid1D*), 256

CylindricalUniformGrid2D (class *fipy.meshes.cylindricalUniformGrid2D*), 256

D

DefaultAsymmetricSolver, 145, 172

DefaultAsymmetricSolver (in module *fipy.solvers*), 323

DefaultAsymmetricSolver (in module *fipy.solvers.petsc*), 298

DefaultAsymmetricSolver (in module *fipy.solvers.pyAMG*), 303

DefaultAsymmetricSolver (in module *fipy.solvers.pyamgx*), 310

DefaultAsymmetricSolver (in module *fipy.solvers.scipy*), 316

DefaultSolver (in module *fipy.solvers*), 323

DefaultSolver (in module *fipy.solvers.petsc*), 298

DefaultSolver (in module *fipy.solvers.pyAMG*), 303

DefaultSolver (in module *fipy.solvers.pyamgx*), 310

DefaultSolver (in module *fipy.solvers.scipy*), 316

deprecate () (in module *fipy.tools.decorators*), 408

dictToXML () (*fipy.tools.Vitals* method), 433

dictToXML () (*fipy.tools.vitals.Vitals* method), 417

DiffusionTerm (class in *fipy.terms*), 360

DiffusionTerm (class in *fipy.terms.diffusionTerm*), 336

DiffusionTermCorrection (class in *fipy.terms*), 361

DiffusionTermCorrection (class in *fipy.terms.diffusionTerm*), 336

DiffusionTermCorrection (class in *fipy.terms.diffusionTermCorrection*), 337

DiffusionTermNoCorrection (class in *fipy.terms*), 361

DiffusionTermNoCorrection (class in *fipy.terms.diffusionTerm*), 336

DiffusionTermNoCorrection (class in *fipy.terms.diffusionTermNoCorrection*), 337

DistanceVariable (class in *fipy.variables*), 513

DistanceVariable (class in *fipy.variables.distanceVariable*), 445

divergence () (*fipy.variables.FaceVariable* property), 497

divergence () (*fipy.variables.faceVariable.FaceVariable* property), 451

divide () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 395

divide () (*fipy.tools.PhysicalField* method), 427

doctype (*fipy.tools.Vitals* attribute), 433

doctype (*fipy.tools.vitals.Vitals* attribute), 417

dontCacheMe () (*fipy.variables.Variable* method), 486

dontCacheMe () (*fipy.variables.variable.Variable* method), 475

dot () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 395

dot () (*fipy.tools.PhysicalField* method), 427

dot () (*fipy.variables.Variable* method), 486

dot () (*fipy.variables.variable.Variable* method), 475

dot () (in module *fipy.tools.numerix*), 409, 412

DummyComm (class in *fipy.tools.comms.dummyComm*), 382

DummySolver (class in *fipy.solvers*), 323

DummySolver (class in *fipy.solvers.petsc*), 298

DummySolver (class in *fipy.solvers.petsc.dummySolver*), 294

DummySolver (in module *fipy.solvers.pyAMG*), 303

DummySolver (in module *fipy.solvers.pyamgx*), 310

DummySolver (in module *fipy.solvers.scipy*), 316

DummyViewer (class in *fipy.viewers*), 571

E

environment variable

FIPY_CACHE, 32

FIPY_DISPLAY_MATRIX, 31

FIPY_INCLUDE_NUMERIX_ALL, 32, 222

FIPY_INLINE, 31

FIPY_INLINE_COMMENT, 32

FIPY_SOLVERS, 23, 31, 32

FIPY_VERBOSE_SOLVER, 32

FIPY_VIEWER, 32

examples.cahnHilliard.mesh2DCoupled module, 203

examples.cahnHilliard.sphere module, 206

examples.convection.exponential1D.mesh1D module, 143

examples.convection.exponential1DSource.mesh1D module, 145

examples.convection.robin module, 146

examples.convection.source module, 148

examples.diffusion.anisotropy module, 141

examples.diffusion.circle module, 130

examples.diffusion.coupled module, 125

examples.diffusion.electrostatics module, 134

examples.diffusion.mesh1D module, 105

examples.diffusion.mesh20x20

- module, 127
 - examples.diffusion.nthOrder.input4thOrder1D
 - module, 139
 - examples.flow.stokesCavity
 - module, 209
 - examples.levelSet.advection.circle
 - module, 201
 - examples.levelSet.advection.mesh1D
 - module, 199
 - examples.levelSet.distanceFunction.circle
 - module, 198
 - examples.levelSet.distanceFunction.mesh1D
 - module, 197
 - examples.phase.anisotropy
 - module, 174
 - examples.phase.binaryCoupled
 - module, 158
 - examples.phase.impingement.mesh20x20
 - module, 180
 - examples.phase.impingement.mesh40x1
 - module, 177
 - examples.phase.polyxtal
 - module, 184
 - examples.phase.polyxtalCoupled
 - module, 190
 - examples.phase.quaternary
 - module, 167
 - examples.phase.simple
 - module, 149
 - examples.reactiveWetting.liquidVapor1D
 - module, 215
 - examples.updating.update0_1to1_0
 - module, 226
 - examples.updating.update1_0to2_0
 - module, 222
 - examples.updating.update2_0to3_0
 - module, 221
 - execButNoTest () (in module *fipy.tests.doctestPlus*), 377
 - exp, 144, 146, 163, 179, 182
 - ExplicitDiffusionTerm, 107, 179, 181
 - ExplicitDiffusionTerm (class in *fipy.terms*), 361
 - ExplicitDiffusionTerm (class in *fipy.terms.explicitDiffusionTerm*), 337
 - ExplicitUpwindConvectionTerm (class in *fipy.terms*), 363
 - ExplicitUpwindConvectionTerm (class in *fipy.terms.explicitUpwindConvectionTerm*), 338
 - ExplicitVariableError, 354
 - ExponentialConvectionTerm, 227
 - ExponentialConvectionTerm (class in *fipy.terms*), 365
 - ExponentialConvectionTerm (class in *fipy.terms.exponentialConvectionTerm*), 339
 - ExponentialNoiseVariable (class in *fipy.variables*), 501
 - ExponentialNoiseVariable (class in *fipy.variables.exponentialNoiseVariable*), 449
 - extendVariable () (*fipy.variables.DistanceVariable* method), 517
 - extendVariable () (*fipy.variables.distanceVariable.DistanceVariable* method), 449
 - extendVariable () (*fipy.meshes.abstractMesh.AbstractMesh* property), 250
 - exteriorFaces () (*fipy.meshes.abstractMesh.AbstractMesh* property), 250
 - exteriorFaces () (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 278
 - extrude () (*fipy.meshes.mesh2D.Mesh2D* method), 268
- ## F
- faceCellIDs () (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 278
 - faceCellIDs () (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 279
 - faceCellIDs () (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 279
 - faceCenters () (*fipy.meshes.abstractMesh.AbstractMesh* property), 250
 - faceGrad () (*fipy.variables.CellVariable* property), 492
 - faceGrad () (*fipy.variables.cellVariable.CellVariable* property), 441
 - faceGrad () (*fipy.variables.ModularVariable* property), 499
 - faceGrad () (*fipy.variables.modularVariable.ModularVariable* property), 459
 - faceGradAverage () (*fipy.variables.CellVariable* property), 492
 - faceGradAverage () (*fipy.variables.cellVariable.CellVariable* property), 441
 - faceGradNoMod () (*fipy.variables.ModularVariable* property), 499
 - faceGradNoMod () (*fipy.variables.modularVariable.ModularVariable* property), 459
 - faceNormals () (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 278
 - faceNormals () (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 279
 - faceNormals () (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 279
 - facesBack () (*fipy.meshes.abstractMesh.AbstractMesh* property), 250

`facesBottom()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 250
`facesDown()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 250
`facesFront()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 251
`facesLeft()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 251
`facesRight()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 251
`facesTop()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 251
`facesUp()` (*fipy.meshes.abstractMesh.AbstractMesh* property), 252
`FaceTerm` (class in *fipy.terms.faceTerm*), 340
`faceValue()` (*fipy.variables.CellVariable* property), 492
`faceValue()` (*fipy.variables.cellVariable.CellVariable* property), 441
`FaceVariable`, 112
`FaceVariable` (class in *fipy.variables*), 496
`FaceVariable` (class in *fipy.variables.faceVariable*), 451
`faceVertexIDs()` (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 279
`faceVertexIDs()` (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 279
`failFn()` (*fipy.steps.steps.Stepper* static method), 328
`figaspect()` (*fipy.viewers.matplotlibViewer.matplotlibViewer* method), 533
`finalize_options()` (*fipy.tests.test.test* method), 379
FiPy, 99
`fipy.boundaryConditions` module, 240
`fipy.boundaryConditions.boundaryCondition` module, 237
`fipy.boundaryConditions.constraint` module, 238
`fipy.boundaryConditions.fixedFlux` module, 238
`fipy.boundaryConditions.fixedValue` module, 238
`fipy.boundaryConditions.nthOrderBoundaryCondition` module, 239
`fipy.boundaryConditions.test` module, 239
`fipy.matrices` module, 243
`fipy.matrices.offsetSparseMatrix` module, 243
`fipy.matrices.petscMatrix` module, 243
`fipy.matrices.scipyMatrix` module, 243
`fipy.matrices.sparseMatrix` module, 243
`fipy.matrices.test` module, 243
`fipy.meshes` module, 280
`fipy.meshes.abstractMesh` module, 246
`fipy.meshes.builders` module, 245
`fipy.meshes.builders.abstractGridBuilder` module, 245
`fipy.meshes.builders.grid1DBuilder` module, 245
`fipy.meshes.builders.grid2DBuilder` module, 245
`fipy.meshes.builders.grid3DBuilder` module, 245
`fipy.meshes.builders.periodicGrid1DBuilder` module, 245
`fipy.meshes.builders.utilityClasses` module, 245
`fipy.meshes.cylindricalGrid1D` module, 253
`fipy.meshes.cylindricalGrid2D` module, 253
`fipy.meshes.cylindricalNonUniformGrid1D` module, 255
`fipy.meshes.cylindricalNonUniformGrid2D` module, 255
`fipy.meshes.cylindricalUniformGrid1D` module, 256
`fipy.meshes.cylindricalUniformGrid2D` module, 256
`fipy.meshes.factoryMeshes` module, 257
`fipy.meshes.gmshMesh` module, 259
`fipy.meshes.grid1D` module, 265
`fipy.meshes.grid2D` module, 265
`fipy.meshes.grid3D` module, 265
`fipy.meshes.mesh` module, 265
`fipy.meshes.mesh1D` module, 266
`fipy.meshes.mesh2D` module, 267
`fipy.meshes.nonUniformGrid1D` module, 269

