

FiPy Manual

Release 4.0.1+0.gcde50d518.dirty

**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

Sep 16, 2025

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	Using FiPy	15
4	Solvers	31
5	Viewers	45
6	Frequently Asked Questions	47
7	Efficiency	57
8	Theoretical and Numerical Background	59
9	Design and Implementation	67
10	Virtual Kinetics of Materials Laboratory	73
11	Contributors	75
12	Publications	77
13	Presentations	79
14	Change Log	81
15	Git practices	109
16	Continuous Integration	111
17	Conda Lockfiles	113
18	README-like documents	115
19	Making a Release	117
20	Glossary	121

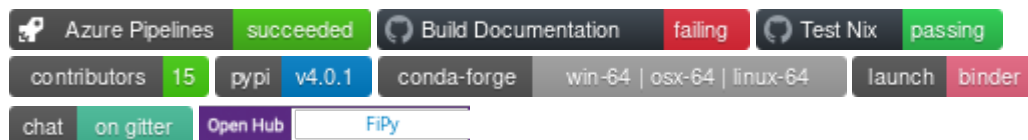
II	Examples	125
21	Selected Examples	129
III	fipy Package Documentation	133
22	How to Read the Modules Documentation	135
23	fipy	139
24	examples	959
	Bibliography	1231
	Python Module Index	1233
	Index	1239

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 [1] [2] [3] treatments of polycrystalline, dendritic, and electrochemical phase transformations, as well as drug eluting stents [4], reactive wetting [5], photovoltaics [6] and a level set treatment of the electrodeposition process [7].

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

We offer several modes to communicate with the *FiPy* developers and with other users.

1.3.1 Contact

In order to discuss *FiPy* with other users and with the developers, we encourage you to use one of the following modes of communication. We monitor all of these, so there is no need to post to more than one of them.

You may want to read the following resource about asking effective questions: <http://www.catb.org/~esr/faqs/smart-questions.html>

If you are having trouble, we are able to offer much more effective help if you provide a [minimal reproducible example](#).

GitHub Discussions

<https://github.com/usnistgov/fipy/discussions>

Suitable for open-ended conversations, troubleshooting, showing off...

If a discussion highlights a bug or feature request, it's easy for us to migrate *GitHub Discussions* to *GitHub Issues*.

GitHub Issues

<https://github.com/usnistgov/fipy/issues>

Suitable for bug reports, feature requests, and patch submissions.

StackOverflow

<https://stackoverflow.com/questions/tagged/fipy>

Suitable for questions that (probably) have definitive answers ("How do I...?"). It doesn't work so well for back-and-forth conversations, which are better suited to *GitHub Discussions*. Further, it's [bad at math](#) and they tend to delete answers that link to our existing documentation, meaning that we'd need to expend considerable effort, using an inferior tool, to duplicate things we've already written.

Seriously, use *GitHub Discussions*.

Mailing List

Attention: The mailing list is deprecated. Please use *GitHub Discussions*, instead.

You can sign up for the mailing list by sending a [subscription email](mailto:fipy+subscribe@list.nist.gov) to `<mailto:fipy+subscribe@list.nist.gov>`.

Once you are subscribed, you can post messages to the list simply by addressing email to `<mailto:fipy@list.nist.gov>`.

To get off the list, send a message to `<mailto:fipy+unsubscribe@list.nist.gov>`.

Send a message to `<mailto:fipy+help@list.nist.gov>` to learn other mailing list configurations you can change.

The list is hosted as a Google group. If you are subscribed with a Google account, you can interact with the list, configure your subscription, and see the archives at <https://list.nist.gov/fipy>.

List Archive

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

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

Older messages are archived at <https://www.mail-archive.com/fipy@nist.gov/>.

(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](#).)

We welcome collaborative efforts on this project.

1.4 Conventions and Notation

FiPy is driven by *Python* script files than 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.

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

Anac^onda.org 3.4.5

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 different versions of *Python*, 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*.

2.2.1 Install conda

Install `conda` or install `micromamba` on your computer.

2.2.2 Create a conda environment

Use one of the following methods to create a self-contained `conda` environment and then download and populate the environment with the prerequisites for *FiPy* from the `conda-forge` channel at <https://anaconda.org>. See [this discussion](#) of the merits of and relationship between the different methods.

- `Conda` environment files

This option is the most upgradable in the future and probably the best for development.

```
$ conda env create --name <MYFIPYENV> \  
  --file environments/<SOLVER>-environment.yml
```

Note: You can try to include multiple solver suites using `conda env update`, but be aware that different suites may have incompatible requirements, or may restrict installation to obsolete versions of Python. Given that *FiPy* can only use one solver suite during a run, installing more than one solver in an environment isn't necessary.

Attention: Successively updating an environment can be unpredictable, as later packages may conflict with earlier ones. Unfortunately, `conda env create` [does not support multiple environment files](#).

Alternatively, combine the different `environments/<SOLVER>-environment.yml` files you wish to use, along with `environment.yml` files for any other packages you are interested in (`conda-merge` may prove useful). Then execute:

```
$ conda env create --name <MYFIPYENV> --file <MYMERGEDENVIRONMENT>.yml
```

- `conda-lock` lockfiles

This option will pin all the packages, so is the most reproducible, but not particularly upgradable. For most, this is the safest way to generate a *FiPy* environment that consistently works.

```
$ conda-lock install --name <MYFIPYENV> \  
  environments/locks/conda-<SOLVER>-lock.yml
```

or, to be really explicit (and obviating the need for `conda-lock`):

```
$ conda create --name <MYFIPYENV> \  
  --file environments/locks/conda-<SOLVER>-<PLATFORM>.lock
```

- Directly from `conda-forge`, picking and choosing desired packages

This option is the most flexible, but has the highest risk of missing or incompatible packages.

e.g.:

```
$ conda create --name <MYFIPYENV> --channel conda-forge \  
  python=3 numpy scipy matplotlib-base packaging mpich \  
  mpi4py petsc4py mayavi "gmsht <4.0|>=4.5.2"
```

2.2.3 Install FiPy

```
$ conda install --name <MYFIPYENV> --channel conda-forge fipy
```

Note: The `fipy conda-forge` package used to be “batteries included”, but we found this to be too fragile. It now only includes the bare minimum for *FiPy* to function.

2.2.4 Enable conda environment

Enable your new environment with:

```
$ conda activate <MYFIPYENV>
```

or:

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

or, on *Windows*:

```
$ activate <MYFIPYENV>
```

You’re now ready to move on to *Using FiPy*.

Note: `conda` can be quite slow to resolve all dependencies when performing an installation. You may wish to consider using the alternative `mamba` installation manager to speed things up.

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.

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*. Historical versions of *FiPy* can be obtained from <https://github.com/usnistgov/fipy/releases>.

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:

```
$ python -m pip install fipy
```

or by unpacking the archive and typing:

```
$ python -m pip install .
```

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

```
$ python -m pip install --editable .
```

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 3.10.10 | ...
...
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* no longer supports Python 2.

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 or $\geq 4.5.2$.

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 ([8]) 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.

Create a conda environment for development, followed by:

```
$ source activate <MYFIPYENV>
$ python -m pip install scikit-fmm
$ git clone https://github.com/usnistgov/fipy.git
$ cd fipy
$ python -m pip install --editable .
```

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 4.0.1+0.gcde50d518.dirty. (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](#) [9] 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. Go to [nix.dev](#) to get started with Nix.

Installing

Once you have a working Nix installation use:

```
$ nix develop
```

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

```
$ nix develop --command bash -c "fipy_test"
```

Note: The SciPy solvers are the only available solvers currently.

Chapter 3

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](#) [10] or with [Dive Into Python](#) [11]. Deeper insight into Python can be obtained from the [12].

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

3.1 Logging

Diagnostic information about a *FiPy* run can be obtained using the `logging` module. For example, at the beginning of your script, you can add:

```
>>> import logging
>>> log = logging.getLogger("fipy")
>>> console = logging.StreamHandler()
>>> console.setLevel(logging.INFO)
>>> log.addHandler(console)
```

in order to see informational messages in the terminal. To have more verbose debugging information save to a file:

```
>>> logfile = logging.FileHandler(filename="fipy.log")
>>> logfile.setLevel(logging.DEBUG)
>>> log.addHandler(logfile)

>>> log.setLevel(logging.DEBUG)
```

To restrict logging to, e.g., information about the *PETSc* solvers:

```
>>> petsc = logging.Filter('fipy.solvers.petsc')
>>> logfile.addFilter(petsc)
```

More complex configurations can be specified by setting the `FIPY_LOG_CONFIG` environment variable. In this case, it is not necessary to add any logging instructions to your own script. Example configuration files can be found in `FiPySource/fipy/tools/logging/`.

If *Solving in Parallel*, the `mpilogging` package enables reporting which MPI rank each log entry comes from. For example:

```
>>> from mpilogging import MPIScatteredFileHandler
>>> mpilog = MPIScatteredFileHandler(filepattern="fipy.%(mpirank)d_of_%(mpisize)d.log")
>>> mpilog.setLevel(logging.DEBUG)
>>> log.addHandler(mpilog)
```

will generate a unique log file for each MPI rank.

3.2 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:

```
$ fipy_test
```

Note: You may need to first run:

```
$ python -m build
```

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 *Continuous Integration* dashboard.

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

3.2.1 Parallel Tests

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

```
$ mpirun -np {# of processors} fipy_test --trilinos
```

or:

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

3.3 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.

3.3.1 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:

--petsc

Forces the use of the *PETSc* solvers.

--pyamgx

Forces the use of the *pyamgx* solvers.

--scipy

Forces the use of the *SciPy* solvers.

--trilinos

Forces the use of the *Trilinos* solvers.

--lsmlib

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

--skfmm

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

3.3.2 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_LOG_CONFIG

Specifies a *JSON*-formatted logging configuration file, suitable for passing to *logging.config.dictConfig()*. Example configuration files can be found in *FiPySource/fipy/tools/logging/*.

FIPY_SOLVERS

Forces the use of the specified suite of linear *Solvers*. Valid (case-insensitive) choices are “petsc”, “pyamgx”, “scipy”, and “trilinos”.

FIPY_DEFAULT_CRITERION

Changes the default solver *Convergence* criterion to the specified value. Valid choices are “legacy”, “unscaled”, “RHS”, “matrix”, “initial”, “solution”, “preconditioned”, “natural”, “default”. A value of “default” is admittedly circular, but it works.

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.

PETSC_OPTIONS

PETSc configuration options. Set to “-help” and run a script with *PETSc* solvers in order to see what options are possible. Ignored if solver is not *PETSc*.

3.4 Solver Suites

Numerical solution of partial differential equations calls for solving sparse linear algebra. *FiPy* supports several different *Solvers*. To the greatest extent possible, they have all been configured to do the “same thing”, but each presents different capabilities in terms of matrix preconditioning and overall performance tuning.

3.5 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 *PETSc* and *Trilinos*:

```
$ export OMP_NUM_THREADS=1
```

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

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

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

or:

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

instead of:

```
$ python myScript.py
```

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"
```

3.5.1 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. This issue is not limited to *FiPy* or the parallel solver suites it uses; see [Can There Be Too Much Parallelism?](#) for a deeper look at the different threading settings available and their effects.

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* threads $\times$ *MPI* ranks = *Slurm* tasks). *OpenMP* threading always slows down *FiPy* performance.

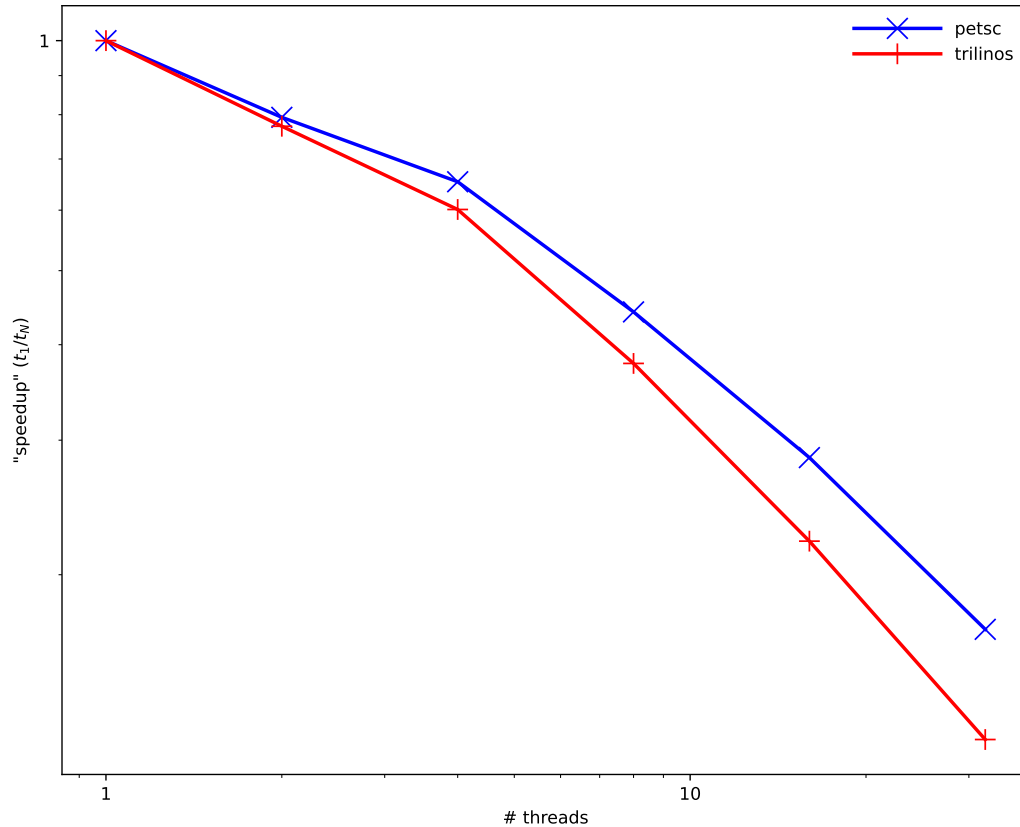


Fig. 1: Effect of having more *OpenMP* threads for each *MPI* rank¹.

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*TM.

¹ Calculations are of a [Method of Manufactured Solutions Allen-Cahn problem](#).

3.6 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.

3.7 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 equations 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*.

Attention: Coupled equations are not compatible with *Higher Order Diffusion* terms. This is not a practical limitation, as any higher order terms can be decomposed into multiple 2nd-order equations. For example, the pair of coupled Cahn-Hilliard & Allen-Cahn 4th- and 2nd-order equations

$$\begin{aligned}\frac{\partial C}{\partial t} &= \nabla \cdot \left[M \nabla \left(\frac{\partial f(c, \phi)}{\partial C} - \kappa_C \nabla^2 C \right) \right] \\ \frac{\partial \phi}{\partial t} &= -L \left(\frac{\partial f(c, \phi)}{\partial \phi} - \kappa_\phi \nabla^2 \phi \right)\end{aligned}$$

can be decomposed to three 2nd-order equations

$$\begin{aligned}\frac{\partial C}{\partial t} &= \nabla \cdot (M \nabla \mu) \\ \mu &= \frac{\partial f(c, \phi)}{\partial C} - \kappa_C \nabla^2 C \\ \frac{\partial \phi}{\partial t} &= -L \left(\frac{\partial f(c, \phi)}{\partial \phi} - \kappa_\phi \nabla^2 \phi \right)\end{aligned}$$

3.8 Boundary Conditions

3.8.1 Default boundary conditions

If no constraints are applied, solutions are conservative, i.e., all boundaries are zero flux. For the equation

$$\frac{\partial \phi}{\partial t} = \nabla \cdot (\vec{a} \phi) + \nabla \cdot (b \nabla \phi)$$

the condition on the boundary S is

$$\hat{n} \cdot (\vec{a} \phi + b \nabla \phi) = 0 \quad \text{on } S.$$

3.8.2 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.

3.8.3 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)
```

3.8.4 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.

3.8.5 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.

3.8.6 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](#)

3.8.7 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:

$$\begin{aligned} \phi_f &\approx \phi_P + (\vec{d}_{Pf} \cdot \nabla\phi)_f \\ &\approx \phi_P + (\hat{n} \cdot \nabla\phi)_f (\vec{d}_{Pf} \cdot \hat{n})_f \end{aligned} \tag{3.1}$$

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

Substituting this expression into the Robin condition:

$$\begin{aligned} \hat{n} \cdot (\vec{a}\phi + b\nabla\phi)_f &= g \\ \hat{n} \cdot \left[\vec{a}\phi_P + \vec{a}(\hat{n} \cdot \nabla\phi)_f (\vec{d}_{Pf} \cdot \hat{n})_f + b\nabla\phi \right]_f &\approx g \\ (\hat{n} \cdot \nabla\phi)_f &\approx \frac{g_f - (\hat{n} \cdot \vec{a})_f \phi_P}{(\vec{d}_{Pf} \cdot \vec{a})_f + b_f} \end{aligned} \tag{3.2}$$

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 (3.1)

$$\begin{aligned}\phi_f &\approx \phi_P + \frac{g_f - (\hat{n} \cdot \vec{a})_f \phi_P}{\left(\vec{d}_{Pf} \cdot \vec{a}\right)_f + b_f} \left(\vec{d}_{Pf} \cdot \hat{n}\right)_f \\ &\approx \frac{g_f \left(\hat{n} \cdot \vec{d}_{Pf}\right)_f + b_f \phi_P}{\left(\vec{d}_{Pf} \cdot \vec{a}\right)_f + b_f}\end{aligned}\tag{3.3}$$

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

Substituting (3.2) into the discretization of the *DiffusionTerm*:

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

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 (3.3):

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

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 $g \equiv h T_\infty$.

3.8.8 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 $\phi = 0.25$.

3.9 Adaptive Stepping

Step size can be controlled with the *steppystones* package. Demonstrations of its use are found in *examples.phase.binary* and *examples.phase.binaryCoupled*.

Note: The old *fipy.steps* classes are now deprecated. They were undocumented and did not work very well.

3.10 Manual

You can view the manual online at <http://pages.nist.gov/fipy> (a PDF version is available from the variant popup menu in the bottom right of those documentation pages). Alternatively, it may be possible to build a fresh copy by issuing the following command in the docs/ directory:

```
$ make html
```

or:

```
$ make latexpdf
```

Note: This mechanism is intended primarily for the developers. At a minimum, you will need [Sphinx](#), plus all of its prerequisites. We are currently building with Sphinx v7.0.

We install via conda:

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

Bibliographic citations require the *sphinxcontrib-bibtex* package:

```
$ python -m pip install sphinxcontrib-bibtex
```

Some documentation uses *numpydoc* styling:

```
$ python -m 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
$ python -m 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
```


Chapter 4

Solvers

FiPy requires either *PETSc*, *pyamgx*, *SciPy*, or *Trilinos* solver suites to be installed in order to solve linear systems. *PETSc* and *Trilinos* are the most complete of the solvers due to their numerous preconditioning and solver capabilities and they also allow *FiPy* to *run in parallel*. The *Python* interface for *PETSc* is better maintained than for *Trilinos* and tends to be easier to install. The sparse linear algebra solvers from the popular *SciPy* package are widely available and easy to install. Although they do not perform as well as the other suites and lack many of the features of *PETSc* or *Trilinos*, they may be the easiest linear solver choice when first installing and testing *FiPy*. *pyamgx* offers the possibility of solving sparse linear systems on the GPU; be aware that both hardware and software configuration is non-trivial. The *Pysparse* solver suite is no longer supported, as it requires *Python* 2.7; if you need *Pysparse*, please use *FiPy* 3.99.

FiPy chooses the solver suite based on system availability or based on the user supplied *Command-line Flags and Environment Variables*. For example, passing `--trilinos`:

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

uses a *SciPy* solver regardless of the command line argument.

Note: *FiPy* has not been designed to mix different solver suites during a given problem run

In the absence of *Command-line Flags and Environment Variables*, *FiPy*'s order of precedence when choosing the solver suite for generic solvers is *PETSc* followed by *Trilinos*, *SciPy*, and *pyamgx*.

Table 1: Solver Suite Features

	<i>PETSc</i>	<i>PyAMG</i>	<i>pyamgx</i>	<i>SciPy</i>	<i>Trilinos</i>
<i>FIPY_SOLVERS</i>	<code>petsc</code>		<code>pyamgx</code>	<code>scipy</code>	<code>trilinos</code>
solvers	✓		✓	✓	✓
preconditioners	✓	✓	✓	✓	✓
parallel	✓		¹		✓
<i>linux</i>	✓	✓	✓	✓	✓
<i>macOS</i>	✓	✓		✓	✓
<i>Windows</i>		✓		✓	
requirements	<code>petsc4py</code> , <code>mpi4py</code>	<code>pyamg</code> , <code>scipy</code>	<code>pyamgx</code> , <code>scipy</code> , <i>python</i> ≥ 3.5	<code>scipy</code>	<code>PyTrilinos</code> , <code>mpi4py</code> , <i>python</i> ≥3.7,<3.9 ²³

4.1 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).

4.2 PyAMG

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

The PyAMG package provides adaptive multigrid preconditioners that can be used in conjunction with the *SciPy* solvers. While PyAMG also has solvers, they are not currently implemented in *FiPy*.

4.3 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.

¹ While *AMGX* matrix solve takes advantage of GPU parallelism, the pyamgx library uses *SciPy* to build the matrix and thus suffers a significant serial bottleneck.

² *PyTrilinos* may be compatible with newer versions of Python, but these are the most recent versions we've been able to get to install using *conda* (3.7 on *linux* and 3.8 on *macOS*).

³ There is a more actively developed *PyTrilinos2* package, which may be compatible with more recent versions of *Python*, but *FiPy* does not yet work with it.

4.4 SciPy

<https://docs.scipy.org/doc/scipy/reference/sparse.html>

The `scipy.sparse` module provides a basic set of serial Krylov solvers and a limited collection of preconditioners.

4.5 Trilinos

<https://trilinos.github.io/>

Trilinos provides a complete set of sparse matrices, solvers, and preconditioners. Trilinos preconditioning allows for iterative solutions to some difficult problems, and it enables parallel execution of *FiPy* (see *Solving in Parallel* for more details).

4.6 Performance Comparison

Comparing different solver suites, or even different solvers, has historically been difficult. The different suites have different interpretations of *Convergence* and tolerance. *FiPy* 4.0 harmonizes the different suites so that, to the greatest extent possible, all interpret *Convergence* and tolerance the same way. In the course of doing this, a number of inefficiencies were found in the way that *FiPy* built sparse matrices. To see the impact of these changes, we examine the serial and parallel scaling performance of the different solver suites for two different benchmark problems.

Note: Although the legacy *Pysparse* solver suite is no longer supported, its performance is included here for reference.

4.6.1 Serial Performance

Serial performance is compared for the different suites.

The following plot shows the serial scaling behavior for the different solvers. We compare solution time vs number mesh cells for a diffusion problem.

We can see:

- For sufficiently large problems, building the matrix can be expected to scale as the number of cells N and solving the matrix should scale as $N \ln N$. There are not enough data points to differentiate these slopes.
- Below about 1000 cells, the time to prepare the matrix is insensitive to mesh size and this dominates the overall elapsed time.
- There is nearly three orders of magnitude between the fastest solver/preconditioner and the slowest. This particular problem is not especially sensitive to choice of solver and preconditioner, as preparing the matrix takes the majority of the overall time, but it can be worth optimizing the choice for more complex systems of equations.

⁴ *FiPy* version 3.4.4 has different interpretations of *Convergence* for different solver suites (and even for different solvers). Benchmarks used a patched version (371d28468) that provided more logging information and normalized interpretation of tolerance, but without any of the improvements in matrix and solver efficiency of version 4.0.

⁵ Calculations are of diffusion of a binary alloy in a frozen two-phase field. Solutions are on a square `Grid2D`. The initial condition is sampled from the center of a well-evolved 1024×1024 nucleation simulation. All available solvers and preconditioners are attempted. Solution tolerance is $1e-10$ using the "RHS" *convergence criterion*. Simulations were run on an AMD Epyc 7702 CPU with 64 cores featuring two-thread Simultaneous Multi-Threading (SMT) and 512 GB of memory. `OMP_NUM_THREADS` was set to 1.

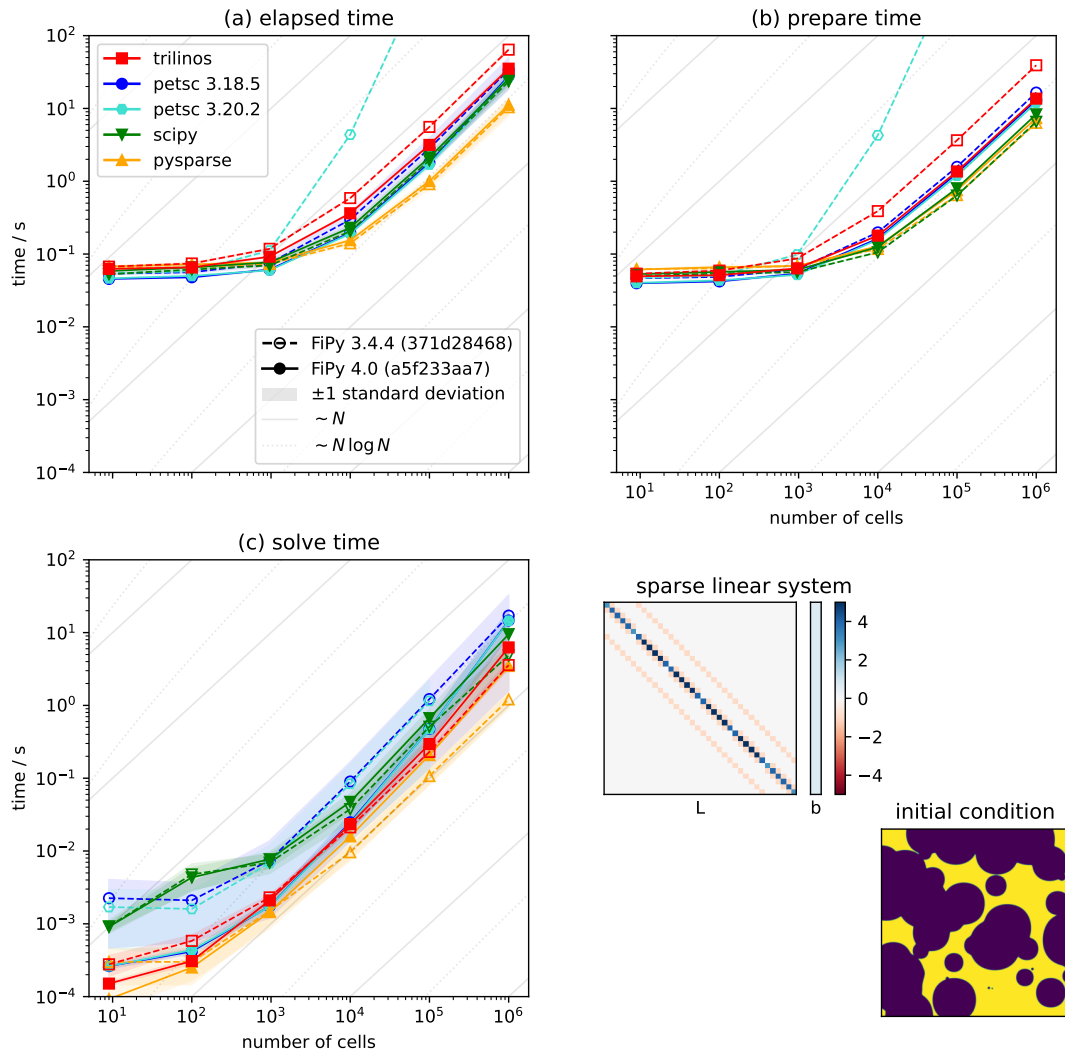


Fig. 1: Comparison of serial performance for different solver suites, solvers and preconditioners, and different versions of *FiPy*⁴. (a) Total elapsed time, (b) time to prepare the matrix, and (c) time to solve the matrix as functions of mesh size. ^{Page 33, 5}

- Matrix preparation time is terrible when older *FiPy* is combined with newer *PETSc*. *PETSc* 3.19 introduced changes to “provide reasonable performance when no preallocation information is provided”. Our experience is opposite that; *FiPy* did not supply preallocation information prior to version 4.0, but matrix preparation performance was fine with older *PETSc* releases. *FiPy* 4.0 does supply preallocation information and matrix preparation time is comparable for all tested versions of *PETSc*.
- There is considerable dispersion about the mean solve time for different solvers and preconditioners. On the other hand, the time to prepare the matrix is insensitive to the choice of solver and preconditioner and shows a high degree of consistency from run to run.

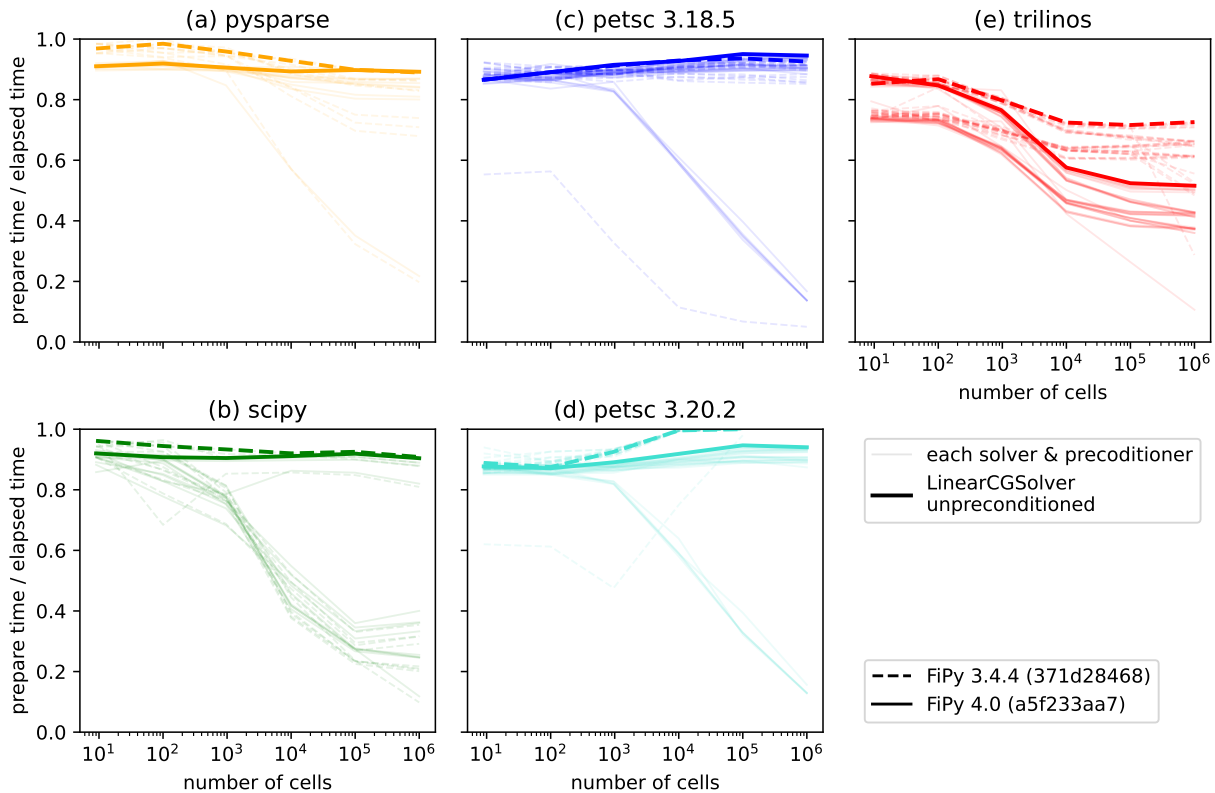


Fig. 2: Ratio of time to prepare the matrix to the combined time to prepare and solve the matrix for different solver suites, solvers and preconditioners, and different versions of *FiPy*^{Page 33, 4Page 33, 5}. The thick lines highlight LinearCGSolver with no preconditioner, one of the better-performing combinations available in all suites.

In principle, we’d like to spend as little time preparing the matrix, relative to solving it, as possible. This metric can be deceptive. If we compare the case of unpreconditioned LinearCGSolver, one of the fastest combinations for all suites *for this problem*, we see that *Trilinos* has the lowest ratio of prepare to elapsed time. Examination of elapsed time, the quantity we really care about, shows that *Trilinos* takes three times as long to both prepare and solve as *Pyparse* or *SciPy* and twice as long as *PETSc*.

For your own work, focus on identifying the solver and preconditioner with the lowest overall time to build and solve; this will counterintuitively have the highest ratio of prepare-to-elapsed time. Prepare time to elapsed time is a more useful metric for the *FiPy* developers; just as *FiPy* 4.0 brought considerable reductions in matrix build time, we will continue to seek opportunities to optimize.

4.6.2 Parallel Performance

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.

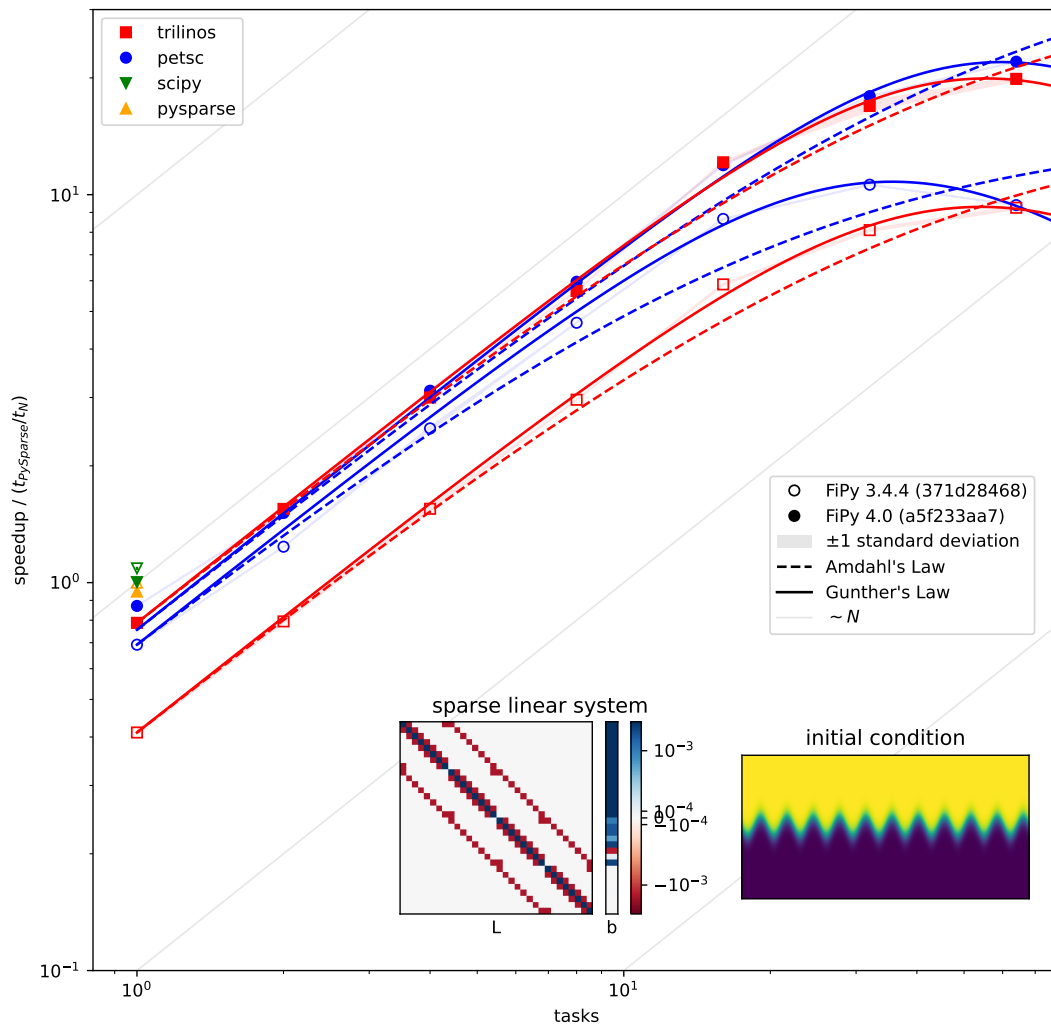


Fig. 3: Parallel scaling behavior of different solver packages and different versions of *FiPy*^{Page 33, 46}.

A few things can be observed in this plot:

- *PETSc*, *Pysparse*, *Trilinos*, and *SciPy* have comparable serial performance, with *SciPy* edging out the other three for this particular problem.

⁶ Calculations are of a [Method of Manufactured Solutions Allen-Cahn](#) problem. Solutions are on a 2048×1024 Grid2D and the `LinearCGSolver` with no preconditioner is used for all solver suites. Solution tolerance is $1e-10$ using the "RHS" *convergence criterion*. Five replicates of each simulation were run on an AMD Epyc 7702 CPU with 64 cores featuring two-thread Simultaneous Multi-Threading (SMT) and 512 GB of memory. `OMP_NUM_THREADS` was set to 1.

- *FiPy* 4.0 is roughly the same speed in serial, but more than twice as fast in parallel compared to *FiPy* 3.4.4 when using the *PETSc* solvers. *FiPy* 4.0 is roughly twice as fast using the *Trilinos* solvers, whether in serial or parallel.
- *FiPy* 4.0 exhibits better parallel scaling than *FiPy* 3.4.4. *Amdahl's Law*, $\text{speedup} = p / (1 + \sigma(p - 1))$, does not fit the performance data nearly as well as *Gunther's Law*, $\text{speedup} = p / (1 + \sigma(p - 1) + \kappa p(p - 1))$, where p is the number of parallel tasks, σ is the fraction limited by serial processes, and κ is “coherency” (which is somewhat nebulous).

Table 2: Parallel scaling fitting parameters (smaller numbers are better)

		Amdahl serial / %	Gunther serial / %	coherency
FiPy 3.4.4	petsc	4.7(3)	0.91(9)	0.00078(2)
	trilinos	2.6(1)	0.8(1)	0.00034(2)
FiPy 4.0	petsc	1.70(8)	0.13(7)	0.00028(1)
	trilinos	2.2(1)	0.4(1)	0.00032(3)

At least one source of less-than-optimal 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.

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.

4.7 Convergence

Different solver suites take different approaches to testing convergence. We endeavor to harmonize this behavior by allowing the strings in the “criterion” column to be passed as an argument when instantiating a *Solver*. Convergence is detected if `residual < tolerance * scale`.