fipy.meshes.nonUniformGrid2D
 module, 269
 fipy.meshes.nonUniformGrid3D
 module, 270
 fipy.meshes.periodicGrid1D
 module, 271
 fipy.meshes.periodicGrid2D
 module, 272
 fipy.meshes.periodicGrid3D
 module, 273
 fipy.meshes.representations
 module, 245
 fipy.meshes.representations.abstractRepresentation
 module, 245
 fipy.meshes.representations.gridRepresentation
 module, 245
 fipy.meshes.representations.meshRepresentation
 module, 245
 fipy.meshes.skewedGrid2D
 module, 276
 fipy.meshes.test
 module, 276
 fipy.meshes.topologies
 module, 246
 fipy.meshes.topologies.abstractTopology
 module, 246
 fipy.meshes.topologies.gridTopology
 module, 246
 fipy.meshes.topologies.meshTopology
 module, 246
 fipy.meshes.tri2D
 module, 276
 fipy.meshes.uniformGrid
 module, 277
 fipy.meshes.uniformGrid1D
 module, 278
 fipy.meshes.uniformGrid2D
 module, 278
 fipy.meshes.uniformGrid3D
 module, 279
 fipy.numerix
 module, 228
 fipy.solvers
 module, 321
 fipy.solvers.petsc
 module, 298
 fipy.solvers.petsc.comms
 module, 294
 fipy.solvers.petsc.comms.parallelPETScCommWrapper
 module, 293
 fipy.solvers.petsc.comms.petscCommWrapper
 module, 294
 fipy.solvers.petsc.comms.serialPETScCommWrapper
 module, 294
 fipy.solvers.petsc.dummySolver
 module, 294
 fipy.solvers.petsc.linearBicgSolver
 module, 295
 fipy.solvers.petsc.linearCGSSolver
 module, 295
 fipy.solvers.petsc.linearGMRESSolver
 module, 295
 fipy.solvers.petsc.linearLUSolver
 module, 296
 fipy.solvers.petsc.linearPCGSolver
 module, 296
 fipy.solvers.petsc.petscKrylovSolver
 module, 297
 fipy.solvers.petsc.petscSolver
 module, 297
 fipy.solvers.pyAMG
 module, 303
 fipy.solvers.pyAMG.linearCGSSolver
 module, 300
 fipy.solvers.pyAMG.linearGeneralSolver
 module, 301
 fipy.solvers.pyAMG.linearGMRESSolver
 module, 301
 fipy.solvers.pyAMG.linearLUSolver
 module, 302
 fipy.solvers.pyAMG.linearPCGSolver
 module, 302
 fipy.solvers.pyAMG.preconditioners
 module, 300
 fipy.solvers.pyAMG.preconditioners.smoothedAggregation
 module, 300
 fipy.solvers.pyamgx
 module, 310
 fipy.solvers.pyamgx.aggregationAMGSolver
 module, 305
 fipy.solvers.pyamgx.classicalAMGSolver
 module, 306
 fipy.solvers.pyamgx.linearBiCGStabSolver
 module, 307
 fipy.solvers.pyamgx.linearCGSolver
 module, 307
 fipy.solvers.pyamgx.linearFGMRESSolver
 module, 308
 fipy.solvers.pyamgx.linearGMRESSolver
 module, 309
 fipy.solvers.pyamgx.linearLUSolver
 module, 309
 fipy.solvers.pyamgx.preconditioners
 module, 305
 fipy.solvers.pyamgx.preconditioners.preconditioners
 module, 305
 fipy.solvers.pyamgx.pyAMGXSolver
 module, 310

<code>fipy.solvers.pyamgx.smoothers</code> module, 305	<code>fipy.terms.diffusionTerm</code> module, 336
<code>fipy.solvers.pyamgx.smoothers.smoothers</code> module, 305	<code>fipy.terms.diffusionTermCorrection</code> module, 337
<code>fipy.solvers.scipy</code> module, 316	<code>fipy.terms.diffusionTermNoCorrection</code> module, 337
<code>fipy.solvers.scipy.linearBicgstabSolver</code> module, 313	<code>fipy.terms.explicitDiffusionTerm</code> module, 337
<code>fipy.solvers.scipy.linearCGSSolver</code> module, 314	<code>fipy.terms.explicitSourceTerm</code> module, 338
<code>fipy.solvers.scipy.linearGMRESSolver</code> module, 314	<code>fipy.terms.explicitUpwindConvectionTerm</code> module, 338
<code>fipy.solvers.scipy.linearLUSolver</code> module, 315	<code>fipy.terms.exponentialConvectionTerm</code> module, 339
<code>fipy.solvers.scipy.linearPCGSolver</code> module, 315	<code>fipy.terms.faceTerm</code> module, 340
<code>fipy.solvers.scipy.scipyKrylovSolver</code> module, 316	<code>fipy.terms.firstOrderAdvectionTerm</code> module, 341
<code>fipy.solvers.scipy.scipySolver</code> module, 316	<code>fipy.terms.hybridConvectionTerm</code> module, 342
<code>fipy.solvers.solver</code> module, 319	<code>fipy.terms.implicitDiffusionTerm</code> module, 344
<code>fipy.solvers.test</code> module, 321	<code>fipy.terms.implicitDiffusionTerm.DiffusionTerm</code> object, 131
<code>fipy.steppers</code> module, 328	<code>fipy.terms.implicitSourceTerm</code> module, 344
<code>fipy.steppers.pidStepper</code> module, 327	<code>fipy.terms.nonDiffusionTerm</code> module, 344
<code>fipy.steppers.pseudoRKQSStepper</code> module, 327	<code>fipy.terms.powerLawConvectionTerm</code> module, 344
<code>fipy.steppers.stepper</code> module, 328	<code>fipy.terms.residualTerm</code> module, 346
<code>fipy.terms</code> module, 354	<code>fipy.terms.sourceTerm</code> module, 346
<code>fipy.terms.abstractBinaryTerm</code> module, 331	<code>fipy.terms.term</code> module, 347
<code>fipy.terms.abstractConvectionTerm</code> module, 331	<code>fipy.terms.test</code> module, 350
<code>fipy.terms.abstractDiffusionTerm</code> module, 331	<code>fipy.terms.transientTerm</code> module, 350
<code>fipy.terms.abstractUpwindConvectionTerm</code> module, 331	<code>fipy.terms.transientTerm.TransientTerm</code> object, 131
<code>fipy.terms.advectionTerm</code> module, 331	<code>fipy.terms.unaryTerm</code> module, 351
<code>fipy.terms.asymmetricConvectionTerm</code> module, 334	<code>fipy.terms.upwindConvectionTerm</code> module, 351
<code>fipy.terms.binaryTerm</code> module, 334	<code>fipy.terms.vanLeerConvectionTerm</code> module, 353
<code>fipy.terms.cellTerm</code> module, 334	<code>fipy.tests</code> module, 379
<code>fipy.terms.centralDiffConvectionTerm</code> module, 334	<code>fipy.tests.doctestPlus</code> module, 377
<code>fipy.terms.coupledBinaryTerm</code> module, 336	<code>fipy.tests.lateImportTest</code> module, 378

fipy.tests.test
 module, 378
 fipy.tests.testProgram
 module, 379
 fipy.tools
 module, 417
 fipy.tools.comms
 module, 382
 fipy.tools.comms.abstractCommWrapper
 module, 381
 fipy.tools.comms.dummyComm
 module, 382
 fipy.tools.debug
 module, 408
 fipy.tools.decorators
 module, 408
 fipy.tools.dimensions
 module, 408
 fipy.tools.dimensions.DictWithDefault
 module, 382
 fipy.tools.dimensions.NumberDict
 module, 382
 fipy.tools.dimensions.physicalField
 module, 382
 fipy.tools.dump
 module, 183, 408
 fipy.tools.inline
 module, 409
 fipy.tools.numerix
 module, 409
 fipy.tools.parser
 module, 180, 416
 fipy.tools.test
 module, 416
 fipy.tools.vector
 module, 416
 fipy.tools.vitals
 module, 417
 fipy.variables
 module, 478
 fipy.variables.addOverFacesVariable
 module, 435
 fipy.variables.arithmeticCellToFaceVariable
 module, 435
 fipy.variables.betaNoiseVariable
 module, 435
 fipy.variables.binaryOperatorVariable
 module, 437
 fipy.variables.cellToFaceVariable
 module, 437
 fipy.variables.cellVariable
 module, 437
 fipy.variables.cellVariable.CellVariable
 object, 130
 fipy.variables.constant
 module, 445
 fipy.variables.constraintMask
 module, 445
 fipy.variables.coupledCellVariable
 module, 445
 fipy.variables.distanceVariable
 module, 445
 fipy.variables.exponentialNoiseVariable
 module, 449
 fipy.variables.faceGradContributionsVariable
 module, 451
 fipy.variables.faceGradVariable
 module, 451
 fipy.variables.faceVariable
 module, 451
 fipy.variables.gammaNoiseVariable
 module, 453
 fipy.variables.gaussCellGradVariable
 module, 455
 fipy.variables.gaussianNoiseVariable
 module, 455
 fipy.variables.harmonicCellToFaceVariable
 module, 457
 fipy.variables.histogramVariable
 module, 457
 fipy.variables.interfaceAreaVariable
 module, 458
 fipy.variables.interfaceFlagVariable
 module, 458
 fipy.variables.leastSquaresCellGradVariable
 module, 458
 fipy.variables.levelSetDiffusionVariable
 module, 458
 fipy.variables.meshVariable
 module, 458
 fipy.variables.minmodCellToFaceVariable
 module, 458
 fipy.variables.modCellGradVariable
 module, 458
 fipy.variables.modCellToFaceVariable
 module, 458
 fipy.variables.modFaceGradVariable
 module, 458
 fipy.variables.modPhysicalField
 module, 458
 fipy.variables.modularVariable
 module, 458
 fipy.variables.noiseVariable
 module, 460
 fipy.variables.operatorVariable
 module, 461
 fipy.variables.scharfetterGummelFaceVariable
 module, 461