Table 3: Residual Criteria

criterion	residual	scale	<i>PETSc</i> ^{Page 38, 7}	<i>pyamgx</i> ⁸	<i>SciPy</i> ⁹	<i>Trilinos</i> ¹⁰
unscaled	$\ \mathbf{L}\vec{x} - \vec{b}\ _2$	1	¹¹	ABSOLUTE	¹¹	AZ_noscaled
RHS	$\ \mathbf{L}\vec{x} - \vec{b}\ _2$	$\ \vec{b}\ _2$	KSP_NORM_UNPRECOND	¹¹	default	AZ_rhs
matrix	$\ \mathbf{L}\vec{x} - \vec{b}\ _2$	$\ \mathbf{L}\ _\infty$	¹¹	¹¹	¹¹	AZ_Anorm
initial	$\ \mathbf{L}\vec{x} - \vec{b}\ _2$	$\ \mathbf{L}\vec{x} - \vec{b}\ _2^{(0)}$	¹¹	RELATIVE_INI_CORE	¹¹	AZ_r0
solution	$\ \mathbf{L}\vec{x} - \vec{b}\ _\infty$	$\ \mathbf{L}\ _\infty * \ \vec{x}\ _1 + \ \vec{b}\ _\infty$				AZ_sol
preconditioned	$\ \mathbf{P}^{-1}(\mathbf{L}\vec{x} - \vec{b})\ _2$	$\ \vec{b}\ _2$	KSP_NORM_PRECOND			
natural	$\sqrt{(\mathbf{L}\vec{x} - \vec{b})\mathbf{P}^{-1}(\mathbf{L}\vec{x} - \vec{b})}$	$\ \vec{b}\ _2$	KSP_NORM_NATURAL			
legacy			KSP_NORM_DEFAULT (RHS or preconditioned)	initial	RHS	initial
default			RHS	RHS	RHS	RHS

⁷ <https://petsc.org/release/docs/manual/ksp/#sec-convergentests>

⁸ *AMGX REFERENCE MANUAL*: 2.3 General Settings: convergence, October 2017, API Version 2, https://github.com/NVIDIA/AMGX/blob/main/doc/AMGX_Reference.pdf

⁹ https://github.com/scipy/scipy/blob/2d1d5b042a09e131ffe191726aa6829b33590970/scipy/sparse/linalg/_solve/iterative.py#L30

¹⁰ *AztecOO Users Guide*: 3.1 Aztec Options: options[AZ_conv], SAND REPORT SAND2004-3796, Updated August 2007, For AztecOO Version 3.6 in Trilinos Release 8.0, <https://trilinos.github.io/pdfs/AztecOOUserGuide.pdf>

¹¹ Implemented by *FiPy* using intrinsic solver capabilities.

Note: *PyAMG* is a set of preconditioners applied on top of *SciPy*, so is not explicitly included in these tables.

4.7.1 Tolerance

The default tolerance is 10^{-5} (the legacy tolerance, prior to *FiPy* 4.0, was 10^{-10}).

- *SciPy* and *Trilinos* can fail with *tolerance=1e-10*. (*SCIPY_MAXIT* or *AZ_loss*, respectively) because they are unable to make the residual any smaller than $\mathcal{O}(10^{-9})$.
- *tolerance=1e-5* is the default for *PETSc* and *SciPy*.
- *pyamgx* defaults to 10^{-12}
- *Trilinos* does not specify, but has examples that illustrate 10^{-8} .

4.7.2 default

The setting `criterion="default"` applies the same scaling (RHS) to all solvers. This behavior is new in *FiPy* 4.0; prior to that, the default behavior was the same as `criterion="legacy"`.

4.7.3 legacy

The setting `criterion="legacy"` restores the behavior of *FiPy* prior to version 4.0 and is equivalent to what the particular suite and solver does if not specifically configured. The `legacy` row of the table is a best effort at documenting what will happen.

Note:

- All LU solvers use "initial" scaling.
 - *PETSc* accepts `KSP_NORM_DEFAULT` in order to “use the default for the current `KSPTYPE`”. Discerning the actual behavior would require burning the code in a bowl of chicken entrails. (It is reasonable to assume `KSP_NORM_PRECONDITIONED` for left-preconditioned solvers and `KSP_NORM_UNPRECONDITIONED` otherwise.)
 - Even the *PETSc* documentation says that `KSP_NORM_NATURAL` is “weird”).
-

4.7.4 absolute_tolerance

PETSc and *SciPy* Krylov solvers accept an additional `absolute_tolerance` parameter, such that convergence is detected if `residual < max(tolerance * scale, absolute_tolerance)`.

4.7.5 divergence_tolerance

PETSc Krylov solvers accept a third `divergence_tolerance` parameter, such that a divergence is detected if $\text{residual} > \text{divergence\_tolerance} * \text{scale}$. Because of the way the convergence test is coded, if the initial residual is much larger than the norm of the right-hand-side vector, *PETSc* will abort with `KSP_DIVERGED_DTOL` without ever trying to solve. If this occurs, either `divergence_tolerance` should be increased or another convergence criterion should be used.

Note: See `examples.diffusion.mesh1D`, `examples.diffusion.steadyState.mesh1D.inputPeriodic`, `examples.elphf.diffusion.mesh1D`, `examples.elphf.phaseDiffusion`, `examples.phase.binary`, `examples.phase.quaternary`, and `examples.reactiveWetting.liquidVapor1D` for several examples where `criterion="initial"` is used to address this situation.

Note: `divergence_tolerance` never caused a problem in previous versions of *FiPy* because the default behavior of *PETSc* is to zero out the initial guess before trying to solve and then never do a test against `divergence_tolerance`. This resulted in behavior (number of iterations and ultimate residual) that was very different from the other solver suites and so *FiPy* now directs *PETSc* to use the initial guess.

4.7.6 Reporting

Different solver suites also report different levels of detail about why they succeed or fail. This information is captured as a *Convergence* or *Divergence* property of the *Solver* after calling `solve()` or `sweep()`.

Table 4: Convergence Status Codes

	<i>PETSc</i>	<i>pyamgx</i>	<i>SciPy</i>	<i>Trilinos</i>
<i>Convergence</i>	Convergence criteria met.	AMGX_SOLVE_SUCCESS	0	AZ_normal
<i>IterationConvergence</i>	Requested iterations complete (and no residual calculated).	KSP_CONVERGED_ITS		
<i>AbsoluteToleranceConvergence</i>	Converged, residual is as small as seems reasonable on this machine.	KSP_CONVERGED_ATOL		
<i>RHSZeroConvergence</i>	Converged, $\mathbf{b} = 0$ so the exact solution is $\mathbf{x} = 0$.			
<i>RelativeToleranceConvergence</i>	Converged, relative error appears to be less than tolerance.	KSP_CONVERGED_RTOL		
<i>HappyBreakdownConvergence</i>	“Exact” solution found and more iterations will just make things worse.	KSP_CONVERGED_HAPPY_BREAKD		

continues on next page

Table 4 – continued from previous page

	<i>PETSc</i>	<i>pyamgx</i>	<i>SciPy</i>	<i>Trili- nos</i>
<i>LossOfAccuracyConvergence</i>	The iterative solver has terminated due to a lack of accuracy in the recursive residual (caused by rounding errors).			AZ_loss
<i>IteratingConvergence</i>	Solve still in progress.	KSP_CONVERGED_ITERATING		

Table 5: Divergence Status Codes

		<i>PETSc</i>	<i>pyamgx</i>	<i>SciPy</i>	<i>Trilinos</i>
<i>BreakdownDivergence</i>	Illegal input or the iterative solver has broken down.	KSP_DIVERGED_BREAKDOWN	AMGX_SOLVE_FAILED	<0	AZ_breakdown
<i>IterationDivergence</i>	Maximum number of iterations was reached.	KSP_DIVERGED_ITS	AMGX_SOLVE_DIVERGEI	>0	AZ_maxits
<i>PreconditioningDivergence</i>	The system involving the preconditioner was ill-conditioned.	KSP_DIVERGED_PC_FAILED			
<i>IllConditionedPreconditionerDivergence</i>	An inner product of the form $\mathbf{x}^T \mathbf{P}^{-1} \mathbf{x}$ not positive, so the preconditioning matrix $\mathbf{P}$ does not appear to be positive definite.	KSP_DIVERGED_INDEFINITE_PC			
<i>IllConditionedDivergence</i>	The matrix $\mathbf{L}$ appears to be ill-conditioned.	KSP_DIVERGED_INDEFINITE_MA			AZ_ill_cond
<i>StagnatedDivergence</i>	The method stagnated.				

continues on next page

Table 5 – continued from previous page

		<i>PETSc</i>	<i>pyamgx</i>	<i>SciPy</i>	<i>Trili- nos</i>
<i>OutOfRangeDivergence</i>	A scalar quantity became too small or too large to continue computing.	<i>KSP_DIVERGED_NANORINF</i>			
<i>NullDivergence</i>	Breakdown when solving the Hessenberg system within GMRES.	<i>KSP_DIVERGED_NULL</i>			
<i>ToleranceDivergence</i>	The residual norm increased by a factor of divtol.	<i>KSP_DIVERGED_DTOL</i>			

Chapter 5

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.

5.1 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.

5.2 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.

Frequently Asked Questions

6.1 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*.

6.1.1 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.

6.1.2 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 (8.3), as

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

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

6.1.3 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*.

6.1.4 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!*.

6.2 How can I see what I'm doing?

6.2.1 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](#).

6.2.2 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>.

6.2.3 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*.

6.2.4 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](#), [examples.phase.binary](#), [examples.phase.binaryCoupled](#), 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.

6.3 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 = LinearCGSolver(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_1to1_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 = LinearCGSolver(iterations=1234, tolerance=5e-6)
:
:
>>> while elapsedTime < totalElapsedTime:
...     myVar.updateOld()
...     while residual > desiredResidual:
...         residual = eq.sweep(var=myVar, dt=timeStep, ...)
...         elapsedTime += timeStep
```

6.4 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.

6.5 How do I represent boundary conditions?

See the *Boundary Conditions* section for more details.

6.6 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 in one or more locations.

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/mailinglists/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>.

6.7 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.

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

See *Solving in Parallel*.

6.9 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*.

6.10 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>>.

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 [13].

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 (s)	<i>--inline</i>	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.

7.1 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

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 [14], Patankar [15] or Versteeg and Malalasekera [16].

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 [17]. Analogously, the discretization of equations with the FVM reduces to the FDM on Cartesian grids.

8.1 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

$$\frac{\partial \phi}{\partial t} = H(\phi, \lambda_i) \quad (8.1)$$

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,

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} + \underbrace{\nabla \cdot (\vec{u}\phi)}_{\text{convection}} = \underbrace{[\nabla \cdot (\Gamma_i \nabla)]^n \phi}_{\text{diffusion}} + \underbrace{S_\phi}_{\text{source}} \quad (8.2)$$

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.*, $\nabla \cdot (\Gamma_1 \nabla [\nabla \cdot \Gamma_2 \nabla \phi])$] [18] [19] [20]] when $n = 2$, or a phase field crystal term [*i.e.*, $\nabla \cdot (\Gamma_1 \nabla [\nabla \cdot \Gamma_2 \nabla \{\nabla \cdot \Gamma_3 \nabla \phi\}])$] [21]] 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.

8.2 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.

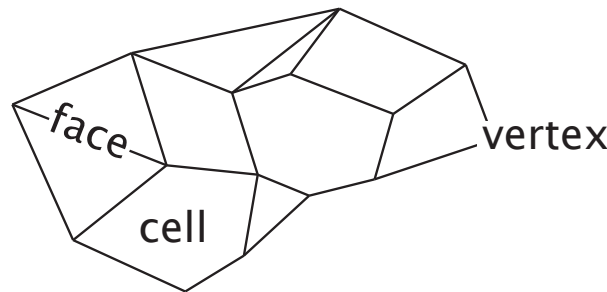


Fig. 1: Mesh

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.

8.2.1 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.

Boundary Conditions

The natural boundary condition for CC-FVM is no-flux. For (8.2), the boundary condition is

$$\hat{n} \cdot [\vec{u}\phi - (\Gamma_i \nabla)^n] = 0$$

8.2.2 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.

8.3 Discretization

The first step in the discretization of Equation (8.2) 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 (8.2) will be examined separately.

8.3.1 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

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{(\rho_P \phi_P - \rho_P^{\text{old}} \phi_P^{\text{old}}) V_P}{\Delta t} \quad (8.3)$$

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)
```

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

The discretization for the convection term is given by

$$\begin{aligned} \int_V \nabla \cdot (\vec{u}\phi) dV &= \int_S (\vec{n} \cdot \vec{u}) \phi dS \\ &\simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f \end{aligned} \quad (8.4)$$

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)
```

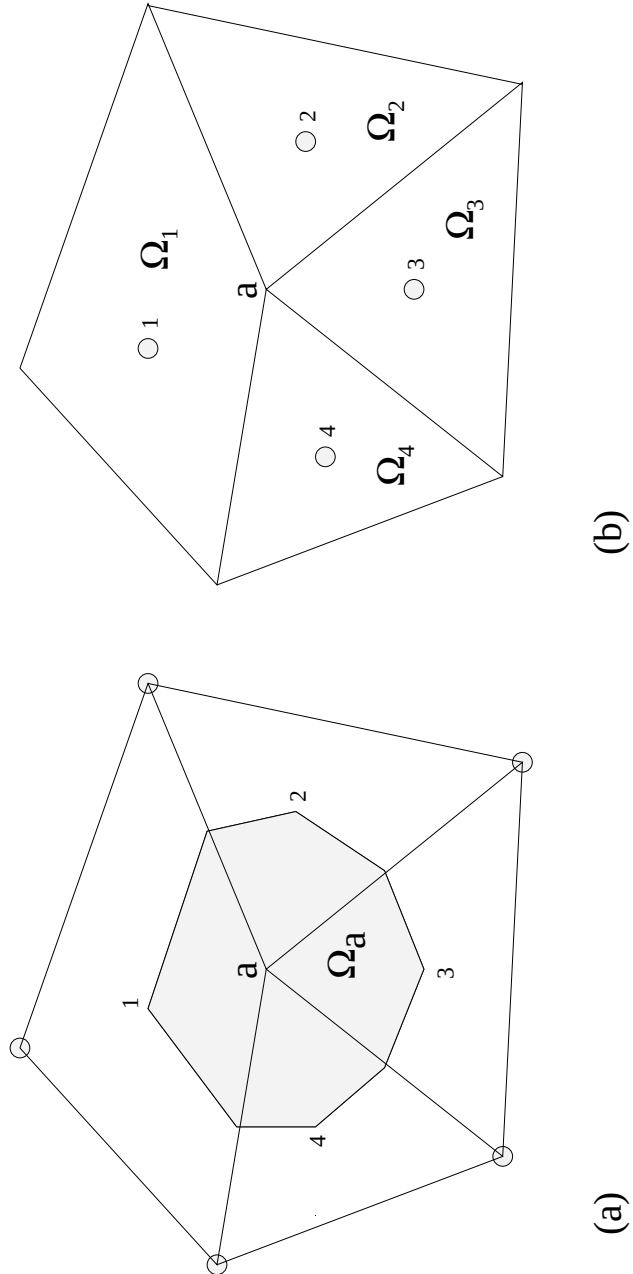


Fig. 2: CV structure for an unstructured mesh
(a) Ω_a represents a vertex-based CV and (b) Ω_1 , Ω_2 , Ω_3 and Ω_4 represent cell centered CVs.

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.

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

The discretization for the diffusion term is given by

$$\begin{aligned} \int_V \nabla \cdot (\Gamma \nabla \{\dots\}) dV &= \int_S \Gamma (\vec{n} \cdot \nabla \{\dots\}) dS \\ &\simeq \sum_f \Gamma_f (\vec{n} \cdot \nabla \{\dots\})_f A_f \end{aligned} \quad (8.5)$$

$\{\dots\}$ 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* [14].

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.

Note: While this multiple-coefficient form is still supported, *Coupled and Vector Equations* are the recommended approach for higher order expressions.

8.3.4 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,

$$\int_V S_\phi dV \simeq S_\phi V_P. \quad (8.6)$$

Including any negative dependence of S_ϕ on ϕ increases solution stability. The dependence can only be included in a linear manner so Equation (8.6) 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))
```

8.4 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 encapsulated in the suites of solvers described in *Solvers* and include most common solvers such as the conjugate gradient method and LU decomposition.

Combining Equations (8.3), (8.4), (8.5) and (8.6), the complete discretization for equation (8.2) 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 (8.7) 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

$$a_P \phi_P = \sum_f a_A \phi_A + b_P, \quad (8.7)$$

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

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

8.5 Numerical Schemes

The coefficients of equation (8.7) 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

$$\frac{1}{1 - \alpha_f} > P_f > -\frac{1}{\alpha_f} \quad (8.8)$$

must be satisfied. The parameter α_f is defined by the chosen scheme, depending on Equation (8.8). The various differencing schemes are:

the central differencing scheme,

where

$$\alpha_f = \frac{1}{2} \quad (8.9)$$

so that $|P_f| < 2$ satisfies Equation (8.8). Thus, the central differencing scheme is only numerically stable for a low values of P_f .

the upwind scheme,

where

$$\alpha_f = \begin{cases} 1 & \text{if } P_f > 0, \\ 0 & \text{if } P_f < 0. \end{cases} \quad (8.10)$$

Equation (8.10) satisfies the inequality in Equation (8.8) 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

$$\alpha_f = \frac{(P_f - 1) \exp(P_f) + 1}{P_f (\exp(P_f) - 1)}. \quad (8.11)$$

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

$$\alpha_f = \begin{cases} \frac{P_f-1}{P_f} & \text{if } P_f > 2, \\ \frac{1}{2} & \text{if } |P_f| < 2, \\ -\frac{1}{P_f} & \text{if } P_f < -2. \end{cases} \quad (8.12)$$

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

$$\alpha_f = \begin{cases} \frac{P_f-1}{P_f} & \text{if } P_f > 10, \\ \frac{(P_f-1)+(1-P_f/10)^5}{P_f} & \text{if } 0 < P_f < 10, \\ \frac{(1-P_f/10)^5-1}{P_f} & \text{if } -10 < P_f < 0, \\ -\frac{1}{P_f} & \text{if } P_f < -10. \end{cases} \quad (8.13)$$

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.

9.1 Design

9.1.1 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.

9.1.2 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.

9.1.3 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.

9.1.4 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.

9.1.5 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*.

9.1.6 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*, *PETSc*, *Trilinos*, ...).
- easily integrated with low-level languages such as C (*weave*, *blitz*, *PyRex*).

9.2 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 (8.2) 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 (8.2), 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 (8.2).