fipy.variables.surfactantConvectionVariable object, 132
 module, 462
 fipy.variables.surfactantVariable
 module, 464
 fipy.variables.test
 module, 466
 fipy.variables.unaryOperatorVariable
 module, 466
 fipy.variables.uniformNoiseVariable
 module, 466
 fipy.variables.variable
 module, 468
 fipy.viewers, 230
 module, 107, 128, 140, 144, 146, 150, 164, 180, 182, 212, 556
 fipy.viewers.matplotlibViewer
 module, 534
 fipy.viewers.matplotlibViewer.matplotlibViewer
 module, 519
 fipy.viewers.matplotlibViewer.matplotlibViewer.FixedFluxViewer
 module, 521
 fipy.viewers.matplotlibViewer.matplotlibViewer.FixedFluxContourViewer (class in *fipy.boundaryConditions.fixedFlux*), 238
 module, 522
 fipy.viewers.matplotlibViewer.matplotlibViewer.FixedValueViewer (class in *fipy.boundaryConditions.fixedValue*), 238
 module, 524
 fipy.viewers.matplotlibViewer.matplotlibViewer.FixedValue (class in *fipy.boundaryConditions.fixedValue*), 238
 module, 525
 fipy.viewers.matplotlibViewer.matplotlibViewer.SparseMatrixViewer (class in *fipy.tools.physicalField.PhysicalField* method), 395
 module, 527
 fipy.viewers.matplotlibViewer.matplotlibViewer.StreamlineViewer (class in *fipy.tools.physicalField.PhysicalField* method), 427
 module, 527
 fipy.viewers.matplotlibViewer.matplotlibViewer.GammaFactorViewer
 module, 530
 fipy.viewers.matplotlibViewer.matplotlibViewer.GammaNoiseVariable (class in *fipy.variables.gammaNoiseVariable*), 453
 module, 532
 fipy.viewers.matplotlibViewer.test
 module, 534
 fipy.viewers.mayaviViewer
 module, 548
 fipy.viewers.mayaviViewer.mayaviClient
 module, 545
 fipy.viewers.mayaviViewer.mayaviDaemon
 module, 547
 fipy.viewers.mayaviViewer.test
 module, 548
 fipy.viewers.multiViewer
 module, 553
 fipy.viewers.test
 module, 553
 fipy.viewers.testinteractive
 module, 553
 fipy.viewers.tsvViewer
 module, 553
 fipy.viewers.tsvViewer.TSVViewer

[getNearestCell\(\)](#) (*fipy.meshes.abstractMesh.AbstractMesh* method), 252
[getNearestCell\(\)](#) (*fipy.meshes.histogramVariable* class in *fipy.variables.histogramVariable*), 457
[getscType\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 395
[getscType\(\)](#) (*fipy.tools.physicalField* method), 427
[getscType\(\)](#) (*fipy.variables.Variable* method), 486
[getscType\(\)](#) (*fipy.variables.variable.Variable* method), 475
[getShape\(\)](#) (in module *fipy.tools.numerix*), 413
[getUnit\(\)](#) (in module *fipy.tools.numerix*), 413
[Gist1DViewer](#), 230
[globalValue\(\)](#) (*fipy.variables.CellVariable* property), 493
[globalValue\(\)](#) (*fipy.variables.cellVariable.CellVariable* property), 442
[globalValue\(\)](#) (*fipy.variables.FaceVariable* property), 497
[globalValue\(\)](#) (*fipy.variables.faceVariable.FaceVariable* property), 452
[Gmsh](#), 99
[Gmsh2D](#) (class in *fipy.meshes*), 287
[Gmsh2D](#) (class in *fipy.meshes.gmshMesh*), 259
[Gmsh2DIn3DSpace](#) (class in *fipy.meshes*), 291
[Gmsh2DIn3DSpace](#) (class in *fipy.meshes.gmshMesh*), 263
[Gmsh3D](#) (class in *fipy.meshes*), 291
[Gmsh3D](#) (class in *fipy.meshes.gmshMesh*), 263
[GmshGrid2D](#) (class in *fipy.meshes*), 292
[GmshGrid2D](#) (class in *fipy.meshes.gmshMesh*), 264
[GmshGrid3D](#) (class in *fipy.meshes*), 292
[GmshGrid3D](#) (class in *fipy.meshes.gmshMesh*), 264
[grad\(\)](#) (*fipy.variables.CellVariable* property), 493
[grad\(\)](#) (*fipy.variables.cellVariable.CellVariable* property), 442
[grad\(\)](#) (*fipy.variables.ModularVariable* property), 499
[grad\(\)](#) (*fipy.variables.modularVariable.ModularVariable* property), 459
[Grid1D](#), 139, 143, 145, 150, 158, 167, 177, 200, 229
[Grid1D\(\)](#) (in module *fipy.meshes*), 281
[Grid1D\(\)](#) (in module *fipy.meshes.factoryMeshes*), 258
[Grid2D](#), 127, 180, 197, 198, 210, 227
[Grid2D\(\)](#) (in module *fipy.meshes*), 280
[Grid2D\(\)](#) (in module *fipy.meshes.factoryMeshes*), 257
[Grid2DGistViewer](#), 228
[Grid3D\(\)](#) (in module *fipy.meshes*), 280
[Grid3D\(\)](#) (in module *fipy.meshes.factoryMeshes*), 257
H
[harmonicFaceValue\(\)](#) (*fipy.variables.CellVariable* property), 493
[harmonicFaceValue\(\)](#) (*fipy.variables.cellVariable.CellVariable* property), 442
[HistogramVariable](#) (class in *fipy.variables*), 509
[ImplicitDiffusionTerm](#) (class in *fipy.terms*), 366
[ImplicitDiffusionTerm](#) (class in *fipy.terms.hybridConvectionTerm*), 342
[ImplicitDiffusionTerm](#) (in module *fipy.terms.implicitDiffusionTerm*), 344
[ImplicitSourceTerm](#), 153, 179, 181
[ImplicitSourceTerm](#) (class in *fipy.terms*), 361
[ImplicitSourceTerm](#) (class in *fipy.terms.implicitSourceTerm*), 344
[inBaseUnits\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 395
[inBaseUnits\(\)](#) (*fipy.tools.physicalField* method), 428
[inBaseUnits\(\)](#) (*fipy.variables.Variable* method), 486
[inBaseUnits\(\)](#) (*fipy.variables.variable.Variable* method), 476
[IncorrectSolutionVariable](#), 355
[inDimensionless\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 395
[inDimensionless\(\)](#) (*fipy.tools.physicalField* method), 428
[initialize_options\(\)](#) (*fipy.tests.test.test* method), 379
[inRadians\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 396
[inRadians\(\)](#) (*fipy.tools.physicalField* method), 428
[inSIUnits\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 396
[inSIUnits\(\)](#) (*fipy.tools.physicalField* method), 428
[interfaceVar\(\)](#) (*fipy.variables.SurfactantVariable* property), 511
[interfaceVar\(\)](#) (*fipy.variables.surfactantVariable.SurfactantVariable* property), 466
[interiorFaceCellIDs\(\)](#) (*fipy.meshes.abstractMesh.AbstractMesh* property), 252
[interiorFaceIDs\(\)](#) (*fipy.meshes.abstractMesh.AbstractMesh* property), 252
[interiorFaces\(\)](#) (*fipy.meshes.abstractMesh.AbstractMesh* property), 252
[inUnitsOf\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 396
[inUnitsOf\(\)](#) (*fipy.tools.physicalField* method), 429
[inUnitsOf\(\)](#) (*fipy.variables.Variable* method), 486

<code>inUnitsOf()</code> (<i>fiPy.variables.variable.Variable</i> method), 476	<code>LinearBicgSolver</code> (class in <i>fiPy.solvers.petsc.linearBicgSolver</i>), 295
<code>IPython</code> , 99	<code>LinearBiCGStabSolver</code> (class in <i>fiPy.solvers.pyamgx</i>), 311
<code>isAngle()</code> (<i>fiPy.tools.dimensions.physicalField.PhysicalUnit</i> method), 406	<code>LinearBiCGStabSolver</code> (class in <i>fiPy.solvers.pyamgx.linearBiCGStabSolver</i>), 311
<code>isclose()</code> (in module <i>fiPy.tools.numerix</i>), 410, 413	<code>LinearBicgstabSolver</code> (class in <i>fiPy.solvers.scipy</i>), 317
<code>isCompatible()</code> (<i>fiPy.tools.dimensions.physicalField.PhysicalUnit</i> method), 397	<code>LinearBicgstabSolver</code> (class in <i>fiPy.solvers.scipy.linearBicgstabSolver</i>), 313
<code>isCompatible()</code> (<i>fiPy.tools.dimensions.physicalField.PhysicalUnit</i> method), 406	<code>LinearCGSolver</code> (class in <i>fiPy.solvers.pyamgx</i>), 310
<code>isCompatible()</code> (<i>fiPy.tools.PhysicalField</i> method), 429	<code>LinearCGSolver</code> (class in <i>fiPy.solvers.pyamgx.linearCGSolver</i>), 307
<code>isDimensionless()</code> (<i>fiPy.tools.dimensions.physicalField.PhysicalUnit</i> method), 407	<code>LinearCGSSolver</code> (class in <i>fiPy.solvers</i>), 324
<code>isDimensionlessOrAngle()</code> (<i>fiPy.tools.dimensions.physicalField.PhysicalUnit</i> method), 407	<code>LinearCGSSolver</code> (class in <i>fiPy.solvers.petsc</i>), 299
<code>isFloat()</code> (in module <i>fiPy.tools.numerix</i>), 413	<code>LinearCGSSolver</code> (class in <i>fiPy.solvers.pyAMG</i>), 303
<code>isInt()</code> (in module <i>fiPy.tools.numerix</i>), 413	<code>LinearCGSSolver</code> (class in <i>fiPy.solvers.pyAMG.linearCGSSolver</i>), 300
<code>isInverseAngle()</code> (<i>fiPy.tools.dimensions.physicalField.PhysicalUnit</i> method), 407	<code>LinearCGSSolver</code> (class in <i>fiPy.solvers.scipy</i>), 316
<code>itemset()</code> (<i>fiPy.tools.dimensions.physicalField.PhysicalField</i> method), 397	<code>LinearCGSSolver</code> (class in <i>fiPy.solvers.scipy.linearCGSSolver</i>), 314
<code>itemset()</code> (<i>fiPy.tools.PhysicalField</i> method), 429	<code>LinearFGMRESSolver</code> (class in <i>fiPy.solvers.pyamgx</i>), 311
<code>itemset()</code> (<i>fiPy.variables.Variable</i> method), 487	<code>LinearFGMRESSolver</code> (class in <i>fiPy.solvers.pyamgx.linearFGMRESSolver</i>), 308
<code>itemset()</code> (<i>fiPy.variables.variable.Variable</i> method), 476	<code>LinearGeneralSolver</code> (class in <i>fiPy.solvers.pyAMG</i>), 304
<code>itemsizes()</code> (<i>fiPy.tools.dimensions.physicalField.PhysicalField</i> property), 397	<code>LinearGeneralSolver</code> (class in <i>fiPy.solvers.pyAMG.linearGeneralSolver</i>), 301
<code>itemsizes()</code> (<i>fiPy.tools.PhysicalField</i> property), 429	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers</i>), 324
<code>itemsizes()</code> (<i>fiPy.variables.Variable</i> property), 487	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers.petsc</i>), 299
<code>itemsizes()</code> (<i>fiPy.variables.variable.Variable</i> property), 476	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers.pyAMG</i>), 303
<code>Iterator</code> , 228	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers.pyAMG.linearGMRESSolver</i>), 301
J	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers</i>), 324
<code>justErrorVector()</code> (<i>fiPy.terms.term.Term</i> method), 348	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers.petsc.linearGMRESSolver</i>), 295
<code>justResidualVector()</code> (<i>fiPy.terms.term.Term</i> method), 348	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers.pyAMG</i>), 303
L	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers.pyAMG.linearGMRESSolver</i>), 301
<code>L1error()</code> (in module <i>fiPy.steps</i>), 328	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers.pyamgx.linearGMRESSolver</i>), 309
<code>L1norm()</code> (in module <i>fiPy.tools.numerix</i>), 414	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers.scipy</i>), 316
<code>L2error()</code> (in module <i>fiPy.steps</i>), 328	<code>LinearGMRESSolver</code> (class in <i>fiPy.solvers.scipy.linearGMRESSolver</i>), 314
<code>L2norm()</code> (in module <i>fiPy.tools.numerix</i>), 414	<code>LinearLUSolver</code> , 164, 227
<code>leastSquaresGrad()</code> (<i>fiPy.variables.CellVariable</i> property), 494	<code>LinearLUSolver</code> (class in <i>fiPy.solvers</i>), 323
<code>leastSquaresGrad()</code> (<i>fiPy.variables.cellVariable.CellVariable</i> property), 442	<code>LinearLUSolver</code> (class in <i>fiPy.solvers.petsc</i>), 298
<code>LinearBicgSolver</code> (class in <i>fiPy.solvers</i>), 324	<code>LinearLUSolver</code> (class in <i>fiPy.solvers.petsc.linearLUSolver</i>), 296
<code>LinearBicgSolver</code> (class in <i>fiPy.solvers.petsc</i>), 299	

- LinearLUSolver (class in *fipy.solvers.pyAMG*), 304
 LinearLUSolver (class in *fipy.solvers.pyAMG.linearLUSolver*), 302
 LinearLUSolver (class in *fipy.solvers.pyamgx*), 312
 LinearLUSolver (class in *fipy.solvers.pyamgx.linearLUSolver*), 309
 LinearLUSolver (class in *fipy.solvers.scipy*), 317
 LinearLUSolver (class in *fipy.solvers.scipy.linearLUSolver*), 315
 LinearPCGSolver (class in *fipy.solvers*), 324
 LinearPCGSolver (class in *fipy.solvers.petsc*), 298
 LinearPCGSolver (class in *fipy.solvers.petsc.linearPCGSolver*), 296
 LinearPCGSolver (class in *fipy.solvers.pyAMG*), 304
 LinearPCGSolver (class in *fipy.solvers.pyAMG.linearPCGSolver*), 302
 LinearPCGSolver (class in *fipy.solvers.scipy*), 317
 LinearPCGSolver (class in *fipy.solvers.scipy.linearPCGSolver*), 315
 LinearPCGSolver (in module *fipy.solvers.pyamgx*), 311
 LinearPCGSolver (in module *fipy.solvers.pyamgx.linearCGSolver*), 308
 LINFerror () (in module *fipy.steps*), 329
 LINFnorm () (in module *fipy.tools.numerix*), 414
 linux, 99
 loadtxt, 180, 182
 log, 163, 169
 log () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 397
 log () (*fipy.tools.PhysicalField* method), 429
 log () (*fipy.viewers.Matplotlib1DViewer* property), 558
 log () (*fipy.viewers.matplotlibViewer.Matplotlib1DViewer* property), 536
 log () (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 520
 log () (*fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer* property), 533
 log10 () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 397
 log10 () (*fipy.tools.PhysicalField* method), 430
M
 macOS, 99
 mag () (*fipy.variables.Variable* property), 487
 mag () (*fipy.variables.variable.Variable* property), 476
 Matplotlib, 99
 Matplotlib1DViewer (class in *fipy.viewers*), 557
 Matplotlib1DViewer (class in *fipy.viewers.matplotlibViewer*), 535
 Matplotlib1DViewer (class in *fipy.viewers.matplotlibViewer.matplotlib1DViewer*), 519
 Matplotlib2DContourViewer (class in *fipy.viewers.matplotlibViewer.matplotlib2DContourViewer*), 521
 Matplotlib2DGridContourViewer (class in *fipy.viewers*), 559
 Matplotlib2DGridContourViewer (class in *fipy.viewers.matplotlibViewer*), 537
 Matplotlib2DGridContourViewer (class in *fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer*), 522
 Matplotlib2DGridViewer (class in *fipy.viewers*), 558
 Matplotlib2DGridViewer (class in *fipy.viewers.matplotlibViewer*), 536
 Matplotlib2DGridViewer (class in *fipy.viewers.matplotlibViewer.matplotlib2DGridViewer*), 524
 Matplotlib2DViewer (class in *fipy.viewers*), 561
 Matplotlib2DViewer (class in *fipy.viewers.matplotlibViewer*), 539
 Matplotlib2DViewer (class in *fipy.viewers.matplotlibViewer.matplotlib2DViewer*), 525
 MatplotlibSparseMatrixViewer (class in *fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer*), 527
 MatplotlibStreamViewer (class in *fipy.viewers*), 564
 MatplotlibStreamViewer (class in *fipy.viewers.matplotlibViewer*), 542
 MatplotlibStreamViewer (class in *fipy.viewers.matplotlibViewer.matplotlibStreamViewer*), 527
 MatplotlibVectorViewer (class in *fipy.viewers*), 540
 MatplotlibVectorViewer (class in *fipy.viewers.matplotlibViewer*), 540
 MatplotlibVectorViewer (class in *fipy.viewers.matplotlibViewer.matplotlibVectorViewer*), 530
 MatplotlibViewer () (in module *fipy.viewers*), 556
 MatplotlibViewer () (in module *fipy.viewers.matplotlibViewer*), 534
 matrix, 212
 matrix () (*fipy.terms.term.Term* property), 349
 MatrixIllConditionedWarning, 320, 322
 max () (*fipy.variables.Variable* method), 487
 max () (*fipy.variables.variable.Variable* method), 476
 MaxAll () (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 293
 MaxAll () (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 381
 MaximumIterationWarning, 319, 321
 MayaVi, 99