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

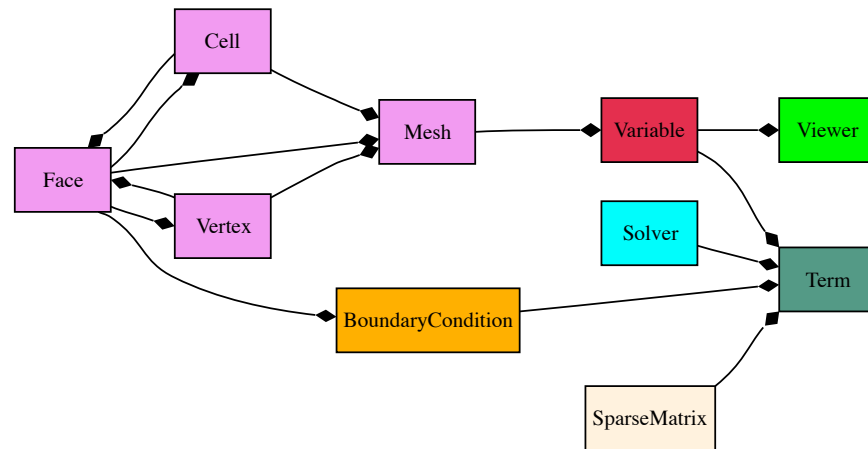


Fig. 1: Primary object relationships in *FiPy*.

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 10

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](#).

Chapter 11

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 12

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](#).

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 14

Change Log

14.1 Version 4.0.1+0.gcde50d518.dirty

This release

- Drops support for *Pysparse*, which required *Python 2.7*.
- Drops support for *Python 2.7*, which is years beyond end of life.
- Updates the packaging and installation of *FiPy*.

Note: Also see the coincident changes in *Version 3.99 - 2025-09-16*.

14.1.1 Pulls

- Migrate to `pyproject.toml` (#1154)
- Fix links (#1152)
- Remove deprecated `faceGradAverage` (#1149)
- Drop Pysparse (#1145)

14.1.2 Fixes

- #1137: `setuptools.sandbox` is deprecated
- #1110: Google search returns FiPy 3.4.4; not 3.4.5
- #1108: Legacy editable install is deprecated
- #1107: The test command is disabled and references to it are deprecated
- #1055: Update Scipy version in Nix build to remove test failure
- #1016: CONTACT link broken on `pages.nist.gov`
- #986: Redirect `ctcms.nist.gov/fipy` to `pages.nist.gov/fipy`
- #969: Get rid of bone-headed redirects for non-NIST websites

- #891: import warnings: syntax & deprecation
- #872: `fipy.test()` has `setuptools` failure in a directory with anything in it
- #803: Drop Python 2.7
- #677: local variable `exitErr` referenced before assignment in `fipy.tests.test.test.run_tests()`
- #189: switch from `setuptools` to `distribute`?

14.2 Version 3.99 - 2025-09-16

This release harmonizes the sparse *Solvers*:

- Uses consistent interpretation of *Convergence* criteria.
- Improves efficiency at building matrices.
- Supports more preconditioners.
- Logs solver status and performance.
- Greatly expands documentation of the configuration and performance of the supported *Solvers*.
- Changes default tolerance to 10^{-5} . This is adequate for most problems.
- Removes “*pyamg*” from *FIPY_SOLVERS* and `--pyamg` from the *Command-line Flags*. *PyAMG* is implemented as a preconditioner for *SciPy*, not as a solver suite in its own right.

14.2.1 Pulls

- Reorganize Table of Contents (#1143)
- Link to proper CI for documentation (#1134)
- Improve URL check implementation (#1130) Thanks to @arpitjain099.
- Patch `sphinx_readme` (#1125)
- Resolve issues introduced by PETSc 3.23.0 (#1116)
- Generate root `.rst` files with `sphinx_readme` (#1105) Thanks to @obscurerichard.
- Update byte-code for Python 3.13 (#1095)
- Handle exporting anonymous conda environment (#1082)
- Fix representation of dotted variables (and other method calls) (#1074)
- Prevent `setuptools 72.x` (#1067)
- `build(nix)`: update to using latest Nix master version (#1061)
- Harmonize solvers (#1030)
- Deprecate *fipy.steppers* in favor of *steppyngstounes* (#777)

14.2.2 Fixes

- [#1115](#): “Create Anaconda environment from environments” times out on nightly CI
- [#1114](#): VTK tests broke with *vtk-base=9.4.2*
- [#1087](#): Recent version of matplotlib won’t plot with colorbar. Thanks to [@cgadal](#).
- [#1069](#): Cannot dot *CellVariables*
- [#1063](#): Nightly PETSc build unable to load *libscotcherr-6.so*
- [#1009](#): Matplotlib 3.8 fails to plot irregular meshes
- [#1002](#): Setting *nx*, *ny*, *nz* to non-positive integer should generate an error. Thanks to [@sridhar-mani](#).

14.3 Version 3.4.5 - 2024-06-25

This maintenance release:

- Addresses compatibility with recent releases of Python 3.12, NumPy 2.0, SciPy 1.14, and PETSc 3.20.
- Adds *conda-lock* environment lock files with specified compatible versions of FiPy prerequisites.
- Fixes numerous documentation errors.

Attention: SciPy 1.13.0 generates one test suite error for `fipy.matrices.scipyMatrix._ScipyMatrix.CSR`. Either ignore the test failure or upgrade to SciPy $\geq 1.13.1$

Attention: PETSc 3.21 crashes our test suite when running in parallel ([#1054](#)). PETSc ≤ 3.20 is recommended, although *petsc 3.20.2_*_102* is broken on macOS.

14.3.1 Pulls

- Introduce Timer context manager ([#995](#))
- switch nix recipe to flake ([#992](#))
- Tweak documentation ([#991](#))
- Log much more information about FiPy environment ([#990](#))
- Fix inclusion of *environments/README.rst* ([#988](#))
- Environment pinning ([#985](#))

14.3.2 Fixes

- [#1049](#): Numpy 2.0.0 breaks things
- [#1010](#): *examples.diffusion.mesh1D* No-flux - steady-state doesn't always give zero
- [#1000](#): *examples.diffusion.mesh1D* constrains a gradient but calls it a flux
- [#997](#): *future.standard_library* breaking python 3.12 compatibility
- [#967](#): Sign error in Robin condition
- [#963](#): PETSc 3.20.0 broke the world
- [#961](#): Representation of index variables is broken
- [#952](#): Uncaught Exception from the no-flux steady-state diffusion example
- [#944](#): Having problem with Viewer
- [#865](#): Sphinx search is broken on website
- [#729](#): Python 3.8
- [#673](#): Deprecations don't properly format properties
- [#512](#): Default coefficient of *ImplicitSourceTerm* is 0

14.4 Version 3.4.4 - 2023-06-27

This maintenance release adds *Logging* and resolves compatibility issues with recent builds of *PETSc* and *NumPy*.

14.4.1 Pulls

- Fix numpy 1.25 issues ([#930](#))
- Get CI working again ([#925](#))
- Discourage StackOverflow ([#876](#))
- Add Logging ([#875](#))
- Add tests for the Nix build ([#791](#))

14.4.2 Fixes

- [#896](#): Poor garbage collection with petsc4py 3.18.3 (was “Memory leak in *term.justErrorVector()*”, but this isn't strictly a leak)

14.5 Version 3.4.3 - 2022-06-15

This maintenance release adds a new example contributed by [@Jon83Carvalho](#), clarifies many points in the documentation, migrates all *Continuous Integration* to *Azure*, updates to using *wheels* for distribution, and substantially refactors matrices to work more consistently across solvers.

14.5.1 Pulls

- Update CI documentation to refer only to Azure (#863)
- Refine azure runs (#851)
- Debug CIs (#848)
- Collect contact information on single page (#847)
- Set up CI with Azure Pipelines (#822)
- Replace deprecated numpy types (#798)
- Move trilinos tests to Py3k (#797)
- Fix Python 2.7 conda environment (#795)
- fix: stop divide by zero warning in LU solvers (#790)
- Introduce *SharedTemporaryFile* (bis) (#769)
- Raise *ImportError* before trying to unpack solvers (#768)
- Disable TVTK tests if its prerequisites aren't met (#764)
- Tabulate versions of FiPy dependencies when tests are run (#763)
- Debug CI failures (#749)
- Stokes Cavity - non-Newtonian (#748) Thanks to [@Jon83Carvalho](#).
- Refactor matrices (#721)

14.5.2 Fixes

- #862: Could not load the Qt platform plugin “xcb”
- #858: CI issues
- #856: *FaceVariable* does not accumulate properly in parallel
- #850: Switch to wheels
- #849: *linux-py27-pysparse* fails
- #841: *Matplotlib2DViewer* should accept color map as string
- #836: Document that coupled and high-order diffusion terms are incompatible
- #833: *fipy.tools.dump* undocumented that it always gzips
- #828: *colorbar=True* no longer works Stokes flow example
- #826: Gmsh load issue
- #818: Document that *GridND* meshes are always Cartesian
- #811: In python 3.9 `__repr__` throws an exception with abs

- #801: CircleCI test-36-trilinos-serial extremely slow
- #800: CircleCI conda2_env is really slow and ends up installing FiPy 3.3
- #796: *examples.phase.polyxtal* freezes on CircleCI with Py3k and scipy solvers
- #792: *circleQuad* example fails with Gmsh > 4.4
- #781: *MatplotlibViewer.axes* property is not documented
- #778: Binder failed build
- #762: Equations on Website don't show right
- #742: No documentation for *Variable.mag*
- #735: *pip install fipy* fails
- #734: Document the residual
- #688: try-except not needed for circle Viewer
- #676: Default no-flux condition is not explicitly stated
- #609: Parallelizing of Gmsh meshes not clearly documented
- #400: Fix *FaceVariable.globalValue* method

14.6 Version 3.4.2.1 - 2020-08-01

This release fixes assorted viewer issues, fixes a problem with convection boundary conditions, and introduces spherical meshes.

Attention: There are [known failures](#) with the VTK viewers (bitrot has started to set in since the [demise of Python 2.7](#)). There's also a new parallel failure in *NonUniformGrid1D* that we need to figure out.

14.6.1 Pulls

- Move mailing list (#747)
- *Spherical1D* (*Uniform* and *NonUniform*) meshes (#732) Thanks to [@klkuhlm](#).
- fix Neumann BCs using constraints with convection terms (#719) Thanks to [@atismer](#).
- Add vertex index inversions (#716)

14.6.2 Fixes

- #726: *MayaviClient* not compatible with Python 3
- #663: *datamin/datamax* argument ignored by viewer
- #662: Issues Scaling *Colorbar* with *Datamin* and *Datamax* Args

14.7 Version 3.4.1 - 2020-02-14

This release is primarily for compatibility with `numpy` 1.18.

14.7.1 Pulls

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

14.7.2 Fixes

- #703: FORTRAN array ordering is deprecated

14.8 Version 3.4 - 2020-02-06

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

14.8.1 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)

14.8.2 Fixes

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

14.9 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.

14.9.1 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.

14.9.2 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

14.10 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*).

14.10.1 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)

14.10.2 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”

14.11 Version 3.1.3 - 2017-01-17

14.11.1 Fixes

- #502: *gmane* is defunct

14.12 Version 3.1.2 - 2016-12-24

14.12.1 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)

14.12.2 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

14.13 Version 3.1.1 - 2015-12-17

14.13.1 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*

14.14 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.

14.14.1 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*?

14.15 Version 3.0.1 - 2012-10-03

14.15.1 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

14.16 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.

14.16.1 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

14.17 Version 2.1.3 - 2012-01-17

14.17.1 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

14.18 Version 2.1.2 - 2011-04-20

The significant changes since version 2.1.1 are:

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

14.18.1 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*

14.19 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.

14.19.1 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

14.20 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 *getCells* 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.

14.20.1 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

14.21 Version 2.0.3 - 2010-03-17

14.21.1 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

14.22 Version 2.0.2 - 2009-06-11

14.22.1 Fixes

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

14.23 Version 2.0.1 - 2009-04-23

14.23.1 Fixes

- #154: Update manuals

14.24 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

14.25 Version 1.2.3 - 2009-01-0

14.25.1 Fixes

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

14.26 Version 1.2.2 - 2008-12-30

14.26.1 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

14.27 Version 1.2.1 - 2008-02-08

14.27.1 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

14.28 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.

14.28.1 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

14.29 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](#).

14.30 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`.

14.30.1 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

14.31 Version 0.1.1

14.32 Version 0.1 - 2004-11-05

Original release

Chapter 15

Git practices

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

15.1 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"
```

15.1.1 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:

```
$ fipy_test
```

15.1.2 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.

15.1.3 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 16

Continuous Integration



We use the *Azure* and *GitHub Actions* cloud services for *Continuous Integration* (CI). These CIs are configured in *FiPySource/.azure/pipelines.yml*, *FiPySource/.github/workflows/Docs4NIST.yml*, and *FiPySource/.github/workflows/nix.yml*.

Note: In order to focus on breakages introduced by changes to *FiPy*, a *pull request* is normally built with one of the *Conda Lockfiles*, whereas the nightly builds use an *environment.yml* in order to catch breakages introduced by *FiPy*'s prerequisites.

A *pull request* may be tested with the latest prerequisites by setting the `CONDA_ENVIRONMENT_NOT_LOCK` environment variable in *Azure* at queue time.

Chapter 17

Conda Lockfiles

The `conda-lock` lockfiles in `environments/locks/` can be updated with:

```
$ for solver in petsc scipy trilinos
do
  conda-lock lock \
    --file environments/${solver}-environment.yml \
    --lockfile environments/locks/conda-${solver}-lock.yml
  conda-lock render \
    --filename-template environments/locks/conda-${solver}-${platform}.lock \
    environments/locks/conda-${solver}-lock.yml
done
```

Attention: Do not merge new lockfiles to master without validating that everything still works.

Attention: As of 2025-04-30, locking `environment/locks/trilinos-environment.yml` is extremely slow.

Attention: Due to an issue with URL encoding, it may be necessary to replace `%21` with `!` in the `environments/locks/conda-${solver}-lock.yml` files before calling **conda-lock render**.

See [conda/conda-lock#764](#), [mamba-org/mamba#3737](#), [conda/conda#14481](#).

Chapter 18

README-like documents

The contents of

- `CHANGELOG.rst`
- `INSTALLATION.rst`
- `README.rst`

are managed by the `sphinx-reading` extension. In order to make changes

- Make edits to the corresponding files in `docs/source/`.
- Run:

```
$ make -C docs html
```

to re-render the 3 affected files.

- Add and commit the resulting changes.

Chapter 19

Making a Release

Attention: Administrators Only!

19.1 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:

```
$ fipy_changelog --after=<x.y>
```

or:

```
$ fipy_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`!

19.2 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 from [Azure Artifacts](#) and place in `FiPySource/dist/`:

- `dist-Linux/fipy-x.y-none-any.whl`
- `dist-Linux/fipy-x.y.tar.gz`

Note: Artifacts are also created and tested in `dist-Windows_NT`, but they should be the same.

19.3 Upload

Attach `dist/fipy-x.y-none-any.whl` to a [GitHub release](#) associated with tag `x.y`.

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

```
$ twine upload dist/fipy-${FIPY_VERSION}*
```

19.4 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

19.5 Announce

Make an announcement to fipy@list.nist.gov

Chapter 20

Glossary

AMG

Algebraic multigrid method for solving sparse matrices. AMG solves linear systems with a general non-symmetric coefficient matrix. See [https://en.wikipedia.org/wiki/Multigrid_method#Algebraic_multigrid_\(AMG\)](https://en.wikipedia.org/wiki/Multigrid_method#Algebraic_multigrid_(AMG)).

AppVeyor

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

Azure

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

BiCG

Biconjugate gradient method for solving sparse matrices. BiCG solves linear systems with a general non-symmetric coefficient matrix. See https://en.wikipedia.org/wiki/Biconjugate_gradient_method.

BiCGSTAB

Biconjugate gradient (stabilized) method for solving sparse matrices. BiCGSTAB solves linear systems with a general non-symmetric coefficient matrix. See https://en.wikipedia.org/wiki/Biconjugate_gradient_stabilized_method.

CG

Conjugate Gradient method for solving sparse matrices. CG solves linear systems with a symmetric positive-definite coefficient matrix. See https://en.wikipedia.org/wiki/Conjugate_gradient_method.

CGS

Conjugate Gradient Squared method for solving sparse matrices, a variant of *BiCG*. CGS solves linear systems with a general non-symmetric coefficient matrix. See https://en.wikipedia.org/wiki/Conjugate_gradient_squared_method.

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>.

FGMRES

Flexible Inner-Outer Preconditioned *GMRES* algorithm for solving sparse matrices. FGMRES solves systems with a general non-symmetric coefficient matrix. See <https://doi.org/10.1137/0914028>.

GitHub Actions

A cloud-based *Continuous Integration* tool. See <https://github.com/features/actions>.

GMRES

Generalized Minimal RESidual method for solving sparse matrices. GMRES solves systems with a general non-symmetric coefficient matrix. See https://en.wikipedia.org/wiki/Generalized_minimal_residual_method.

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/>.

JOR

Jacobi over-relaxation method for solving sparse matrices. JOR solves systems with a general non-symmetric coefficient matrix.

JSON

JavaScript Object Notation. A text format suitable for storing structured information such as `dict` or `list`. See <https://www.json.org/>.

linux

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

LU

Lower-Upper decomposition method for solving sparse matrices. LU solves systems with a general non-symmetric coefficient matrix using partial pivoting. See https://en.wikipedia.org/wiki/LU_decomposition.

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 this package, 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>.

PCG

Preconditioned conjugate gradient method for solving sparse matrices. PCG solves systems with a symmetric positive definite coefficient matrix. See https://en.wikipedia.org/wiki/Conjugate_gradient_method.

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>.

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

Pysparse is an archaic sparse matrix library for *Python*. As it requires *Python* 2.7, Pysparse is no longer supported. If absolutely required, Pysparse can be used in *FiPy* 3.99. See <http://pysparse.sourceforge.net>.

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 <https://trilinos.github.io/pytrilinos.html>.

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* solver suite for linear solutions. See <http://www.scipy.org/>.

Sphinx

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

steppystones

This package provides iterators that simplify both deterministic and adaptive stepping in time (or other independent variables). See <https://pages.nist.gov/steppystones/en/latest>.

TravisCI

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

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:

```
$ fipy_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:

```
$ fipy_copy_script x.py 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 21

Selected Examples

Many *FiPy examples* are primarily used for integration testing. The following examples are curated to help with understanding how *FiPy* is used.

21.1 Diffusion Examples

Selected illustrations of diffusion problems.

- `examples.diffusion.mesh1D`
- `examples.diffusion.coupled`
- `examples.diffusion.mesh20x20`
- `examples.diffusion.circle`
- `examples.diffusion.electrostatics`
- `examples.diffusion.nthOrder.input4thOrder1D`
- `examples.diffusion.anisotropy`

21.2 Convection Examples

Selected illustrations of convection problems.

- `examples.convection.exponential1D.mesh1D`
- `examples.convection.exponential1DSource.mesh1D`
- `examples.convection.robin`
- `examples.convection.source`

21.3 Phase Field Examples

Selected illustrations of phase field (Allen-Cahn) moving interface problems.

- `examples.phase.simple`
- `examples.phase.binary`
- `examples.phase.binaryCoupled`
- `examples.phase.quaternary`
- `examples.phase.anisotropy`
- `examples.phase.impingement.mesh40x1`
- `examples.phase.impingement.mesh20x20`
- `examples.phase.polyxtal`
- `examples.phase.polyxtalCoupled`

21.4 Level Set Examples

Selected illustrations of level-set moving interface problems.

- `examples.levelSet.distanceFunction.mesh1D`
- `examples.levelSet.distanceFunction.circle`
- `examples.levelSet.advection.mesh1D`
- `examples.levelSet.advection.circle`

21.5 Cahn-Hilliard Examples

Selected illustrations of Cahn-Hilliard (spinodal decomposition) problems.