Mayavi, [99](#)
 MayaviClient (class in *fipy.viewers*), [566](#)
 MayaviClient (class in *fipy.viewers.mayaviViewer*), [548](#)
 MayaviClient (class in *fipy.viewers.mayaviViewer.mayaviClient*), [545](#)
 MayaviDaemon (class in *fipy.viewers.mayaviViewer.mayaviDaemon*), [547](#)
 Mesh (class in *fipy.meshes.mesh*), [265](#)
 Mesh1D (class in *fipy.meshes.mesh1D*), [266](#)
 Mesh2D (class in *fipy.meshes.mesh2D*), [267](#)
 MeshAdditionError, [265](#)
 MeshDimensionError, [571](#)
 method1 () (package.subpackage.base.Base method), [234](#)
 method2 () (package.subpackage.base.Base method), [234](#)
 method2 () (package.subpackage.object.Object method), [236](#)
 min () (fipy.variables.Variable method), [487](#)
 min () (fipy.variables.variable.Variable method), [476](#)
 MinAll () (fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper method), [293](#)
 MinAll () (fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper method), [381](#)
 minmodFaceValue () (fipy.variables.CellVariable property), [494](#)
 minmodFaceValue () (fipy.variables.cellVariable.CellVariable property), [443](#)
 ModularVariable, [181](#)
 ModularVariable (class in *fipy.variables*), [498](#)
 ModularVariable (class in *fipy.variables.modularVariable*), [458](#)
 module
 examples.cahnHilliard.mesh2DCoupled, [203](#)
 examples.cahnHilliard.sphere, [206](#)
 examples.convection.exponential1D.mesh1D, [143](#)
 examples.convection.exponential1DSource.mesh1D, [145](#)
 examples.convection.robin, [146](#)
 examples.convection.source, [148](#)
 examples.diffusion.anisotropy, [141](#)
 examples.diffusion.circle, [130](#)
 examples.diffusion.coupled, [125](#)
 examples.diffusion.electrostatics, [134](#)
 examples.diffusion.mesh1D, [105](#)
 examples.diffusion.mesh20x20, [127](#)
 examples.diffusion.nthOrder.input4thOrder1D, [245](#)
 examples.flow.stokesCavity, [209](#)
 examples.levelSet.advection.circle, [201](#)
 examples.levelSet.advection.mesh1D, [199](#)
 examples.levelSet.distanceFunction.circle, [198](#)
 examples.levelSet.distanceFunction.mesh1D, [197](#)
 examples.phase.anisotropy, [174](#)
 examples.phase.binaryCoupled, [158](#)
 examples.phase.impingement.mesh20x20, [180](#)
 examples.phase.impingement.mesh40x1, [177](#)
 examples.phase.polyxtal, [184](#)
 examples.phase.polyxtalCoupled, [190](#)
 examples.phase.quaternary, [167](#)
 examples.phase.simple, [149](#)
 examples.reactiveWetting.liquidVapor1D, [215](#)
 examples.updating.update0_1to1_0, [216](#)
 examples.updating.update1_0to2_0, [221](#)
 examples.updating.update2_0to3_0, [221](#)
 fipy.boundaryConditions, [240](#)
 fipy.boundaryConditions.boundaryCondition, [237](#)
 fipy.boundaryConditions.constraint, [238](#)
 fipy.boundaryConditions.fixedFlux, [238](#)
 fipy.boundaryConditions.fixedValue, [238](#)
 fipy.boundaryConditions.nthOrderBoundaryCondition, [239](#)
 fipy.boundaryConditions.test, [239](#)
 fipy.matrices, [243](#)
 fipy.matrices.offsetSparseMatrix, [243](#)
 fipy.matrices.petscMatrix, [243](#)
 fipy.matrices.scipyMatrix, [243](#)
 fipy.matrices.sparseMatrix, [243](#)
 fipy.matrices.test, [243](#)
 fipy.meshes, [280](#)
 fipy.meshes.abstractMesh, [246](#)
 fipy.meshes.builders, [245](#)
 fipy.meshes.builders.abstractGridBuilder, [245](#)
 fipy.meshes.builders.grid1DBuilder, [245](#)

fipy.meshes.builders.grid2DBuilder, 245
 fipy.meshes.builders.grid3DBuilder, 245
 fipy.meshes.builders.periodicGrid1DBuilder, 245
 fipy.meshes.builders.utilityClasses, 245
 fipy.meshes.cylindricalGrid1D, 253
 fipy.meshes.cylindricalGrid2D, 253
 fipy.meshes.cylindricalNonUniformGrid1D, 253
 fipy.meshes.cylindricalNonUniformGrid2D, 255
 fipy.meshes.cylindricalUniformGrid1D, 256
 fipy.meshes.cylindricalUniformGrid2D, 256
 fipy.meshes.factoryMeshes, 257
 fipy.meshes.gmshMesh, 259
 fipy.meshes.grid1D, 265
 fipy.meshes.grid2D, 265
 fipy.meshes.grid3D, 265
 fipy.meshes.mesh, 265
 fipy.meshes.mesh1D, 266
 fipy.meshes.mesh2D, 267
 fipy.meshes.nonUniformGrid1D, 269
 fipy.meshes.nonUniformGrid2D, 269
 fipy.meshes.nonUniformGrid3D, 270
 fipy.meshes.periodicGrid1D, 271
 fipy.meshes.periodicGrid2D, 272
 fipy.meshes.periodicGrid3D, 273
 fipy.meshes.representations, 245
 fipy.meshes.representations.abstractRepresentation, 245
 fipy.meshes.representations.gridRepresentation, 245
 fipy.meshes.representations.meshRepresentation, 245
 fipy.meshes.skewedGrid2D, 276
 fipy.meshes.test, 276
 fipy.meshes.topologies, 246
 fipy.meshes.topologies.abstractTopology, 246
 fipy.meshes.topologies.gridTopology, 246
 fipy.meshes.topologies.meshTopology, 246
 fipy.meshes.tri2D, 276
 fipy.meshes.uniformGrid, 277
 fipy.meshes.uniformGrid1D, 278
 fipy.meshes.uniformGrid2D, 278
 fipy.meshes.uniformGrid3D, 279
 fipy.numerix, 228
 fipy.solvers, 321
 fipy.solvers.petsc, 298
 fipy.solvers.petsc.comms, 294
 fipy.solvers.petsc.comms.parallelPETScCommWrapper, 294
 fipy.solvers.petsc.comms.petscCommWrapper, 294
 fipy.solvers.petsc.comms.serialPETScCommWrapper, 294
 fipy.solvers.petsc.dummySolver, 294
 fipy.solvers.petsc.linearBicgSolver, 295
 fipy.solvers.petsc.linearCGSSolver, 295
 fipy.solvers.petsc.linearGMRESSolver, 295
 fipy.solvers.petsc.linearLUSolver, 296
 fipy.solvers.petsc.linearPCGSolver, 296
 fipy.solvers.petsc.petscKrylovSolver, 297
 fipy.solvers.petsc.petscSolver, 297
 fipy.solvers.pyAMG, 303
 fipy.solvers.pyAMG.linearCGSSolver, 300
 fipy.solvers.pyAMG.linearGeneralSolver, 301
 fipy.solvers.pyAMG.linearGMRESSolver, 301
 fipy.solvers.pyAMG.linearLUSolver, 302
 fipy.solvers.pyAMG.linearPCGSolver, 302
 fipy.solvers.pyAMG.preconditioners, 302
 fipy.solvers.pyAMG.preconditioners.smoothedAggr, 302
 fipy.solvers.pyamgx, 310
 fipy.solvers.pyamgx.aggregationAMGSolver, 305
 fipy.solvers.pyamgx.classicalAMGSolver, 306
 fipy.solvers.pyamgx.linearBiCGStabSolver, 307
 fipy.solvers.pyamgx.linearCGSolver, 307
 fipy.solvers.pyamgx.linearFGMRESSolver, 308
 fipy.solvers.pyamgx.linearGMRESSolver, 309
 fipy.solvers.pyamgx.linearLUSolver, 309
 fipy.solvers.pyamgx.preconditioners,

[305](#)
[fipy.solvers.pyamgx.preconditioners.preconditioners,](#)
[305](#)
[fipy.solvers.pyamgx.pyAMGXSolver,](#)
[310](#)
[fipy.solvers.pyamgx.smoothers,](#) [305](#)
[fipy.solvers.pyamgx.smoothers.smoothers,](#)
[305](#)
[fipy.solvers.scipy,](#) [316](#)
[fipy.solvers.scipy.linearBicgstabSolver,](#)
[313](#)
[fipy.solvers.scipy.linearCGSSolver,](#)
[314](#)
[fipy.solvers.scipy.linearGMRESSolver,](#)
[314](#)
[fipy.solvers.scipy.linearLUSolver,](#)
[315](#)
[fipy.solvers.scipy.linearPCGSolver,](#)
[315](#)
[fipy.solvers.scipy.scipyKrylovSolver,](#)
[316](#)
[fipy.solvers.scipy.scipySolver,](#) [316](#)
[fipy.solvers.solver,](#) [319](#)
[fipy.solvers.test,](#) [321](#)
[fipy.steppers,](#) [328](#)
[fipy.steppers.pidStepper,](#) [327](#)
[fipy.steppers.pseudoRKQSSolver,](#) [327](#)
[fipy.steppers.stepper,](#) [328](#)
[fipy.terms,](#) [354](#)
[fipy.terms.abstractBinaryTerm,](#) [331](#)
[fipy.terms.abstractConvectionTerm,](#)
[331](#)
[fipy.terms.abstractDiffusionTerm,](#)
[331](#)
[fipy.terms.abstractUpwindConvectionTerm,](#)
[331](#)
[fipy.terms.advectionTerm,](#) [331](#)
[fipy.terms.asymmetricConvectionTerm,](#)
[334](#)
[fipy.terms.binaryTerm,](#) [334](#)
[fipy.terms.cellTerm,](#) [334](#)
[fipy.terms.centralDiffConvectionTerm,](#)
[334](#)
[fipy.terms.coupledBinaryTerm,](#) [336](#)
[fipy.terms.diffusionTerm,](#) [336](#)
[fipy.terms.diffusionTermCorrection,](#)
[337](#)
[fipy.terms.diffusionTermNoCorrection,](#)
[337](#)
[fipy.terms.explicitDiffusionTerm,](#)
[337](#)
[fipy.terms.explicitSourceTerm,](#) [338](#)
[fipy.terms.explicitUpwindConvectionTerm,](#)
[338](#)
[fipy.terms.exponentialConvectionTerm,](#)
[339](#)
[fipy.terms.faceTerm,](#) [340](#)
[fipy.terms.firstOrderAdvectionTerm,](#)
[341](#)
[fipy.terms.hybridConvectionTerm,](#) [342](#)
[fipy.terms.implicitDiffusionTerm,](#)
[344](#)
[fipy.terms.implicitSourceTerm,](#) [344](#)
[fipy.terms.nonDiffusionTerm,](#) [344](#)
[fipy.terms.powerLawConvectionTerm,](#)
[344](#)
[fipy.terms.residualTerm,](#) [346](#)
[fipy.terms.sourceTerm,](#) [346](#)
[fipy.terms.term,](#) [347](#)
[fipy.terms.test,](#) [350](#)
[fipy.terms.transientTerm,](#) [350](#)
[fipy.terms.unaryTerm,](#) [351](#)
[fipy.terms.upwindConvectionTerm,](#) [351](#)
[fipy.terms.vanLeerConvectionTerm,](#)
[353](#)
[fipy.tests,](#) [379](#)
[fipy.tests.doctestPlus,](#) [377](#)
[fipy.tests.lateImportTest,](#) [378](#)
[fipy.tests.test,](#) [378](#)
[fipy.tests.testProgram,](#) [379](#)
[fipy.tools,](#) [417](#)
[fipy.tools.comms,](#) [382](#)
[fipy.tools.comms.abstractCommWrapper,](#)
[381](#)
[fipy.tools.comms.dummyComm,](#) [382](#)
[fipy.tools.debug,](#) [408](#)
[fipy.tools.decorators,](#) [408](#)
[fipy.tools.dimensions,](#) [408](#)
[fipy.tools.dimensions.DictWithDefault,](#)
[382](#)
[fipy.tools.dimensions.NumberDict,](#)
[382](#)
[fipy.tools.dimensions.physicalField,](#)
[382](#)
[fipy.tools.dump,](#) [183](#), [408](#)
[fipy.tools.inline,](#) [409](#)
[fipy.tools.numerix,](#) [409](#)
[fipy.tools.parser,](#) [180](#), [416](#)
[fipy.tools.test,](#) [416](#)
[fipy.tools.vector,](#) [416](#)
[fipy.tools.vitals,](#) [417](#)
[fipy.variables,](#) [478](#)
[fipy.variables.addOverFacesVariable,](#)
[435](#)
[fipy.variables.arithmeticCellToFaceVariable,](#)
[435](#)
[fipy.variables.betaNoiseVariable,](#)
[435](#)