- `examples.cahnHilliard.mesh2DCoupled`
- `examples.cahnHilliard.sphere`

21.6 Fluid Flow Examples

Selected illustrations of fluid flow problems.

- `examples.flow.stokesCavity`

21.7 Reactive Wetting Examples

Selected illustrations of multi-phase evolution.

- `examples.reactiveWetting.liquidVapor1D`

21.8 Updating FiPy

Demonstrations of how to migrate from older versions of *FiPy*.

- `examples.updating.update2_0to3_0`
- `examples.updating.update1_0to2_0`
- `examples.updating.update0_1to1_0`

Part III

fipy Package Documentation

Chapter 22

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.

package

Each chapter describes one of the main sub-packages of the *fipy* package.

22.1 package

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.

Modules

package.subpackage

Each chapter describes one of the main sub-packages of the *fipy* package.

22.1.1 package.subpackage

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.

Modules

`package.subpackage.base`

This module can be found in the file `package/subpackage/base.py`. You make it available to your script by either::

`package.subpackage.object`

`package.subpackage.base`

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`.

Classes

`Base()`

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::

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.

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.

package.subpackage.object

Classes

Object(arg1[, arg2, arg3])

This method, like all those whose names begin and end with “__” are special.

class package.subpackage.object.**Object**(arg1, arg2=None, arg3='string')

Bases: *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'`.

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()

Object provides a new definition for the behavior of *method2()*, whereas the behavior of *method1()* is defined by *Base*.

<i>fipy</i>	An object oriented, partial differential equation (PDE) solver
<i>examples</i>	Demonstration scripts and high-level tests of the <i>fipy</i> package

Chapter 23

fipy

An object oriented, partial differential equation (PDE) solver

FiPy is 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 [1] [2] [3] treatments of polycrystalline, dendritic, and electrochemical phase transformations, as well as drug eluting stents [4], reactive wetting [5], photovoltaics [6] and a level set treatment of the electrodeposition process [7].

Functions

<code>doctest_raw_input(prompt)</code>	Replacement for <code>raw_input()</code> that works in doctests
<code>test(*args, **kwargs)</code>	Test <i>FiPy</i> . Equivalent to::

`fipy.doctest_raw_input(prompt)`

Replacement for `raw_input()` that works in doctests

This routine attempts to be savvy about running in parallel.

`fipy.test(*args, **kwargs)`

Test *FiPy*. Equivalent to:

```
$ fipy_test --modules
```

Use:

```
$ fipy_test --help
```

for a full list of options. Options can be passed in the same way as they are appended at the command line. For example, to test *FiPy* with *Trilinos* and inlining switched on, use

```
>>> fipy.test(trilinos=True, inline=True)
```

At the command line this would be:

```
$ fipy_test --modules --trilinos --inline
```

Note: A **fipy_test** option like `--deprecation-errors` is equivalent to the `test()` argument `deprecation_errors`.

Modules

<i>fipy.boundaryConditions</i>	Boundary conditions
<i>fipy.matrices</i>	Sparse matrices
<i>fipy.meshes</i>	Domain geometry and topology
<i>fipy.solvers</i>	Solving sparse linear systems
<i>fipy.steppers</i>	(Obsolete) utilities for iterating time steps
<i>fipy.terms</i>	<i>Discretizations</i> of partial differential equation expressions
<i>fipy.testFiPy</i>	Test suite for <i>FiPy</i> modules
<i>fipy.tests</i>	Unit testing scripts
<i>fipy.tools</i>	Utility modules, functions, and values
<i>fipy.variables</i>	Collections of values supporting lazy evaluation
<i>fipy.viewers</i>	Tools for displaying the values of <i>Variable</i> objects

23.1 fipy.boundaryConditions

Boundary conditions

Modules

<i>fipy.boundaryConditions.boundaryCondition</i>	Boundary condition base class
<i>fipy.boundaryConditions.constraint</i>	Restriction on value of a <i>Variable</i>
<i>fipy.boundaryConditions.fixedFlux</i>	Boundary condition of order 1
<i>fipy.boundaryConditions.fixedValue</i>	Boundary condition of order 0
<i>fipy.boundaryConditions.nthOrderBoundaryCondition</i>	Boundary condition of specified derivative order
<i>fipy.boundaryConditions.test</i>	Test boundary conditions

23.1.1 fipy.boundaryConditions.boundaryCondition

Boundary condition base class

Classes

<i>BoundaryCondition</i> (faces, value)	Generic boundary condition base class.

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.

`__repr__()`

Return repr(self).

23.1.2 fipy.boundaryConditions.constraint

Restriction on value of a *Variable*

Classes

<i>Constraint</i> (value[, where])	Holds a <i>Variable</i> to <i>value</i> at <i>where</i>

class fipy.boundaryConditions.constraint.**Constraint**(*value*, *where*=None)

Bases: `object`

Holds a *Variable* to *value* at *where*

see `constrain()`

`__repr__()`

Return repr(self).

23.1.3 fipy.boundaryConditions.fixedFlux

Boundary condition of order 1

Classes

<i>FixedFlux</i> (faces, value)	Adds a Neumann contribution to the system of equations.

class fipy.boundaryConditions.fixedFlux.**FixedFlux**(*faces*, *value*)

Bases: *BoundaryCondition*

Adds a Neumann contribution to the system of equations.

Implements

$$\hat{n} \cdot \vec{J}|_{\text{faces}} = \text{value}$$

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.

__repr__()

Return repr(self).

23.1.4 fipy.boundaryConditions.fixedValue

Boundary condition of order 0

Classes

<i>FixedValue</i> (faces, value)	Adds a Dirichlet contribution to the system of equations.

class fipy.boundaryConditions.fixedValue.**FixedValue**(*faces*, *value*)

Bases: *BoundaryCondition*

Adds a Dirichlet contribution to the system of equations.

Implements

$$\phi|_{\text{faces}} = \text{value}$$

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.

`__repr__()`

Return `repr(self)`.

23.1.5 fipy.boundaryConditions.nthOrderBoundaryCondition

Boundary condition of specified derivative order

Classes

<code>NthOrderBoundaryCondition(faces, value, order)</code>	Adds an appropriate contribution to the system of equations

class `fipy.boundaryConditions.nthOrderBoundaryCondition.NthOrderBoundaryCondition`(*faces*,
value,
order)

Bases: *BoundaryCondition*

Adds an appropriate contribution to the system of equations

Implements

$$\hat{n} \cdot \nabla^{\text{order}} \phi|_{\text{faces}} = \text{value}$$

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.

`__repr__()`

Return `repr(self)`.

23.1.6 fipy.boundaryConditions.test

Test boundary conditions

23.2 fipy.matrices

Sparse matrices

Modules

<code>fipy.matrices.offsetSparseMatrix</code>

<code>fipy.matrices.petscMatrix</code>

<code>fipy.matrices.scipyMatrix</code>

<code>fipy.matrices.sparseMatrix</code>

<code>fipy.matrices.test</code>

<code>fipy.matrices.trilinosMatrix</code>

23.2.1 fipy.matrices.offsetSparseMatrix

Functions

<code>OffsetSparseMatrix</code> (SparseMatrix, ...)	Used in binary terms.

`fipy.matrices.offsetSparseMatrix.OffsetSparseMatrix`(SparseMatrix, numberOfVariables,
numberOfEquations, equationIndex, varIndex)

Used in binary terms. *equationIndex* and *varIndex* need to be set statically before instantiation.

23.2.2 fipy.matrices.petscMatrix

23.2.3 fipy.matrices.scipyMatrix

23.2.4 fipy.matrices.sparseMatrix

23.2.5 fipy.matrices.test

23.2.6 fipy.matrices.trilinosMatrix

23.3 fipy.meshes

Domain geometry and topology

Modules

<code>fipy.meshes.abstractMesh</code>	
<code>fipy.meshes.builders</code>	
<code>fipy.meshes.cylindricalGrid1D</code>	
<code>fipy.meshes.cylindricalGrid2D</code>	
<code>fipy.meshes.cylindricalNonUniformGrid1D</code>	1D Mesh
<code>fipy.meshes.cylindricalNonUniformGrid2D</code>	2D rectangular Mesh
<code>fipy.meshes.cylindricalUniformGrid1D</code>	1D Mesh
<code>fipy.meshes.cylindricalUniformGrid2D</code>	2D cylindrical rectangular Mesh with constant spacing in x and constant spacing in y
<code>fipy.meshes.factoryMeshes</code>	
<code>fipy.meshes.gmshMesh</code>	
<code>fipy.meshes.grid1D</code>	
<code>fipy.meshes.grid2D</code>	
<code>fipy.meshes.grid3D</code>	
<code>fipy.meshes.mesh</code>	
<code>fipy.meshes.mesh1D</code>	Generic mesh class using numerix to do the calculations
<code>fipy.meshes.mesh2D</code>	Generic mesh class using numerix to do the calculations
<code>fipy.meshes.nonUniformGrid1D</code>	1D Mesh
<code>fipy.meshes.nonUniformGrid2D</code>	2D rectangular Mesh
<code>fipy.meshes.nonUniformGrid3D</code>	
<code>fipy.meshes.periodicGrid1D</code>	Periodic 1D Mesh
<code>fipy.meshes.periodicGrid2D</code>	2D periodic rectangular Mesh
<code>fipy.meshes.periodicGrid3D</code>	3D periodic rectangular Mesh
<code>fipy.meshes.representations</code>	
<code>fipy.meshes.skewedGrid2D</code>	
<code>fipy.meshes.sphericalNonUniformGrid1D</code>	1D Mesh
<code>fipy.meshes.sphericalUniformGrid1D</code>	1D Mesh
<code>fipy.meshes.test</code>	Test implementation of the mesh
<code>fipy.meshes.topologies</code>	
<code>fipy.meshes.tri2D</code>	
<code>fipy.meshes.uniformGrid</code>	
<code>fipy.meshes.uniformGrid1D</code>	1D Mesh
<code>fipy.meshes.uniformGrid2D</code>	2D rectangular Mesh with constant spacing in x and constant spacing in y

continues on next page

Table 1 – continued from previous page

fipy.meshes.uniformGrid3D

23.3.1 fipy.meshes.abstractMesh

Classes

AbstractMesh(communicator[, ...])

A class encapsulating all commonalities among meshes in FiPy.

Exceptions

*MeshAdditionError***Exception** raised when meshes cannot be concatenated.

```
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
```

`__div__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

`__getstate__()`

Helper for pickle.

`__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).

__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 'Uniform-Grid1D'

__truediv__(other)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

Topology properties

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.)
>>> print(numerix.allclose((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.allclose((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.allclose((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.allclose((6, 7, 8),
...                          numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent to using `cellCenters[2]`.

```
>>> from fipy import *
>>> print(Grid3D(nx=2, ny=2, nz=2).z)
```

(continues on next page)

(continued from previous page)

```
[ 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.
```

exception `fipy.meshes.abstractMesh.MeshAdditionError`

Bases: `Exception`

`Exception` raised when meshes cannot be concatenated.

`__cause__`

exception cause

`__context__`

exception context

`__delattr__(name, /)`

Implement `delattr(self, name)`.

`__getattr__(name, /)`

Return `getattr(self, name)`.

`__reduce__()`

Helper for pickle.

`__repr__()`

Return `repr(self)`.

`__setattr__(name, value, /)`

Implement `setattr(self, name, value)`.

`__str__()`

Return `str(self)`.

`add_note()`

`Exception.add_note(note)` – add a note to the exception

`with_traceback()`

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

23.3.2 `fipy.meshes.builders`

Modules

<code>fipy.meshes.builders.abstractGridBuilder</code>
<code>fipy.meshes.builders.grid1DBuilder</code>
<code>fipy.meshes.builders.grid2DBuilder</code>
<code>fipy.meshes.builders.grid3DBuilder</code>
<code>fipy.meshes.builders.periodicGrid1DBuilder</code>
<code>fipy.meshes.builders.utilityClasses</code>

fipy.meshes.builders.abstractGridBuilder

fipy.meshes.builders.grid1DBuilder

fipy.meshes.builders.grid2DBuilder

fipy.meshes.builders.grid3DBuilder

fipy.meshes.builders.periodicGrid1DBuilder

fipy.meshes.builders.utilityClasses

23.3.3 fipy.meshes.cylindricalGrid1D

23.3.4 fipy.meshes.cylindricalGrid2D

23.3.5 fipy.meshes.cylindricalNonUniformGrid1D

1D Mesh

Classes

<code>CylindricalNonUniformGrid1D([dx, nx, ...])</code>	Creates a 1D cylindrical grid mesh.

```
class fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D(dx=1.0, nx=None,
                                                                              origin=(0.),
                                                                              overlap=2,
                                                                              communica-
                                                                              tor=DummyComm(),
                                                                              *args, **kwargs)
```

Bases: `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
```

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)
```

(continues on next page)

(continued from previous page)

```
[[ 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
```

__div__(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__()

Helper for pickle.

__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
```

__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)
```

(continues on next page)

(continued from previous page)

```
>>> 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__(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
```

`__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 'Uniform-Grid1D'

`__truediv__`(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent to using `cellCenters[2]`.

```
>>> from fipy import *
>>> print(Grid3D(nx=2, ny=2, nz=2).z)
```

(continues on next page)

(continued from previous page)

```
[ 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.
```

23.3.6 fipy.meshes.cylindricalNonUniformGrid2D

2D rectangular Mesh

Classes

<i>CylindricalNonUniformGrid2D</i> ([dx, dy, nx, ...])	Creates a 2D cylindrical grid mesh with horizontal faces numbered first and then vertical faces.

```
class fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D(dx=1.0, dy=1.0,
                                                                              nx=None,
                                                                              ny=None,
                                                                              origin=((0.0,),
                                                                              (0.0,)), overlap=2,
                                                                              communica-
                                                                              tor=DummyComm(),
                                                                              *args, **kwargs)
```

Bases: *NonUniformGrid2D*

Creates a 2D cylindrical grid mesh with horizontal faces numbered first and then vertical faces.

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
```

`__div__` (*other*)

Tests. `>>> from fipy import *` `>>> print((Grid1D(nx=1) / 2.).cellCenters)` `[[ 0.25]]` `>>> AbstractMesh(communicator=None) / 2.` `Traceback (most recent call last): ... NotImplementedError`

`__getstate__` ()

Helper for pickle.

`__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
```

`__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,
```

(continues on next page)

(continued from previous page)

```
...
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__(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
```

`__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 'Uniform-Grid1D'

`__truediv__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

`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)

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]))
...

```

(continues on next page)

(continued from previous page)

```
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.allegal((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.allegal((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.allegal((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.allegal((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.)
>>> print(numerix.allegal((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.allegal((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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.

```

property z

Equivalent 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.

```

23.3.7 fipy.meshes.cylindricalUniformGrid1D

1D Mesh

Classes

<i>CylindricalUniformGrid1D</i> ([dx, nx, origin, ...])	Creates a 1D cylindrical grid mesh.

```
class fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D(dx=1.0, nx=1, origin=(0.),
                                                                    overlap=2, commu-
                                                                    tor=DummyComm(), *args,
                                                                    **kwargs)
```

Bases: *UniformGrid1D*

Creates a 1D cylindrical grid mesh.

```
>>> mesh = CylindricalUniformGrid1D(nx = 3)
>>> print(mesh.cellCenters)
[[ 0.5  1.5  2.5]]
```

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)
```

(continues on next page)

(continued from previous page)

```
[[ 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
```

__div__(other)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__()

Helper for pickle.

__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)`.

__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 'Uniform-Grid1D'`

__truediv__(*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2.` Traceback (most recent call last): ... `NotImplementedError`

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

property exteriorFaces

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
```

(continues on next page)

(continued from previous page)

```

>>> 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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.

```

property z

Equivalent 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):

```

(continues on next page)

(continued from previous page)

```
...
AttributeError: 1D and 2D meshes do not have a "z" attribute.
```

23.3.8 fipy.meshes.cylindricalUniformGrid2D

2D cylindrical rectangular Mesh with constant spacing in x and constant spacing in y

Classes

<code>CylindricalUniformGrid2D(dx, dy, nx, ny, ...)</code>	Creates a 2D cylindrical grid in the radial and axial directions, appropriate for axial symmetry.

```
class fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D(dx=1.0, dy=1.0, nx=1,
                                                                    ny=1, origin=((0,), (0,)),
                                                                    overlap=2, communicator=DummyComm(), *args,
                                                                    **kwargs)
```

Bases: `UniformGrid2D`

Creates a 2D cylindrical grid in the radial and axial directions, appropriate for axial symmetry.

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)
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

__div__ (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__ ()

Helper for pickle.

__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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).

__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 'Uniform-Grid1D'

__truediv__(other)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs**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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent to using `cellCenters[2]`.

```
>>> from fipy import *
>>> print(Grid3D(nx=2, ny=2, nz=2).z)
```

(continues on next page)

(continued from previous page)

```
[ 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.
```

23.3.9 fipy.meshes.factoryMeshes

Functions

<i>CylindricalGrid1D</i> ([dr, nr, Lr, dx, nx, Lx, ...])	Create a 2D cylindrical mesh
<i>CylindricalGrid2D</i> ([dr, dz, nr, nz, Lr, Lz, ...])	Create a 2D cylindrical mesh
<i>Grid1D</i> ([dx, nx, Lx, overlap, communicator])	Create a 1D Cartesian mesh
<i>Grid2D</i> ([dx, dy, nx, ny, Lx, Ly, overlap, ...])	Create a 2D Cartesian mesh
<i>Grid3D</i> ([dx, dy, dz, nx, ny, nz, Lx, Ly, Lz, ...])	Create a 3D Cartesian mesh
<i>SphericalGrid1D</i> ([dr, nr, Lr, dx, nx, Lx, ...])	Create a 1D spherical mesh

`fipy.meshes.factoryMeshes.CylindricalGrid1D(dr=None, nr=None, Lr=None, dx=1.0, nx=None, Lx=None, origin=(0,), overlap=2, communicator=DummyComm())`

Create a 2D cylindrical mesh

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*) – MPI communicator to use. Select *serialComm* to create a serial mesh when running in parallel; mostly used for test purposes. (default: *parallelComm*).

`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=DummyComm())`

Create a 2D cylindrical mesh

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`) – MPI communicator to use. Select `serialComm` to create a serial mesh when running in parallel; mostly used for test purposes. (default: `parallelComm`).

```
fipy.meshes.factoryMeshes.Grid1D(dx=1.0, nx=None, Lx=None, overlap=2,
                                communicator=DummyComm())
```

Create a 1D Cartesian mesh

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`) – MPI communicator to use. Select `serialComm` to create a serial mesh when running in parallel; mostly used for test purposes. (default: `parallelComm`).

```
fipy.meshes.factoryMeshes.Grid2D(dx=1.0, dy=1.0, nx=None, ny=None, Lx=None, Ly=None, overlap=2,
                                communicator=DummyComm())
```

Create a 2D Cartesian mesh

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`) – MPI communicator to use. Select `serialComm` to create a serial mesh when running in parallel; mostly used for test purposes. (default: `parallelComm`).

```
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, communicator=DummyComm())
```

Create a 3D Cartesian mesh

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`) – MPI communicator to use. Select `serialComm` to create a serial mesh when running in parallel; mostly used for test purposes. (default: `parallelComm`).

```
fipy.meshes.factoryMeshes.SphericalGrid1D(dr=None, nr=None, Lr=None, dx=1.0, nx=None, Lx=None,
                                             origin=(0,), overlap=2, communicator=DummyComm())
```

Create a 1D spherical mesh

Factory function to select between `SphericalUniformGrid1D` and `SphericalNonUniformGrid1D`. 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`) – MPI communicator to use. Select `serialComm` to create a serial mesh when running in parallel; mostly used for test purposes. (default: `parallelComm`).

23.3.10 fipy.meshes.gmshMesh

Functions

<code>gmshVersion([communicator])</code>	Determine the version of Gmsh.
<code>openMSHFile(name[, dimensions, ...])</code>	Open a Gmsh <i>MSH</i> file
<code>openPOSFile(name[, communicator, mode])</code>	Open a Gmsh <i>POS</i> post-processing file

Classes

<code>Gmsh2D(arg[, coordDimensions, communicator, ...])</code>	Construct a 2D Mesh using Gmsh
<code>Gmsh2DIn3DSpace(arg[, communicator, ...])</code>	Create a topologically 2D Mesh in 3D coordinates using Gmsh
<code>Gmsh3D(arg[, communicator, overlap, background])</code>	Create a 3D Mesh using Gmsh
<code>GmshFile(filename, communicator, mode[, ...])</code>	Base class for Gmsh mesh storage files.
<code>GmshGrid2D([dx, dy, nx, ny, ...])</code>	Should serve as a drop-in replacement for <i>Grid2D</i>
<code>GmshGrid3D([dx, dy, dz, nx, ny, nz, ...])</code>	Should serve as a drop-in replacement for <i>Grid3D</i>
<code>MSHFile(filename, dimensions[, ...])</code>	Wrapper for Gmsh MSH storage files.
<code>POSFile(filename, communicator, mode[, ...])</code>	Wrapper for Gmsh POS mesh storage files.

Exceptions

<code>GmshException</code>	Exception raised for Gmsh error conditions.
<code>MeshExportError</code>	Exception raised when FiPy mesh cannot be exported to Gmsh.

class fipy.meshes.gmshMesh.**Gmsh2D**(*arg*, *coordDimensions*=2, *communicator*=*DummyComm()*, *overlap*=1, *background*=None)

Bases: *Mesh2D*

Construct a 2D Mesh using Gmsh

If called in parallel, the mesh will be partitioned based on the value of *parallelComm.Nproc*. If an *MSH* file is supplied, it must have been previously partitioned with the number of partitions matching *parallelComm.Nproc*.

```
>>> 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
...
... ''')
```

(continues on next page)

(continued from previous page)

```

... 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};
... 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};

```

(continues on next page)

(continued from previous page)

```

...
... // 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

```

(continues on next page)

(continued from previous page)

```

...
... // 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
>>> 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

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)
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

`__div__` (*other*)

Tests. `>>> from fipy import *` `>>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]]` `>>> AbstractMesh(communicator=None) / 2.` `Traceback (most recent call last): ... NotImplementedError`

`__getstate__` ()

Helper for pickle.

`__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
```

`__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__(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
```

`__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 'Uniform-Grid1D'

`__truediv__` (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

`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)

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.)
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> ignore = mesh.facesTop.value
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print(numerix.allegal((6, 7, 8),
...                        numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

class `fipy.meshes.gmshMesh.Gmsh2DIn3DSpace`(*arg*, *communicator=DummyComm()*, *overlap=1*, *background=None*)

Bases: `Gmsh2D`

Create a topologically 2D Mesh in 3D coordinates using Gmsh

If called in parallel, the mesh will be partitioned based on the value of `parallelComm.Nproc`. If an *MSH* file is supplied, it must have been previously partitioned with the number of partitions matching `parallelComm.Nproc`.

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

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
```

`__div__`(*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

`__getstate__`()

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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__(*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
```

__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 'Uniform-Grid1D'

__truediv__(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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)

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.allegal((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.)
```

(continues on next page)

(continued from previous page)

```

>>> 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.)
>>> 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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

class fipy.meshes.gmshMesh.**Gmsh3D**(*arg*, *communicator=DummyComm()*, *overlap=1*, *background=None*)

Bases: [Mesh](#)

Create a 3D Mesh using Gmsh

If called in parallel, the mesh will be partitioned based on the value of `parallelComm.Nproc`. If an *MSH* file is supplied, it must have been previously partitioned with the number of partitions matching `parallelComm.Nproc`.

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

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],
```

(continues on next page)

(continued from previous page)

```
...          [ 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
```

`__div__` (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__` ()

Helper for pickle.

`__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
```

`__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__ (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)

```

(continues on next page)

(continued from previous page)

```
[[ 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
```

`__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 'Uniform-Grid1D'

`__truediv__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property `z`

Equivalent 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.
```

exception `fipy.meshes.gmshMesh.GmshException`

Bases: `Exception`

`Exception` raised for Gmsh error conditions.

`__cause__`

exception cause

`__context__`

exception context

`__delattr__(name, /)`

Implement `delattr(self, name)`.

`__getattr__(name, /)`

Return `getattr(self, name)`.

`__reduce__()`

Helper for pickle.

`__repr__()`

Return `repr(self)`.

`__setattr__(name, value, /)`

Implement `setattr(self, name, value)`.

`__str__()`

Return `str(self)`.

`add_note()`

`Exception.add_note(note)` – add a note to the exception

`with_traceback()`

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

class fipy.meshes.gmshMesh.**GmshFile**(filename, communicator, mode, fileIsTemporary=False)

Bases: [object](#)

Base class for Gmsh mesh storage files.

class fipy.meshes.gmshMesh.**GmshGrid2D**(dx=1.0, dy=1.0, nx=1, ny=None, coordDimensions=2, communicator=DummyComm(), overlap=1)

Bases: [Gmsh2D](#)

Should serve as a drop-in replacement for *Grid2D*

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
```

`__div__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> Ab-`

structMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__()`

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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__(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

```

__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 'Uniform-Grid1D'

`__truediv__`(*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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
... 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)

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.allclose((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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

class fipy.meshes.gmshMesh.**GmshGrid3D**(*dx=1.0, dy=1.0, dz=1.0, nx=1, ny=None, nz=None, communicator=DummyComm(), overlap=1*)

Bases: *Gmsh3D*

Should serve as a drop-in replacement for *Grid3D*

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)
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

__div__(*other*)

Tests. `>>> from fipy import *` `>>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]]` `>>> AbstractMesh(communicator=None) / 2.` `Traceback (most recent call last): ... NotImplementedError`

__getstate__()

Helper for pickle.

__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
```

__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__(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
```

`__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 'Uniform-Grid1D'

`__truediv__` (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent to using `cellCenters[2]`.

```
>>> from fipy import *
>>> print(Grid3D(nx=2, ny=2, nz=2).z)
```

(continues on next page)

(continued from previous page)

```
[ 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.
```

```
class fipy.meshes.gmshMesh.MSHFile(filename, dimensions, coordDimensions=None,
                                   communicator=DummyComm(), gmshOutput="", mode='r',
                                   fileIsTemporary=False)
```

Bases: [GmshFile](#)

Wrapper for Gmsh MSH storage files.

Class responsible for parsing a Gmsh file and then readying its contents for use by a *Mesh* constructor.

Can handle a partitioned mesh based on *parallelComm.Nproc*. If partitioning, the *.msh* file must either be previously partitioned with the number of partitions matching *Nproc*, or the mesh must be specified with a *.geo* file or multiline string.

Does not support gmsh versions < 2. If partitioning, gmsh version must be >= 2.5.

TODO: Refactor face extraction functions.

Parameters

- **filename** ([str](#)) – Gmsh output file
- **dimensions** ([int](#)) – Dimension of mesh
- **coordDimensions** ([int](#)) – Dimension of shapes
- **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.
- **gmshOutput** ([str](#)) – Output (if any) from Gmsh run that created *.msh* file
- **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.
- **fileIsTemporary** ([bool](#)) – If *True*, *filename* should be cleaned up on deletion

makeMapVariables (*mesh*)

Utility function to make *MeshVariables* that define different domains in the mesh

read()

0. Build *cellsToVertices*
1. Recover needed *vertexCoords* and mapping from file using *cellsToVertices*
2. Build *cellsToVertIDs* proper from *vertexCoords* and vertex map
3. Build faces
4. Build *cellsToFaces*

Isolate relevant data into three files, store in *self.nodesPath* for *\$Nodes*, *self.elemsPath* for *\$Elements*, *self.namesFile* for *\$PhysicalNames*.

Returns *vertexCoords*, *facesToVertexID*, *cellsToFaceID*,
cellGlobalIDMap, *ghostCellGlobalIDMap*.

exception `fiPy.meshes.gmshMesh.MeshExportError`

Bases: [*GmshException*](#)

Exception raised when FiPy mesh cannot be exported to Gmsh.

__cause__

exception cause

__context__

exception context

__delattr__(*name*, /)

Implement `delattr(self, name)`.

__getattr__(*name*, /)

Return `getattr(self, name)`.

__reduce__()

Helper for pickle.

__repr__()

Return `repr(self)`.

__setattr__(*name*, *value*, /)

Implement `setattr(self, name, value)`.

__str__()

Return `str(self)`.

add_note()

`Exception.add_note(note)` – add a note to the exception

with_traceback()

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

class `fiPy.meshes.gmshMesh.POSFile(filename, communicator, mode, fileIsTemporary=False)`

Bases: [*GmshFile*](#)

Wrapper for Gmsh POS mesh storage files.

`fiPy.meshes.gmshMesh.gmshVersion(communicator=DummyComm())`

Determine the version of Gmsh.

We can't trust the generated *.msh* file for the correct version number, so we have to retrieve it from the gmsh binary.

`fiPy.meshes.gmshMesh.openMSHFile(name, dimensions=None, coordDimensions=None, communicator=DummyComm(), 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=DummyComm(), mode='w')`

Open a Gmsh *POS* post-processing file

23.3.11 fipy.meshes.grid1D

23.3.12 fipy.meshes.grid2D

23.3.13 fipy.meshes.grid3D

23.3.14 fipy.meshes.mesh

Classes

<i>Mesh</i> (vertexCoords, faceVertexIDs, cellFaceIDs)	Generic mesh class using numerix to do the calculations

Exceptions

<i>MeshAdditionError</i>

```
class fipy.meshes.mesh.Mesh(vertexCoords, faceVertexIDs, cellFaceIDs, communicator=DummyComm(),
                             _RepresentationClass=<class
                             'fipy.meshes.representations.meshRepresentation._MeshRepresentation'>,
                             _TopologyClass=<class
                             'fipy.meshes.topologies.meshTopology._MeshTopology'>)
```

Bases: *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.

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
```

`__div__` (*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

`__getstate__` ()

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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__(*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
```

__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 'Uniform-Grid1D'

__truediv__(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs**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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.

```

property `z`

Equivalent 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.

```

exception `fipy.meshes.mesh.MeshAdditionError`

Bases: `Exception`

`__cause__`

exception cause

`__context__`

exception context

`__delattr__(name, /)`

Implement `delattr(self, name)`.

`__getattr__(name, /)`

Return `getattr(self, name)`.

`__reduce__()`

Helper for pickle.

`__repr__()`

Return `repr(self)`.

`__setattr__(name, value, /)`

Implement `setattr(self, name, value)`.

`__str__()`

Return `str(self)`.

`add_note()`

`Exception.add_note(note)` – add a note to the exception

`with_traceback()`

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

23.3.15 fipy.meshes.mesh1D

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

Classes

Mesh1D(vertexCoords, faceVertexIDs, cellFaceIDs)

```
class fipy.meshes.mesh1D.Mesh1D(vertexCoords, faceVertexIDs, cellFaceIDs,
                                communicator=DummyComm(), _RepresentationClass=<class
                                'fipy.meshes.representations.meshRepresentation._MeshRepresentation'>,
                                _TopologyClass=<class
                                'fipy.meshes.topologies.meshTopology._Mesh1DTopology'>)
```

Bases: *Mesh*

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)
```

(continues on next page)

(continued from previous page)

```
[[ 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
```

`__div__(other)`

Tests. `>>> from fipy import *` `>>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]]` `>>> AbstractMesh(communicator=None) / 2.` `Traceback (most recent call last): ... NotImplementedError`

`__getstate__()`

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
>>> 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__(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
```

`__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 'Uniform-Grid1D'

`__truediv__` (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent to using `cellCenters[2]`.

```
>>> from fipy import *
>>> print(Grid3D(nx=2, ny=2, nz=2).z)
```

(continues on next page)

(continued from previous page)

```
[ 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.
```

23.3.16 fipy.meshes.mesh2D

Generic mesh class using numerix to do the calculations

Meshes contain cells, faces, and vertices.

This is built for a non-mixed element mesh.

Classes

```
Mesh2D(vertexCoords, faceVertexIDs, cellFaceIDs)
```

```
class fipy.meshes.mesh2D.Mesh2D(vertexCoords, faceVertexIDs, cellFaceIDs,
                                   communicator=DummyComm(), _RepresentationClass=<class
                                   'fipy.meshes.representations.meshRepresentation._MeshRepresentation'>,
                                   _TopologyClass=<class
                                   'fipy.meshes.topologies.meshTopology._Mesh2DTopology'>)
```

Bases: *Mesh*

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

```

__div__(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__()

Helper for pickle.

__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

```

`__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,
```

(continues on next page)

(continued from previous page)

```
...
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__(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
```

`__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 'Uniform-Grid1D'

`__truediv__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

`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)

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]))
...

```

(continues on next page)

(continued from previous page)

```
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.allegal((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.allegal((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.allegal((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.allegal((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.)
>>> print(numerix.allegal((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.allegal((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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.

```

property z

Equivalent 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.

```

23.3.17 fipy.meshes.nonUniformGrid1D

1D Mesh

Classes

<i>NonUniformGrid1D</i> ([dx, nx, overlap, ...])	Creates a 1D grid mesh.

```
class fipy.meshes.nonUniformGrid1D.NonUniformGrid1D(dx=1.0, nx=None, overlap=2,
                                                    communicator=DummyComm(),
                                                    _BuilderClass=<class
'fipy.meshes.builders.grid1DBuilder._NonuniformGrid1DBuilder'>
                                                    _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid1DRepresent
                                                    _TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid1DTopology'>)
```

Bases: [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)
```

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],
```

(continues on next page)

(continued from previous page)

```

...             [ 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

```

`__div__` (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__` ()

Helper for pickle.

`__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
```

`__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__ (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)

```

(continues on next page)

(continued from previous page)

```
[[ 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
```

`__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 'Uniform-Grid1D'`

`__truediv__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2.` Traceback (most recent call last): ... `NotImplementedError`

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property `z`

Equivalent 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.
```

23.3.18 `fipy.meshes.nonUniformGrid2D`

2D rectangular Mesh

Classes

<code>NonUniformGrid2D([dx, dy, nx, ny, overlap, ...])</code>	Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces.

```
class fipy.meshes.nonUniformGrid2D.NonUniformGrid2D(dx=1.0, dy=1.0, nx=None, ny=None, overlap=2,
                                                    communicator=DummyComm(),
                                                    _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid2DRepresent
_TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid2DTopology'>>)
```

Bases: `Mesh2D`

Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces.

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)
```

(continues on next page)

(continued from previous page)

```
[[ 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

```

`__div__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__()`

Helper for pickle.

`__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
```

`__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__(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
```

`__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

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

`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
↳ ],
...
↳ 83333333, 0.5,      0.16666667, 0.5,      0.83333333,
...                               0.5,      0.
↳ 16666667],
...
↳ 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)

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]))
...

```

(continues on next page)

(continued from previous page)

```

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.)
>>> 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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.

```

property z

Equivalent 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):

```

(continues on next page)

(continued from previous page)

```
...
AttributeError: 1D and 2D meshes do not have a "z" attribute.
```

23.3.19 fipy.meshes.nonUniformGrid3D

Classes

<i>NonUniformGrid3D</i> ([dx, dy, dz, nx, ny, nz, ...])	3D rectangular-prism Mesh

```
class fiPy.meshes.nonUniformGrid3D.NonUniformGrid3D(dx=1.0, dy=1.0, dz=1.0, nx=None, ny=None,
                                                    nz=None, overlap=2,
                                                    communicator=DummyComm(),
                                                    _RepresentationClass=<class
'fiPy.meshes.representations.gridRepresentation._Grid3DRepresent
'_TopologyClass=<class
'fiPy.meshes.topologies.gridTopology._Grid3DTopology'>)
```

Bases: *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.

```
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

```

__div__(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__()

Helper for pickle.

__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

```

`__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,
```

(continues on next page)

(continued from previous page)

```
...
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__(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
```

`__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 'Uniform-Grid1D'`

`__truediv__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2.` Traceback (most recent call last): ... `NotImplementedError`

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent to using `cellCenters[2]`.

```
>>> from fipy import *
>>> print(Grid3D(nx=2, ny=2, nz=2).z)
```

(continues on next page)

(continued from previous page)

```
[ 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.
```

23.3.20 fipy.meshes.periodicGrid1D

Periodic 1D Mesh

Classes

<i>PeriodicGrid1D</i> ([dx, nx, overlap])	Creates a Periodic grid mesh.

class fipy.meshes.periodicGrid1D.**PeriodicGrid1D**(dx=1.0, nx=None, overlap=2, *args, **kwargs)

Bases: *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
```

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,))
```

(continues on next page)

(continued from previous page)

```

>>> 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

```

`__div__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

`__getstate__()`

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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__(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
```

__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 'Uniform-Grid1D'

__truediv__(other)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

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).

property cellFaceIDs**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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

23.3.21 fipy.meshes.periodicGrid2D

2D periodic rectangular Mesh

Classes

<code>PeriodicGrid2D</code> (<code>dx</code> , <code>dy</code> , <code>nx</code> , <code>ny</code> , <code>overlap</code> , ...)	Creates a periodic 2D grid mesh with horizontal faces numbered first and then vertical faces.
<code>PeriodicGrid2DLeftRight</code> (<code>dx</code> , <code>dy</code> , <code>nx</code> , <code>ny</code> , ...)	
<code>PeriodicGrid2DTopBottom</code> (<code>dx</code> , <code>dy</code> , <code>nx</code> , <code>ny</code> , ...)	

```
class fipy.meshes.periodicGrid2D.PeriodicGrid2D(dx=1.0, dy=1.0, nx=None, ny=None, overlap=2,
communicator=DummyComm(), *args, **kwargs)
```

Bases: `_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],
```

(continues on next page)

(continued from previous page)

```
...           [ 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
```

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
```

`__div__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

`__getstate__()`

Helper for pickle.

`__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
```

`__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],
```

(continues on next page)

(continued from previous page)

```

...         [ 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__(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
```

__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

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs**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)

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.)
>>> 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),

```

(continues on next page)

(continued from previous page)

```

...
numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.

```

property z

Equivalent 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.

```

```

class fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight(dx=1.0, dy=1.0, nx=None, ny=None,
    overlap=2,
    communicator=DummyComm(), *args,
    **kwargs)

```

Bases: `_BasePeriodicGrid2D`

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,
```

(continues on next page)

(continued from previous page)

```

...         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

```

__div__(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__()

Helper for pickle.