fipy.variables.binaryOperatorVariable, 437
 fipy.variables.cellToFaceVariable, 437
 fipy.variables.cellVariable, 437
 fipy.variables.constant, 445
 fipy.variables.constraintMask, 445
 fipy.variables.coupledCellVariable, 445
 fipy.variables.distanceVariable, 445
 fipy.variables.exponentialNoiseVariable, 449
 fipy.variables.faceGradContributionsVariable, 451
 fipy.variables.faceGradVariable, 451
 fipy.variables.faceVariable, 451
 fipy.variables.gammaNoiseVariable, 453
 fipy.variables.gaussCellGradVariable, 455
 fipy.variables.gaussianNoiseVariable, 455
 fipy.variables.harmonicCellToFaceVariable, 457
 fipy.variables.histogramVariable, 457
 fipy.variables.interfaceAreaVariable, 458
 fipy.variables.interfaceFlagVariable, 458
 fipy.variables.leastSquaresCellGradVariable, 458
 fipy.variables.levelSetDiffusionVariable, 458
 fipy.variables.meshVariable, 458
 fipy.variables.minmodCellToFaceVariable, 458
 fipy.variables.modCellGradVariable, 458
 fipy.variables.modCellToFaceVariable, 458
 fipy.variables.modFaceGradVariable, 458
 fipy.variables.modPhysicalField, 458
 fipy.variables.modularVariable, 458
 fipy.variables.noiseVariable, 460
 fipy.variables.operatorVariable, 461
 fipy.variables.scharfetterGummelFaceVariable, 461
 fipy.variables.surfactantConvectionVariable, 462
 fipy.variables.surfactantVariable, 464
 fipy.variables.test, 466
 fipy.variables.unaryOperatorVariable, 466
 fipy.variables.uniformNoiseVariable, 466
 fipy.variables.variable, 468
 fipy.viewers, 107, 128, 140, 144, 146, 150, 164, 180, 182, 212, 556
 fipy.viewers.matplotlibViewer, 534
 fipy.viewers.matplotlibViewer.matplotlib1DViewer, 519
 fipy.viewers.matplotlibViewer.matplotlib2DContour, 521
 fipy.viewers.matplotlibViewer.matplotlib2DGridC, 522
 fipy.viewers.matplotlibViewer.matplotlib2DGridV, 524
 fipy.viewers.matplotlibViewer.matplotlib2DView, 525
 fipy.viewers.matplotlibViewer.matplotlibSparseM, 527
 fipy.viewers.matplotlibViewer.matplotlibStreamV, 527
 fipy.viewers.matplotlibViewer.matplotlibVectorV, 530
 fipy.viewers.matplotlibViewer.matplotlibViewer, 532
 fipy.viewers.matplotlibViewer.test, 534
 fipy.viewers.mayaviViewer, 548
 fipy.viewers.mayaviViewer.mayaviClient, 548
 fipy.viewers.mayaviViewer.mayaviDaemon, 547
 fipy.viewers.mayaviViewer.test, 548
 fipy.viewers.multiViewer, 553
 fipy.viewers.test, 553
 fipy.viewers.testinteractive, 553
 fipy.viewers.tsvViewer, 553
 fipy.viewers.viewer, 555
 fipy.viewers.vtkViewer, 552
 fipy.viewers.vtkViewer.test, 550
 fipy.viewers.vtkViewer.vtkCellViewer, 550
 fipy.viewers.vtkViewer.vtkFaceViewer, 551
 fipy.viewers.vtkViewer.vtkViewer, 551
 package.subpackage, 236
 package.subpackage.base, 233
 package.subpackage.object, 235
 scipy, 133, 157, 164
 viewers, 172
 MPI, 99
 mpi4py, 99

[mpi4py_comm\(\)](#) (*fiPy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper* property), 294
[multiply\(\)](#) (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 398
[multiply\(\)](#) (*fiPy.tools.PhysicalField* method), 430
[MultiViewer](#) (class in *fiPy.viewers*), 568
[MultiViewer](#) (class in *fiPy.viewers.multiViewer*), 553
N
[name\(\)](#) (*fiPy.tools.dimensions.physicalField.PhysicalUnit* method), 407
[name\(\)](#) (*fiPy.variables.Variable* property), 487
[name\(\)](#) (*fiPy.variables.variable.Variable* property), 476
[nearest\(\)](#) (in module *fiPy.tools.numerix*), 414
[NoiseVariable](#) (class in *fiPy.variables.noiseVariable*), 460
[NonUniformGrid1D](#) (class in *fiPy.meshes.nonUniformGrid1D*), 269
[NonUniformGrid2D](#) (class in *fiPy.meshes.nonUniformGrid2D*), 269
[NonUniformGrid3D](#) (class in *fiPy.meshes.nonUniformGrid3D*), 270
[Norm2\(\)](#) (*fiPy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper* method), 294
[Norm2\(\)](#) (*fiPy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 381
[Nproc\(\)](#) (*fiPy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* property), 293
[Nproc\(\)](#) (*fiPy.solvers.petsc.comms.serialPETScCommWrapper.SerialPETScCommWrapper* property), 294
[Nproc\(\)](#) (*fiPy.tools.comms.abstractCommWrapper.AbstractCommWrapper* property), 381
[Nproc\(\)](#) (*fiPy.tools.comms.dummyComm.DummyComm* property), 382
[NthOrderBoundaryCondition](#), 139
[NthOrderBoundaryCondition](#) (class in *fiPy.boundaryConditions*), 240
[NthOrderBoundaryCondition](#) (class in *fiPy.boundaryConditions.nthOrderBoundaryCondition*), 239
[numarray](#), 100
[Numeric](#), 100
[numericValue\(\)](#) (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 547
[numericValue\(\)](#) (*fiPy.tools.PhysicalField* property), 430
[numericValue\(\)](#) (*fiPy.variables.Variable* property), 487
[numericValue\(\)](#) (*fiPy.variables.variable.Variable* property), 476
[NumPy](#), 100
O
[object](#) (class in *package.subpackage.object*), 235
[OffsetSparseMatrix\(\)](#) (in module *fiPy.matrices.offsetSparseMatrix*), 243
[old\(\)](#) (*fiPy.variables.CellVariable* property), 494
[old\(\)](#) (*fiPy.variables.cellVariable.CellVariable* property), 443
[in OpenMP](#), 100
[openMSHFile\(\)](#) (in module *fiPy.meshes*), 287
[in openMSHFile\(\)](#) (in module *fiPy.meshes.gmshMesh*), 259
[in openPOSFile\(\)](#) (in module *fiPy.meshes*), 287
[openPOSFile\(\)](#) (in module *fiPy.meshes.gmshMesh*), 259
P
[package.subpackage](#)
[package.subpackage.base](#)
[package.subpackage.object](#)
[pandas](#), 100
[ParallelPETScCommWrapper](#) (class in *fiPy.solvers.petsc.comms.parallelPETScCommWrapper*), 293
[parallelRandom\(\)](#) (*fiPy.variables.GaussianNoiseVariable* method), 507
[parallelRandom\(\)](#) (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 457
[parallelRandom\(\)](#) (*fiPy.variables.noiseVariable.NoiseVariable* method), 460
[parse\(\)](#) (in module *fiPy.tools.parser*), 416
[parse_command_line\(\)](#) (*fiPy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 547
[PeriodicGrid1D](#) (class in *fiPy.meshes*), 282
[PeriodicGrid1D](#) (class in *fiPy.meshes.periodicGrid1D*), 271
[PeriodicGrid2D](#) (class in *fiPy.meshes*), 282
[PeriodicGrid2D](#) (class in *fiPy.meshes.periodicGrid2D*), 272
[PeriodicGrid2DLeftRight](#) (class in *fiPy.meshes*), 283
[PeriodicGrid2DLeftRight](#) (class in *fiPy.meshes.periodicGrid2D*), 273

PeriodicGrid2DTopBottom (class in *fipy.meshes*), 283
 PeriodicGrid2DTopBottom (class in *fipy.meshes.periodicGrid2D*), 273
 PeriodicGrid3D (class in *fipy.meshes*), 284
 PeriodicGrid3D (class in *fipy.meshes.periodicGrid3D*), 273
 PeriodicGrid3DFrontBack (class in *fipy.meshes*), 285
 PeriodicGrid3DFrontBack (class in *fipy.meshes.periodicGrid3D*), 275
 PeriodicGrid3DLeftRight (class in *fipy.meshes*), 285
 PeriodicGrid3DLeftRight (class in *fipy.meshes.periodicGrid3D*), 274
 PeriodicGrid3DLeftRightFrontBack (class in *fipy.meshes*), 285
 PeriodicGrid3DLeftRightFrontBack (class in *fipy.meshes.periodicGrid3D*), 275
 PeriodicGrid3DLeftRightTopBottom (class in *fipy.meshes*), 285
 PeriodicGrid3DLeftRightTopBottom (class in *fipy.meshes.periodicGrid3D*), 275
 PeriodicGrid3DTopBottom (class in *fipy.meshes*), 285
 PeriodicGrid3DTopBottom (class in *fipy.meshes.periodicGrid3D*), 274
 PeriodicGrid3DTopBottomFrontBack (class in *fipy.meshes*), 285
 PeriodicGrid3DTopBottomFrontBack (class in *fipy.meshes.periodicGrid3D*), 275
 PETSc, 100
 petsc4py, 100
 PETScCommWrapper (class in *fipy.solvers.petsc.comms.petscCommWrapper*), 294
 PETScKrylovSolver (class in *fipy.solvers.petsc.petscKrylovSolver*), 297
 PETScSolver (class in *fipy.solvers.petsc.petscSolver*), 297
 PhysicalField (class in *fipy.tools*), 417
 PhysicalField (class in *fipy.tools.dimensions.physicalField*), 385
 physicalShape () (*fipy.meshes.skewedGrid2D* property), 286
 physicalShape () (*fipy.meshes.skewedGrid2D.skewedGrid2D* property), 276
 physicalShape () (*fipy.meshes.Tri2D* property), 287
 physicalShape () (*fipy.meshes.tri2D.Tri2D* property), 277
 PhysicalUnit (class in *fipy.tools.dimensions.physicalField*), 401
 pi, 175, 180, 181
 PIDStepper (class in *fipy.steppers.pidStepper*), 327
 pip, 100
 plot () (*fipy.viewers.DummyViewer* method), 571
 plot () (*fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.MatplotlibViewer* method), 527
 plot () (*fipy.viewers.matplotlibViewer.matplotlibViewer.AbstractMatplotlibViewer* method), 533
 plot () (*fipy.viewers.MayaviClient* method), 568
 plot () (*fipy.viewers.mayaviViewer.MayaviClient* method), 550
 plot () (*fipy.viewers.mayaviViewer.mayaviClient.MayaviClient* method), 547
 plot () (*fipy.viewers.MultiViewer* method), 569
 plot () (*fipy.viewers.multiViewer.MultiViewer* method), 553
 plot () (*fipy.viewers.TSVViewer* method), 570
 plot () (*fipy.viewers.tsvViewer.TSVViewer* method), 554
 plot () (*fipy.viewers.viewer.AbstractViewer* method), 555
 plot () (*fipy.viewers.vtkViewer.vtkViewer.VTKViewer* method), 551
 plotMesh () (*fipy.viewers.viewer.AbstractViewer* method), 555
 poll_file () (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 548
 PowerLawConvectionTerm, 170
 PowerLawConvectionTerm (class in *fipy.terms*), 367
 PowerLawConvectionTerm (class in *fipy.terms.powerLawConvectionTerm*), 344
 PreconditionerNotPositiveDefiniteWarning, 320, 322
 PreconditionerWarning, 320, 321
 PRINT () (in module *fipy.tools.debug*), 408
 printPackageInfo () (*fipy.tests.test.test* method), 379
 procID () (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* property), 294
 procID () (*fipy.solvers.petsc.comms.serialPETScCommWrapper.SerialPETScCommWrapper* property), 294
 procID () (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* property), 382
 procID () (*fipy.tools.comms.dummyComm.DummyComm* property), 382
 prune () (in module *fipy.tools.vector*), 416
 PseudoRKQSStepper (class in *fipy.steppers.pseudoRKQSStepper*), 327
 put () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 398
 put () (*fipy.tools.PhysicalField* method), 430
 put () (*fipy.variables.Variable* method), 487
 put () (*fipy.variables.variable.Variable* method), 476
 put () (in module *fipy.tools.numerix*), 410, 411, 414
 putAdd () (in module *fipy.tools.vector*), 416
 PyAMG, 100

- pyamgx, [100](#)
- PyAMGXSolver (class in *fipy.solvers.pyamgx.pyAMGXSolver*), [310](#)
- PyPI, [100](#)
- Pyrex, [100](#)
- Pyparse, [100](#)
- Python, [100](#)
- Python 3, [100](#)
- Python Enhancement Proposals PEP 3000, [100](#)
- PyTrilinos, [100](#)
- PyxViewer, [100](#)
- Q**
- quiver() (*fipy.viewers.MatplotlibVectorViewer* method), [564](#)
- quiver() (*fipy.viewers.matplotlibViewer.MatplotlibVectorViewer* method), [542](#)
- quiver() (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* method), [532](#)
- R**
- random() (*fipy.variables.BetaNoiseVariable* method), [501](#)
- random() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), [252](#)
- random() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), [437](#)
- random() (*fipy.variables.ExponentialNoiseVariable* method), [503](#)
- random() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), [451](#)
- random() (*fipy.variables.GammaNoiseVariable* method), [505](#)
- random() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), [454](#)
- random() (*fipy.variables.noiseVariable.NoiseVariable* method), [460](#)
- random() (*fipy.variables.UniformNoiseVariable* method), [509](#)
- random() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), [467](#)
- rank() (in module *fipy.tools.numerix*), [415](#)
- ravel() (*fipy.tools.dimensions.physicalField.PhysicalField* method), [398](#)
- ravel() (*fipy.tools.PhysicalField* method), [430](#)
- ravel() (*fipy.variables.Variable* method), [487](#)
- ravel() (*fipy.variables.variable.Variable* method), [476](#)
- read() (in module *fipy.tools.dump*), [409](#)
- register_skipper() (in module *fipy.tests.doctestPlus*), [377](#)
- release() (*fipy.variables.CellVariable* method), [495](#)
- release() (*fipy.variables.cellVariable.CellVariable* method), [443](#)
- release() (*fipy.variables.Variable* method), [487](#)
- release() (*fipy.variables.variable.Variable* method), [476](#)
- report_skips() (in module *fipy.tests.doctestPlus*), [377](#)
- reshape() (*fipy.tools.dimensions.physicalField.PhysicalField* method), [398](#)
- reshape() (*fipy.tools.PhysicalField* method), [431](#)
- reshape() (in module *fipy.tools.numerix*), [411](#), [412](#), [415](#)
- ResidualTerm (class in *fipy.terms*), [362](#)
- ResidualTerm (class in *fipy.terms.residualTerm*), [346](#)
- residualVectorAndNorm() (*fipy.terms.term.Term* method), [349](#)
- RHSvector, [212](#)
- RHSvector() (*fipy.terms.term.Term* property), [347](#)
- run() (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), [548](#)
- run_tests() (*fipy.tests.test.test* method), [379](#)
- S**
- save() (*fipy.tools.Vitals* method), [433](#)
- save() (*fipy.tools.vitals.Vitals* method), [417](#)
- ScalarQuantityOutOfRangeWarning, [320](#), [322](#)
- scale() (*fipy.meshes.abstractMesh.AbstractMesh* method), [252](#)
- scaledCellDistances() (*fipy.meshes.abstractMesh.AbstractMesh* property), [252](#)
- scaledCellDistances() (*fipy.meshes.abstractMesh.AbstractMesh* property), [252](#)
- scaledCellVolumes() (*fipy.meshes.abstractMesh.AbstractMesh* property), [252](#)
- scaledFaceAreas() (*fipy.meshes.abstractMesh.AbstractMesh* property), [252](#)
- scaledFaceToCellDistances() (*fipy.meshes.abstractMesh.AbstractMesh* property), [252](#)
- ScharfetterGummelFaceVariable (class in *fipy.variables*), [498](#)
- ScharfetterGummelFaceVariable (class in *fipy.variables.scharfetterGummelFaceVariable*), [461](#)
- ScientificPython, [100](#)
- SciPy, [100](#)
- scipy module, [133](#), [157](#), [164](#)
- scramble() (*fipy.variables.noiseVariable.NoiseVariable* method), [461](#)
- SerialPETScCommWrapper (class in *fipy.solvers.petsc.comms.serialPETScCommWrapper*), [294](#)