__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
```

`__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__(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
```

`__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

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

`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
↳ ],
...
↳ 83333333, 0.5,      0.16666667, 0.5,      0.83333333,
...                                0.5,      0.
↳ 16666667],
...
↳ 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)

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]))
...

```

(continues on next page)

(continued from previous page)

```

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.)
>>> 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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.

```

property z

Equivalent 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):

```

(continues on next page)

(continued from previous page)

```
...
AttributeError: 1D and 2D meshes do not have a "z" attribute.
```

```
class fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom(dx=1.0, dy=1.0, nx=None, ny=None,
                                                         overlap=2,
                                                         communicator=DummyComm(), *args,
                                                         **kwargs)
```

Bases: `_BasePeriodicGrid2D`

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
```

`__div__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> Ab-`

stratMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__()`

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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__(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
```

__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 'Uniform-Grid1D'

`__truediv__`(*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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 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)

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property `z`

Equivalent 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.
```

23.3.22 fipy.meshes.periodicGrid3D

3D periodic rectangular Mesh

Classes

<code>PeriodicGrid3D([dx, dy, dz, nx, ny, nz, ...])</code>	Creates a periodic 3D grid mesh with horizontal faces numbered first and then vertical faces.
<code>PeriodicGrid3DFrontBack([dx, dy, dz, nx, ...])</code>	
<code>PeriodicGrid3DLeftRight([dx, dy, dz, nx, ...])</code>	
<code>PeriodicGrid3DLeftRightFrontBack([dx, dy, ...])</code>	
<code>PeriodicGrid3DLeftRightTopBottom([dx, dy, ...])</code>	
<code>PeriodicGrid3DTopBottom([dx, dy, dz, nx, ...])</code>	
<code>PeriodicGrid3DTopBottomFrontBack([dx, dy, ...])</code>	

```
class fipy.meshes.periodicGrid3D.PeriodicGrid3D(dx=1.0, dy=1.0, dz=1.0, nx=None, ny=None,
                                                  nz=None, overlap=2,
                                                  communicator=DummyComm(), *args, **kwargs)
```

Bases: `_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]]))
```

(continues on next page)

(continued from previous page)

True

property VTKCellDataSetReturns 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

```

`__div__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__()`

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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__(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
```

__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 'Uniform-Grid1D'

__truediv__(other)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs**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),
```

(continues on next page)

(continued from previous page)

```

...                     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.)
>>> 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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.

```

property z

Equivalent 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.

```

```

class fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack(dx=1.0, dy=1.0, dz=1.0, nx=None,
                                                         ny=None, nz=None, overlap=2,
                                                         communicator=DummyComm(), *args,
                                                         **kwargs)

```

Bases: `_BasePeriodicGrid3D`

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,
```

(continues on next page)

(continued from previous page)

```
...
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
```

`__div__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

`__getstate__()`

Helper for pickle.

`__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
```

`__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__`(*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
```

`__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 'Uniform-Grid1D'`

`__truediv__`(*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2.` Traceback (most recent call last): ... `NotImplementedError`

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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),
```

(continues on next page)

(continued from previous page)

```

...                     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

```

(continues on next page)

(continued from previous page)

```

>>> ignore = mesh.facesLeft.value
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print(numerix.allegal((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.allegal((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.allegal((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.)
>>> print(numerix.allegal((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.allegal((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.allegal((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.allegal((6, 7, 8),
...                       numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

```
class fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight(dx=1.0, dy=1.0, dz=1.0, nx=None,
                                                         ny=None, nz=None, overlap=2,
                                                         communicator=DummyComm(), *args,
                                                         **kwargs)
```

Bases: `_BasePeriodicGrid3D`

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)
```

(continues on next page)

(continued from previous page)

```
[[ 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]]
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

__div__(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__()

Helper for pickle.

__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```

>>> 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__(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)

```

(continues on next page)

(continued from previous page)

```
[[ 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
```

`__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 'Uniform-Grid1D'

`__truediv__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

```
class fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack(dx=1.0, dy=1.0, dz=1.0,
                                                                    nx=None, ny=None, nz=None,
                                                                    overlap=2, communicator=DummyComm(), *args,
                                                                    **kwargs)
```

Bases: `_BasePeriodicGrid3D`

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)
```

(continues on next page)

(continued from previous page)

```

>>> 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

```

`__div__` (*other*)

Tests. `>>> from fipy import *` `>>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]]` `>>> AbstractMesh(communicator=None) / 2.` `Traceback (most recent call last): ... NotImplementedError`

`__getstate__` ()

Helper for pickle.

`__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

```

`__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__(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
```

`__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 'Uniform-Grid1D'

`__truediv__` (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent to using `cellCenters[2]`.

```
>>> from fipy import *
>>> print(Grid3D(nx=2, ny=2, nz=2).z)
```

(continues on next page)

(continued from previous page)

```
[ 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.
```

```
class fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom(dx=1.0, dy=1.0, dz=1.0,
                                                                    nx=None, ny=None, nz=None,
                                                                    overlap=2, communicator=DummyComm(), *args,
                                                                    **kwargs)
```

Bases: `_BasePeriodicGrid3D`

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
```

`__div__` (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__()`

Helper for pickle.

`__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
```

`__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__ (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
```

__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 'Uniform-Grid1D'

`__truediv__`(*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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.)
>>> 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.allclose((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.allclose((6, 7, 8),
...                         numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property **z**

Equivalent 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.
```

```
class fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom(dx=1.0, dy=1.0, dz=1.0, nx=None,
                                                         ny=None, nz=None, overlap=2,
                                                         communicator=DummyComm(), *args,
                                                         **kwargs)
```

Bases: `_BasePeriodicGrid3D`

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,
```

(continues on next page)

(continued from previous page)

```
...
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
```

`__div__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

`__getstate__()`

Helper for pickle.

`__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
```

`__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__`(*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
```

`__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 'Uniform-Grid1D'`

`__truediv__`(*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2.` Traceback (most recent call last): ... `NotImplementedError`

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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),
```

(continues on next page)

(continued from previous page)

```

...                     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

```

(continues on next page)

(continued from previous page)

```

>>> ignore = mesh.facesLeft.value
>>> mesh = Grid2D(nx = 3, ny = 2, dx = 0.5, dy = 2.)
>>> print(numerix.allegal((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.allegal((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.allegal((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.)
>>> print(numerix.allegal((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.allegal((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.allegal((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.allegal((6, 7, 8),
...                       numerix.nonzero(mesh.facesTop)[0]))
True
>>> ignore = mesh.facesTop.value

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

```
class fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack(dx=1.0, dy=1.0, dz=1.0,
                                                                    nx=None, ny=None, nz=None,
                                                                    overlap=2, communicator=DummyComm(), *args,
                                                                    **kwargs)
```

Bases: `_BasePeriodicGrid3D`

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],
```

(continues on next page)

(continued from previous page)

```

...             [ 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

```

`__div__`(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__`()

Helper for pickle.

`__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
```

`__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__(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)

```

(continues on next page)

(continued from previous page)

```
[[ 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
```

`__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 'Uniform-Grid1D'`

`__truediv__(other)`

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2.` Traceback (most recent call last): ... `NotImplementedError`

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property `z`

Equivalent 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.
```

23.3.23 fipy.meshes.representations**Modules**

```
fipy.meshes.representations.  
abstractRepresentation  
fipy.meshes.representations.  
gridRepresentation  
fipy.meshes.representations.  
meshRepresentation
```

fipy.meshes.representations.abstractRepresentation**fipy.meshes.representations.gridRepresentation****fipy.meshes.representations.meshRepresentation****23.3.24 fipy.meshes.skewedGrid2D****Classes**

<code>SkewedGrid2D([dx, dy, nx, ny, rand])</code>	Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces.

```
class fipy.meshes.skewedGrid2D.SkewedGrid2D(dx=1.0, dy=1.0, nx=None, ny=1, rand=0, *args,
**kwargs)
```

Bases: `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

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
```

`__div__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__()`

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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__(*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
```

__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 'Uniform-Grid1D'

__truediv__(*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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)

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.allegal((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.)
```

(continues on next page)

(continued from previous page)

```

>>> 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.)
>>> 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

```

property physicalShape

Return physical dimensions of *Grid2D*.

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property `z`

Equivalent 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.
```

23.3.25 `fipy.meshes.sphericalNonUniformGrid1D`

1D Mesh

Classes

<code><i>SphericalNonUniformGrid1D</i>([dx, nx, origin, ...])</code>	Creates a 1D spherical grid mesh.

```
class fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D(dx=1.0, nx=None,
                                                                           origin=(0.), overlap=2,
                                                                           communica-
                                                                           tor=DummyComm(),
                                                                           *args, **kwargs)
```

Bases: `NonUniformGrid1D`

Creates a 1D spherical grid mesh.

```
>>> mesh = SphericalNonUniformGrid1D(nx = 3)
>>> print(mesh.cellCenters)
[[ 0.5  1.5  2.5]]
```

```
>>> mesh = SphericalNonUniformGrid1D(dx = (1, 2, 3))
>>> print(mesh.cellCenters)
[[ 0.5  2.  4.5]]
```

```
>>> print(numerix.allclose(mesh.cellVolumes, (0.5, 13., 94.5)))
True
```



```
>>> mesh = SphericalNonUniformGrid1D(nx = 2, dx = (1, 2, 3))
Traceback (most recent call last):
...
IndexError: nx != len(dx)
```

```
>>> mesh = SphericalNonUniformGrid1D(nx=2, dx=(1., 2.)) + ((1.,),)
>>> print(mesh.cellCenters)
[[ 1.5  3. ]]
>>> print(numerix.allclose(mesh.cellVolumes, (3.5, 28)))
True
```

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
```

`__div__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> Ab-

stratMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__()`

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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__(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
```

__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 'Uniform-Grid1D'

`__truediv__`(*other*)

Tests. `>>> from fipy import *` `>>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]]` `>>> AbstractMesh(communicator=None) / 2.` Traceback (most recent call last): ... `NotImplementedError`

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```


23.3.26 fipy.meshes.sphericalUniformGrid1D

1D Mesh

Classes

<code>SphericalUniformGrid1D(dx, nx, origin, ...)</code>	Creates a 1D spherical grid mesh.

```
class fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D(dx=1.0, nx=1, origin=(0,),
                                                                    overlap=2,
                                                                    communicator=DummyComm(),
                                                                    *args, **kwargs)
```

Bases: `UniformGrid1D`

Creates a 1D spherical grid mesh.

```
>>> mesh = SphericalUniformGrid1D(nx = 3)
>>> print(mesh.cellCenters)
[[ 0.5  1.5  2.5]]
```

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)
```

(continues on next page)

(continued from previous page)

```
[[ 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
```

`__div__(other)`

Tests. `>>> from fipy import *` `>>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]]` `>>> AbstractMesh(communicator=None) / 2.` `Traceback (most recent call last): ... NotImplementedError`

`__getstate__()`

Helper for pickle.

`__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).

__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 'Uniform-Grid1D'

__truediv__(other)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

property exteriorFaces

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
```

(continues on next page)

(continued from previous page)

```

>>> 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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.

```

property z

Equivalent 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):

```

(continues on next page)

(continued from previous page)

```
...
AttributeError: 1D and 2D meshes do not have a "z" attribute.
```

23.3.27 fipy.meshes.test

Test implementation of the mesh

23.3.28 fipy.meshes.topologies

Modules

fipy.meshes.topologies.abstractTopology

fipy.meshes.topologies.gridTopology

fipy.meshes.topologies.meshTopology

fipy.meshes.topologies.abstractTopology

fipy.meshes.topologies.gridTopology

fipy.meshes.topologies.meshTopology

23.3.29 fipy.meshes.tri2D

Classes

Tri2D([dx, dy, nx, ny, ...])

This class creates a mesh made out of triangles.

```
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: *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** (*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.
- **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** (*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 $4 * nx * ny$.
- **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 $4 * nx * ny$.

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
```

`__div__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> Ab-

stratMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__()`

Helper for pickle.

`__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
```

`__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)
```

(continues on next page)

(continued from previous page)

```
[[ 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__(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
```

__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 'Uniform-Grid1D'

`__truediv__`(*other*)

Tests. `>>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[ 0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError`

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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
... 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)

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.)
>>> 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
```

property physicalShape

Return physical dimensions of *Grid2D*.

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

23.3.30 fipy.meshes.uniformGrid

Classes

UniformGrid(communicator[, ...])

Wrapped scaled geometry properties

```
class fipy.meshes.uniformGrid.UniformGrid(communicator, _RepresentationClass=<class
    'fipy.meshes.representations.abstractRepresentation._AbstractRepresentation'>,
    _TopologyClass=<class
    'fipy.meshes.topologies.abstractTopology._AbstractTopology'>)
```

Bases: *AbstractMesh*

Wrapped scaled geometry properties

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],
```

(continues on next page)

(continued from previous page)

```

...             [ 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

```

`__div__` (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__` ()

Helper for pickle.

`__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)

```

(continues on next page)

(continued from previous page)

```
[[ 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).

__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 'Uniform-Grid1D'

__truediv__(other)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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),
```

(continues on next page)

(continued from previous page)

```

...                               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.)
>>> 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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

23.3.31 fipy.meshes.uniformGrid1D

1D Mesh

Classes

<code>UniformGrid1D([dx, nx, origin, overlap, ...])</code>	Creates a 1D grid mesh.

```
class fipy.meshes.uniformGrid1D.UniformGrid1D(dx=1.0, nx=1, origin=(0, ), overlap=2,
communicator=DummyComm(),
_RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid1DRepresentation'>,
_TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid1DTopology'>)
```

Bases: `UniformGrid`

Creates a 1D grid mesh.

```
>>> mesh = UniformGrid1D(nx = 3)
>>> print(mesh.cellCenters)
[[ 0.5  1.5  2.5]]
```

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
```

__div__ (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__ ()

Helper for pickle.

__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).

__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 'Uniform-Grid1D'

__truediv__(other)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

property exteriorFaces

Geometry set and calc

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.)
>>> 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
```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

23.3.32 fipy.meshes.uniformGrid2D

2D rectangular Mesh with constant spacing in x and constant spacing in y

Classes

<code>UniformGrid2D([dx, dy, nx, ny, origin, ...])</code>	Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces.

```
class fipy.meshes.uniformGrid2D.UniformGrid2D(dx=1.0, dy=1.0, nx=1, ny=1, origin=((0, ), (0, )),
        overlap=2, communicator=DummyComm(),
        _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid2DRepresentation'>,
        _TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid2DTopology'>)
```

Bases: `UniformGrid`

Creates a 2D grid mesh with horizontal faces numbered first and then vertical faces.

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)
```

(continues on next page)

(continued from previous page)

```
[[ 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

```

`__div__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

`__getstate__()`

Helper for pickle.

`__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).

__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 'Uniform-Grid1D'

__truediv__(other)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property aspect2D

The physical y vs x aspect ratio of a 2D mesh

property cellCenters

Coordinates of geometric centers of cells

property cellFaceIDs

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),

```

(continues on next page)

(continued from previous page)

```

...                               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.)
>>> 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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

23.3.33 fipy.meshes.uniformGrid3D**Classes**

<i>UniformGrid3D</i> ([dx, dy, dz, nx, ny, nz, ...])	3D rectangular-prism Mesh with uniform grid spacing in each dimension.

```
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=DummyComm(),
        _RepresentationClass=<class
'fipy.meshes.representations.gridRepresentation._Grid3DRepresentation'>,
        _TopologyClass=<class
'fipy.meshes.topologies.gridTopology._Grid3DTopology'>)
```

Bases: *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.

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],
```

(continues on next page)

(continued from previous page)

```

...             [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

```

__div__ (*other*)

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

__getstate__ ()

Helper for pickle.

__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,
```

(continues on next page)

(continued from previous page)

```

...         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).

`__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 'Uniform-Grid1D'

`__truediv__(other)`

Tests. >>> from fipy import * >>> print((Grid1D(nx=1) / 2.).cellCenters) [[0.25]] >>> AbstractMesh(communicator=None) / 2. Traceback (most recent call last): ... NotImplementedError

property `aspect2D`

The physical y vs x aspect ratio of a 2D mesh

property `cellCenters`

Coordinates of geometric centers of cells

property `cellFaceIDs`

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.allegal((6, 7, 8, 9, 10, 11),
...                       numerix.nonzero(mesh.facesBack)[0]))
...

```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```

>>> 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.)
>>> 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

```

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):
...
AttributeError: 1D meshes do not have a "y" attribute.
```

property z

Equivalent 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.
```

23.4 fipy.solvers

Solving sparse linear systems

Module Attributes

<i>DefaultSolver</i>	Solver class for solving symmetric matrices.
<i>DefaultAsymmetricSolver</i>	Solver class for solving asymmetric matrices.
<i>DummySolver</i>	Solver used by tests that don't actually need to solve.
<i>GeneralSolver</i>	Solver class that should solve any matrix.

Exceptions

SerialSolverError()

fipy.solvers.DefaultAsymmetricSolver

Solver class for solving asymmetric matrices.

fipy.solvers.DefaultSolver

Solver class for solving symmetric matrices.

This solver should be both robust and performant.

fipy.solvers.DummySolver

Solver used by tests that don't actually need to solve.

Some tests are intended to confirm the matrix building machinery, but don't actually need to solve (and may not be able to, e.g., zeros on the diagonal).

fipy.solvers.GeneralSolver

Solver class that should solve any matrix.

exception fipy.solvers.SerialSolverError

Bases: `Exception`

__cause__

exception cause

__context__

exception context

__delattr__(*name*, /)

Implement `delattr(self, name)`.

__getattr__(*name*, /)

Return `getattr(self, name)`.

__reduce__()

Helper for pickle.

__repr__()

Return `repr(self)`.

__setattr__(*name*, *value*, /)

Implement `setattr(self, name, value)`.

__str__()

Return `str(self)`.

add_note()

`Exception.add_note(note)` – add a note to the exception

with_traceback()

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

Modules

<code>fipy.solvers.convergence</code>	
<code>fipy.solvers.petsc</code>	
<code>fipy.solvers.preconditioner</code>	
<code>fipy.solvers.pyAMG</code>	
<code>fipy.solvers.pyamgx</code>	
<code>fipy.solvers.scipy</code>	
<code>fipy.solvers.solver</code>	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::
<code>fipy.solvers.test</code>	
<code>fipy.solvers.trilinos</code>	

23.4.1 fipy.solvers.convergence

Classes

<code>AbsoluteToleranceConvergence(solver, ...[, ...])</code>	Absolute tolerance satisfied
<code>BreakdownDivergence(solver, iterations, ...)</code>	Method broke down.
<code>Convergence(solver, iterations, residual, ...)</code>	Information about why a solver converged.
<code>ConvergenceBase(solver, iterations, ...[, ...])</code>	Information about whether and why a solver converged.
<code>Divergence(solver, iterations, residual, ...)</code>	Information about why a solver diverged.
<code>HappyBreakdownConvergence(solver, ...[, ...])</code>	"Exact" solution found and more iterations will just make things worse.
<code>IllConditionedDivergence(solver, iterations, ...)</code>	Matrix was ill-conditioned.
<code>IllConditionedPreconditionerDivergence(...)</code>	Preconditioner is ill-conditioned.
<code>IteratingConvergence(solver, iterations, ...)</code>	Solve still in progress.
<code>IterationConvergence(solver, iterations, ...)</code>	Requested iterations complete (and no residual calculated).
<code>IterationDivergence(solver, iterations, ...)</code>	Exceeded maximum iterations.
<code>LossOfAccuracyConvergence(solver, ...[, ...])</code>	Numerical loss of precision occurred.
<code>NullDivergence(solver, iterations, residual, ...)</code>	Breakdown when solving the Hessenberg system within GMRES.
<code>OutOfRangeDivergence(solver, iterations, ...)</code>	A value became too small, too large, or invalid.
<code>PreconditioningDivergence(solver, ...[, ...])</code>	A problem with the preconditioner.
<code>RHSZeroConvergence(solver, iterations, ...)</code>	$\vec{b} = 0$, so exact solution is $\vec{x} = 0$.
<code>RelativeToleranceConvergence(solver, ...[, ...])</code>	Relative tolerance satisfied
<code>StagnatedDivergence(solver, iterations, ...)</code>	The method stagnated.
<code>ToleranceDivergence(solver, iterations, ...)</code>	Residual norm increased too much.

Exceptions

<i>DivergenceWarning</i> (divergence)	Warning raised when solver diverges.