- [setLimits\(\)](#) (*fipy.viewers.MultiViewer* method), 569
[setLimits\(\)](#) (*fipy.viewers.multiViewer.MultiViewer* method), 553
[setLimits\(\)](#) (*fipy.viewers.viewer.AbstractViewer* method), 555
[setName\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 407
[setup_source\(\)](#) (*fipy.viewers.mayaviViewer.mayaviDaemonMayaviDaemon* method), 548
[setValue\(\)](#) (*fipy.variables.CellVariable* method), 495
[setValue\(\)](#) (*fipy.variables.cellVariable.CellVariable* method), 444
[setValue\(\)](#) (*fipy.variables.FaceVariable* method), 497
[setValue\(\)](#) (*fipy.variables.faceVariable.FaceVariable* method), 452
[setValue\(\)](#) (*fipy.variables.Variable* method), 487
[setValue\(\)](#) (*fipy.variables.variable.Variable* method), 477
[shape\(\)](#) (*fipy.meshes.SkewedGrid2D* property), 286
[shape\(\)](#) (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 276
[shape\(\)](#) (*fipy.meshes.Tri2D* property), 287
[shape\(\)](#) (*fipy.meshes.tri2D.Tri2D* property), 277
[shape\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* property), 398
[shape\(\)](#) (*fipy.tools.PhysicalField* property), 431
[shape\(\)](#) (*fipy.variables.Variable* property), 488
[shape\(\)](#) (*fipy.variables.variable.Variable* property), 477
[sign\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 399
[sign\(\)](#) (*fipy.tools.PhysicalField* method), 431
[sin\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 399
[sin\(\)](#) (*fipy.tools.PhysicalField* method), 431
[sinh\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 399
[sinh\(\)](#) (*fipy.tools.PhysicalField* method), 431
[SkewedGrid2D](#) (class in *fipy.meshes*), 286
[SkewedGrid2D](#) (class in *fipy.meshes.skewedGrid2D*), 276
[SmoothedAggregationPreconditioner](#) (class in *fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner*), 300
[SolutionVariableNumberError](#), 354
[SolutionVariableRequiredError](#), 355
[solve](#), 164
[solve\(\)](#) (*fipy.terms.term.Term* method), 349
[Solver](#) (class in *fipy.solvers*), 322
[Solver](#) (class in *fipy.solvers.solver*), 320
[solver](#) (*fipy.solvers.LinearBicgSolver* attribute), 324
[solver](#) (*fipy.solvers.LinearCGSSolver* attribute), 324
[solver](#) (*fipy.solvers.LinearGMRESSolver* attribute), 324
[solver](#) (*fipy.solvers.LinearPCGSolver* attribute), 324
[solver](#) (*fipy.solvers.petsc.LinearBicgSolver* attribute), 299
[solver](#) (*fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver* attribute), 295
[solver](#) (*fipy.solvers.petsc.LinearCGSSolver* attribute), 299
[solver](#) (*fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver* attribute), 295
[solver](#) (*fipy.solvers.petsc.LinearGMRESSolver* attribute), 299
[solver](#) (*fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver* attribute), 296
[solver](#) (*fipy.solvers.petsc.LinearPCGSolver* attribute), 299
[solver](#) (*fipy.solvers.petsc.linearPCGSolver.LinearPCGSolver* attribute), 296
[SolverConvergenceWarning](#), 319, 321
[SourceTerm](#) (class in *fipy.terms.sourceTerm*), 346
[Sphinx](#), 100
[sqrt](#), 151, 180
[arcsin; cos](#), 133
[sqrt\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 399
[sqrt\(\)](#) (*fipy.tools.PhysicalField* method), 431
[sqrtDot\(\)](#) (in module *fipy.tools.numerix*), 415
[StagnatedSolverWarning](#), 320, 322
[std\(\)](#) (*fipy.variables.Variable* method), 488
[std\(\)](#) (*fipy.variables.variable.Variable* method), 478
[SteadyConvectionDiffusionScEquation](#), 227
[step\(\)](#) (*fipy.steps.steps.Stepper* method), 328
[Stepper](#) (class in *fipy.steps.steps*), 328
[subscribedVariables\(\)](#) (*fipy.variables.Variable* property), 488
[subscribedVariables\(\)](#) (*fipy.variables.variable.Variable* property), 478
[subtract\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 399
[subtract\(\)](#) (*fipy.tools.PhysicalField* method), 432
[successFn\(\)](#) (*fipy.steps.steps.Stepper* static method), 328
[sum\(\)](#) (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 294
[sum\(\)](#) (*fipy.tools.comms.abstractCommWrapper.AbstractCommWrapper* method), 382
[sum\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 400
[sum\(\)](#) (*fipy.tools.PhysicalField* method), 432
[sum\(\)](#) (*fipy.variables.Variable* method), 488
[sum\(\)](#) (*fipy.variables.variable.Variable* method), 478
[sum\(\)](#) (in module *fipy.tools.numerix*), 411, 412, 415
[SurfactantConvectionVariable](#) (class in *fipy.variables*), 511

SurfactantConvectionVariable (class in *fipy.variables.surfactantConvectionVariable*), 462

SurfactantVariable (class in *fipy.variables*), 509

SurfactantVariable (class in *fipy.variables.surfactantVariable*), 464

svn() (*fipy.tools.Vitals* method), 433

svn() (*fipy.tools.vitals.Vitals* method), 417

svncmd() (*fipy.tools.Vitals* method), 433

svncmd() (*fipy.tools.vitals.Vitals* method), 417

sweep, 156, 164, 212

sweep() (*fipy.terms.term.Term* method), 349

sweepFn() (*fipy.steppers stepper.Stepper* static method), 328

sweepMonotonic() (in module *fipy.steppers*), 329

T

take() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 400

take() (*fipy.tools.PhysicalField* method), 432

take() (*fipy.variables.Variable* method), 488

take() (*fipy.variables.variable.Variable* method), 478

take() (in module *fipy.tools.numerix*), 411, 412, 415

tan, 175

tan() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 400

tan() (*fipy.tools.PhysicalField* method), 432

tanh, 151

tanh() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 400

tanh() (*fipy.tools.PhysicalField* method), 433

Term (class in *fipy.terms.term*), 347

TermMultiplyError, 354

test (class in *fipy.tests.test*), 378

testmod() (in module *fipy.tests.doctestPlus*), 377

tostring() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 400

tostring() (*fipy.tools.PhysicalField* method), 433

tostring() (*fipy.variables.Variable* method), 488

tostring() (*fipy.variables.variable.Variable* method), 478

tostring() (in module *fipy.tools.numerix*), 415

TransientTerm, 107, 152, 179, 181

TransientTerm (class in *fipy.terms*), 359

TransientTerm (class in *fipy.terms.transientTerm*), 350

TravisCI, 100

Tri2D (class in *fipy.meshes*), 286

Tri2D (class in *fipy.meshes.tri2D*), 276

Trilinos, 100

TSVViewer (class in *fipy.viewers*), 569

TSVViewer (class in *fipy.viewers.tsvViewer*), 553

tupleToXML() (*fipy.tools.Vitals* method), 433

tupleToXML() (*fipy.tools.vitals.Vitals* method), 417

U

UniformGrid (class in *fipy.meshes.uniformGrid*), 277

UniformGrid1D (class in *fipy.meshes.uniformGrid1D*), 278

UniformGrid2D (class in *fipy.meshes.uniformGrid2D*), 278

UniformGrid3D (class in *fipy.meshes.uniformGrid3D*), 279

UniformNoiseVariable (class in *fipy.variables*), 507

UniformNoiseVariable (class in *fipy.variables.uniformNoiseVariable*), 466

unit() (*fipy.tools.dimensions.physicalField.PhysicalField* property), 401

unit() (*fipy.tools.PhysicalField* property), 433

unit() (*fipy.variables.Variable* property), 488

unit() (*fipy.variables.variable.Variable* property), 478

update_pipeline() (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 548

updateOld() (*fipy.variables.CellVariable* method), 496

updateOld() (*fipy.variables.cellVariable.CellVariable* method), 444

updateOld() (*fipy.variables.ModularVariable* method), 499

updateOld() (*fipy.variables.modularVariable.ModularVariable* method), 459

UpwindConvectionTerm (class in *fipy.terms*), 368

UpwindConvectionTerm (class in *fipy.terms.upwindConvectionTerm*), 351

user_options (*fipy.tests.test.test* attribute), 379

V

value() (*fipy.variables.Variable* property), 488

value() (*fipy.variables.variable.Variable* property), 478

VanLeerConvectionTerm (class in *fipy.terms*), 370

VanLeerConvectionTerm (class in *fipy.terms.vanLeerConvectionTerm*), 353

Variable, 155, 159

Variable (class in *fipy.variables*), 478

Variable (class in *fipy.variables.variable*), 468

VectorCoeffError, 354

vertexCoords() (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 278

vertexCoords() (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 279

vertexCoords() (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 279

view_data() (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 548

Viewer() (in module *fipy.viewers*), 571

viewers

module, 172
 Vitals (*class in fipy.tools*), 433
 Vitals (*class in fipy.tools.vitals*), 417
 VTKCellDataSet () (*fipy.meshes.abstractMesh.AbstractMesh property*), 246
 VTKCellViewer (*class in fipy.viewers*), 570
 VTKCellViewer (*class in fipy.viewers.vtkViewer*), 552
 VTKCellViewer (*class in fipy.viewers.vtkViewer.vtkCellViewer*), 550
 VTKFaceDataSet () (*fipy.meshes.abstractMesh.AbstractMesh property*), 246
 VTKFaceViewer (*class in fipy.viewers*), 571
 VTKFaceViewer (*class in fipy.viewers.vtkViewer*), 552
 VTKFaceViewer (*class in fipy.viewers.vtkViewer.vtkFaceViewer*), 551
 VTKViewer (*class in fipy.viewers.vtkViewer.vtkViewer*), 551
 VTKViewer () (*in module fipy.viewers*), 570
 VTKViewer () (*in module fipy.viewers.vtkViewer*), 552

W

Weave, 100
 Windows, 100
 write () (*in module fipy.tools.dump*), 408

X

x () (*fipy.meshes.abstractMesh.AbstractMesh property*), 252

Y

y () (*fipy.meshes.abstractMesh.AbstractMesh property*), 252

Z

z () (*fipy.meshes.abstractMesh.AbstractMesh property*), 253