```
class fipy.solvers.convergence.AbsoluteToleranceConvergence(solver, iterations, residual, criterion,
                                                            actual_code=None, **kwargs)
```

Bases: *Convergence*

Absolute tolerance satisfied

::

residual < atol * scale

__str__()

Return str(self).

```
class fipy.solvers.convergence.BreakdownDivergence(solver, iterations, residual, criterion,
                                                    actual_code=None, **kwargs)
```

Bases: *Divergence*

Method broke down.

__str__()

Return str(self).

```
class fipy.solvers.convergence.Convergence(solver, iterations, residual, criterion, actual_code=None,
                                            **kwargs)
```

Bases: *ConvergenceBase*

Information about why a solver converged.

__str__()

Return str(self).

```
class fipy.solvers.convergence.ConvergenceBase(solver, iterations, residual, criterion,
                                                actual_code=None, **kwargs)
```

Bases: *object*

Information about whether and why a solver converged.

solver

The linear solver that was invoked.

Type

Solver

iterations

The number of linear iterations the solver performed.

Type

int

criterion

The *Convergence* test used by the solver.

Type

str

tolerance_scale

The multiplier applied to the tolerance in order for this solver to satisfy *criterion*.

Type
float

residual

The unscaled norm of the residual achieved by the solver.

Type
float

status_code

The canonical return value for this type of convergence.

Type
int or str

status_name

The text representation of *status_code*.

Type
str

actual_code

The status value actually returned by the solver.

Type
int or str

__str__()

Return str(self).

class fipy.solvers.convergence.**Divergence**(*solver, iterations, residual, criterion, actual_code=None, **kwargs*)

Bases: [ConvergenceBase](#)

Information about why a solver diverged.

__str__()
Return str(self).

exception fipy.solvers.convergence.**DivergenceWarning**(*divergence*)

Bases: [UserWarning](#)

Warning raised when solver diverges.

__cause__
exception cause

__context__
exception context

__delattr__(*name, /*)
Implement delattr(self, name).

__getattr__(*name, /*)
Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(*name, value, /*)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class fipy.solvers.convergence.**HappyBreakdownConvergence**(*solver, iterations, residual, criterion, actual_code=None, **kwargs*)

Bases: [Convergence](#)

“Exact” solution found and more iterations will just make things worse.

__str__()

Return str(self).

class fipy.solvers.convergence.**IllConditionedDivergence**(*solver, iterations, residual, criterion, actual_code=None, **kwargs*)

Bases: [Divergence](#)

Matrix was ill-conditioned.

__str__()

Return str(self).

class fipy.solvers.convergence.**IllConditionedPreconditionerDivergence**(*solver, iterations, residual, criterion, actual_code=None, **kwargs*)

Bases: [PreconditioningDivergence](#)

Preconditioner is ill-conditioned.

__str__()

Return str(self).

class fipy.solvers.convergence.**IteratingConvergence**(*solver, iterations, residual, criterion, actual_code=None, **kwargs*)

Bases: [Convergence](#)

Solve still in progress.

__str__()

Return str(self).

```
class fipy.solvers.convergence.IterationConvergence(solver, iterations, residual, criterion,  
                                                    actual_code=None, **kwargs)
```

Bases: [Convergence](#)

Requested iterations complete (and no residual calculated).

```
__str__()  
    Return str(self).
```

```
class fipy.solvers.convergence.IterationDivergence(solver, iterations, residual, criterion,  
                                                    actual_code=None, **kwargs)
```

Bases: [Divergence](#)

Exceeded maximum iterations.

```
__str__()  
    Return str(self).
```

```
class fipy.solvers.convergence.LossOfAccuracyConvergence(solver, iterations, residual, criterion,  
                                                         actual_code=None, **kwargs)
```

Bases: [Convergence](#)

Numerical loss of precision occurred.

```
__str__()  
    Return str(self).
```

```
class fipy.solvers.convergence.NullDivergence(solver, iterations, residual, criterion,  
                                               actual_code=None, **kwargs)
```

Bases: [Divergence](#)

Breakdown when solving the Hessenberg system within GMRES.

```
__str__()  
    Return str(self).
```

```
class fipy.solvers.convergence.OutOfRangeDivergence(solver, iterations, residual, criterion,  
                                                    actual_code=None, **kwargs)
```

Bases: [Divergence](#)

A value became too small, too large, or invalid.

```
__str__()  
    Return str(self).
```

```
class fipy.solvers.convergence.PreconditioningDivergence(solver, iterations, residual, criterion,  
                                                         actual_code=None, **kwargs)
```

Bases: [Divergence](#)

A problem with the preconditioner.

```
__str__()  
    Return str(self).
```

```
class fipy.solvers.convergence.RHSZeroConvergence(solver, iterations, residual, criterion,  
                                                  actual_code=None, **kwargs)
```

Bases: [Convergence](#)

$\vec{b} = 0$, so exact solution is $\vec{x} = 0$.

```
__str__()
```

```
    Return str(self).
```

```
class fipy.solvers.convergence.RelativeToleranceConvergence(solver, iterations, residual, criterion,
                                                           actual_code=None, **kwargs)
```

```
    Bases: Convergence
```

```
    Relative tolerance satisfied
```

```
    ::
```

```
        residual < rtol * scale
```

```
__str__()
```

```
    Return str(self).
```

```
class fipy.solvers.convergence.StagnatedDivergence(solver, iterations, residual, criterion,
                                                    actual_code=None, **kwargs)
```

```
    Bases: Divergence
```

```
    The method stagnated.
```

```
__str__()
```

```
    Return str(self).
```

```
class fipy.solvers.convergence.ToleranceDivergence(solver, iterations, residual, criterion,
                                                    actual_code=None, **kwargs)
```

```
    Bases: Divergence
```

```
    Residual norm increased too much.
```

```
__str__()
```

```
    Return str(self).
```

23.4.2 fipy.solvers.petsc

Modules

<code>fipy.solvers.petsc.comms</code>
<code>fipy.solvers.petsc.dummySolver</code>
<code>fipy.solvers.petsc.linearBicgSolver</code>
<code>fipy.solvers.petsc.linearCGSSolver</code>
<code>fipy.solvers.petsc.linearCGSolver</code>
<code>fipy.solvers.petsc.linearGMRESSolver</code>
<code>fipy.solvers.petsc.linearLUSolver</code>
<code>fipy.solvers.petsc.petscConvergence</code>
<code>fipy.solvers.petsc.petscKrylovSolver</code>
<code>fipy.solvers.petsc.petscSolver</code>
<code>fipy.solvers.petsc.preconditioners</code>

fipy.solvers.petsc.comms

Modules

<code>fipy.solvers.petsc.comms. parallelPETScCommWrapper</code>
<code>fipy.solvers.petsc.comms.petscCommWrapper</code>
<code>fipy.solvers.petsc.comms. serialPETScCommWrapper</code>

fipy.solvers.petsc.comms.parallelPETScCommWrapper

Classes

<code>ParallelPETScCommWrapper()</code>	MPI Communicator wrapper

class `fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper`

Bases: `PETScCommWrapper`

MPI Communicator wrapper

Encapsulates capabilities needed for PETSc.

```
__getstate__()
```

Helper for pickle.

```
__repr__()
```

Return repr(self).

fiPy.solvers.petsc.comms.petscCommWrapper

Classes

<i>PETScCommWrapper</i> ([petsc4py_comm])	MPI Communicator wrapper

class fiPy.solvers.petsc.comms.petscCommWrapper.**PETScCommWrapper**(*petsc4py_comm*=*petsc4py.PETSc.COMM_WORLD*)

Bases: *CommWrapper*

MPI Communicator wrapper

Encapsulates capabilities needed for PETSc. Some capabilities are not parallel.

```
__getstate__()
```

Helper for pickle.

```
__repr__()
```

Return repr(self).

fiPy.solvers.petsc.comms.serialPETScCommWrapper

Classes

<i>SerialPETScCommWrapper</i> ()

class fiPy.solvers.petsc.comms.serialPETScCommWrapper.**SerialPETScCommWrapper**

Bases: *PETScCommWrapper*

```
__getstate__()
```

Helper for pickle.

```
__repr__()
```

Return repr(self).

fipy.solvers.petsc.dummySolver**Classes**

<i>DummySolver</i> ([tolerance, criterion, ...])	Solver that doesn't do anything.

```
class fipy.solvers.petsc.dummySolver.DummySolver(tolerance='default', criterion='default',
                                                    iterations='default', precon='default')
```

Bases: *PETScSolver*

Solver that doesn't do anything.

PETSc is intolerant of having zeros on the diagonal

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required residual tolerance.
- **criterion** ({'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}, *optional*) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.petsc.linearBicgSolver**Classes**

<i>LinearBicgSolver</i> ([tolerance, ...])	Interface to the biconjugate gradient solver (<i>BiCG</i>) in <i>PETSc</i> .

```
class fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver(tolerance='default',
                                                         absolute_tolerance=None,
                                                         divergence_tolerance=None,
                                                         criterion='default', iterations='default',
                                                         precon='default')
```

Bases: [PETScKrylovSolver](#)

Interface to the biconjugate gradient solver (*BiCG*) in *PETSc*.

Parameters

- **tolerance** (`float`) – Required relative error tolerance.
- **absolute_tolerance** (`float`) – Required absolute error tolerance.
- **divergence_tolerance** (`float`) – Required divergence error tolerance.
- **criterion** (`{'default', 'unscaled', 'RHS', 'matrix', 'initial', 'preconditioned', 'natural', 'legacy'}`) – Interpretation of tolerance. See [Convergence](#) for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** ([PETScPreconditioner](#) or `str`, *optional*) – Preconditioner to apply to the matrix. A value of `None` means to perform an unpreconditioned solve. While, for consistency with other solver suites, *FiPy* does implement some preconditioner objects for *PETSc*, you can also simply pass one of the `PCType` strings. (default: [DefaultPreconditioner](#)).

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of [DefaultPreconditioner](#)

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

[fipy.solvers.petsc.linearCGSSolver](#)

Classes

[LinearCGSSolver](#)([tolerance, ...])

Interface to the conjugate gradient squared (*CGS*) solver in *PETSc*.

```
class fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver(tolerance='default',
                                                         absolute_tolerance=None,
                                                         divergence_tolerance=None,
                                                         criterion='default', iterations='default',
                                                         precon='default')
```


Bases: [PETScKrylovSolver](#)

Interface to the conjugate gradient squared (*CGS*) solver in [PETSc](#).

Parameters

- **tolerance** (`float`) – Required relative error tolerance.
- **absolute_tolerance** (`float`) – Required absolute error tolerance.
- **divergence_tolerance** (`float`) – Required divergence error tolerance.
- **criterion** (`{'default', 'unscaled', 'RHS', 'matrix', 'initial', 'preconditioned', 'natural', 'legacy'}`) – Interpretation of tolerance. See [Convergence](#) for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** ([PETScPreconditioner](#) or `str`, *optional*) – Preconditioner to apply to the matrix. A value of `None` means to perform an unpreconditioned solve. While, for consistency with other solver suites, *FiPy* does implement some preconditioner objects for PETSc, you can also simply pass one of the `PCType` strings. (default: [DefaultPreconditioner](#)).

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of [DefaultPreconditioner](#)

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

`fipy.solvers.petsc.linearCGSolver`

Classes

LinearCGSolver ([tolerance, ...])	Interface to the conjugate gradient (<i>CG</i>) solver in PETSc .
LinearPCGSolver	alias of LinearCGSolver

```
class fipy.solvers.petsc.linearCGSolver.LinearCGSolver(tolerance='default',
                                                         absolute_tolerance=None,
                                                         divergence_tolerance=None,
                                                         criterion='default', iterations='default',
                                                         precon='default')
```

Bases: [PETScKrylovSolver](#)

Interface to the conjugate gradient (*CG*) solver in [PETSc](#).

Parameters

- **tolerance** (`float`) – Required relative error tolerance.
- **absolute_tolerance** (`float`) – Required absolute error tolerance.
- **divergence_tolerance** (`float`) – Required divergence error tolerance.
- **criterion** (`{'default', 'unscaled', 'RHS', 'matrix', 'initial', 'preconditioned', 'natural', 'legacy'}`) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** (*PETScPreconditioner* or `str`, optional) – Preconditioner to apply to the matrix. A value of `None` means to perform an unpreconditioned solve. While, for consistency with other solver suites, *FiPy* does implement some preconditioner objects for PETSc, you can also simply pass one of the `PCType` strings. (default: *DefaultPreconditioner*).

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of *DefaultPreconditioner*

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

`fipy.solvers.petsc.linearCGSolver.LinearPCGSolver`

alias of *LinearCGSolver*

`fipy.solvers.petsc.linearGMRESSolver`

Classes

LinearGMRESSolver([tolerance, ...])

Interface to the generalized minimal residual (*GMRES*) solver in *PETSc*.

```
class fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver(tolerance='default',
                                                             absolute_tolerance=None,
                                                             divergence_tolerance=None,
                                                             criterion='default',
                                                             iterations='default',
                                                             precon='default')
```

Bases: *PETScKrylovSolver*

Interface to the generalized minimal residual (*GMRES*) solver in *PETSc*.

Parameters

- **tolerance** (`float`) – Required relative error tolerance.
- **absolute_tolerance** (`float`) – Required absolute error tolerance.
- **divergence_tolerance** (`float`) – Required divergence error tolerance.
- **criterion** (`{'default', 'unscaled', 'RHS', 'matrix', 'initial', 'preconditioned', 'natural', 'legacy'}`) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** (*PETScPreconditioner* or `str`, optional) – Preconditioner to apply to the matrix. A value of `None` means to perform an unpreconditioned solve. While, for consistency with other solver suites, *FiPy* does implement some preconditioner objects for PETSc, you can also simply pass one of the `PCType` strings. (default: *DefaultPreconditioner*).

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of *DefaultPreconditioner*

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.petsc.linearLUSolver**Classes**

<i>LinearLUSolver</i> ([tolerance, criterion, ...])	Interface to the <i>LU</i> preconditioner in <i>PETSc</i> .

class fipy.solvers.petsc.linearLUSolver.LinearLUSolver(*tolerance='default', criterion='default', iterations=10, precon=None*)

Bases: *PETScSolver*

Interface to the *LU* preconditioner in *PETSc*.

A direct solve is performed.

Parameters

- **tolerance** (`float`) – Required error tolerance.
- **criterion** (`{'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy'}`) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.

- **precon** – *ignored*

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.petsc.petscConvergence

Classes

<code>KSP_AbsoluteToleranceConvergence(solver, ...)</code>	Residual 2-norm less than abstol
<code>KSP_BreakdownBICGDivergence(solver, ..., ...)</code>	Initial residual is orthogonal to preconditioned initial residual.
<code>KSP_BreakdownDivergence(solver, iterations, ...)</code>	Generic breakdown in method.
<code>KSP_HappyBreakdownConvergence(solver, ..., ...)</code>	
<code>KSP_IndefiniteMatrixDivergence(solver, ...)</code>	
<code>KSP_IndefinitePreconditionerDivergence(...)</code>	It appears the preconditioner is indefinite (has both positive and negative eigenvalues) and this Krylov method (<i>KSPCG</i>) requires it to be positive definite.
<code>KSP_IteratingConvergence(solver, iterations, ...)</code>	This flag is returned if you call <i>KSPGetConvergedReason()</i> while the <i>KSPSolve()</i> is still running.
<code>KSP_IterationConvergence(solver, iterations, ...)</code>	Used by the <i>KSPPREONLY</i> solver after the single iteration of the preconditioner is applied.
<code>KSP_IterationDivergence(solver, iterations, ...)</code>	Ran out of iterations before any convergence criteria was reached
<code>KSP_NanOrInfDivergence(solver, iterations, ...)</code>	Residual norm became Not-a-number or Inf likely due to 0/0.
<code>KSP_NonSymmetricDivergence(solver, ..., ...)</code>	It appears the operator or preconditioner is not symmetric and this Krylov method (<i>KSPCG</i> , <i>KSPMINRES</i> , <i>KSPCR</i>) requires symmetry
<code>KSP_NormalAbsoluteToleranceConvergence(...)</code>	
<code>KSP_NormalRelativeToleranceConvergence(...)</code>	
<code>KSP_NullDivergence(solver, iterations, ...)</code>	
<code>KSP_PreconditioningDivergence(solver, ..., ...)</code>	It was not possible to build or use the requested preconditioner.
<code>KSP_RelativeToleranceConvergence(solver, ...)</code>	Residual 2-norm decreased by a factor of <i>rtol</i> , from 2-norm of right hand side.
<code>KSP_ToleranceDivergence(solver, iterations, ...)</code>	Residual norm increased by a factor of <i>divtol</i> .

```
class fipy.solvers.petsc.petscConvergence.KSP_AbsoluteToleranceConvergence(solver, iterations,
                                                                            residual, criterion,
                                                                            actual_code=None,
                                                                            **kwargs)
```

Bases: *AbsoluteToleranceConvergence*

Residual 2-norm less than abstol

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_BreakdownBICGDivergence(solver, iterations,
                                                                           residual, criterion,
                                                                           actual_code=None,
                                                                           **kwargs)
```

Bases: *KSP_BreakdownDivergence*

Initial residual is orthogonal to preconditioned initial residual. Try a different preconditioner, or a different initial Level.)

`__str__()`

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_BreakdownDivergence(solver, iterations, residual,
                                                                    criterion, actual_code=None,
                                                                    **kwargs)
```

Bases: *BreakdownDivergence*

Generic breakdown in method.

`__str__()`

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_HappyBreakdownConvergence(solver, iterations,
                                                                            residual, criterion,
                                                                            actual_code=None,
                                                                            **kwargs)
```

Bases: *HappyBreakdownConvergence*

`__str__()`

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_IndefiniteMatrixDivergence(solver, iterations,
                                                                            residual, criterion,
                                                                            actual_code=None,
                                                                            **kwargs)
```

Bases: *IllConditionedDivergence*

`__str__()`

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_IndefinitePreconditionerDivergence(solver,
                                                                                    iterations,
                                                                                    residual,
                                                                                    criterion,
                                                                                    ac-
                                                                                    tual_code=None,
                                                                                    **kwargs)
```

Bases: *IllConditionedPreconditionerDivergence*

It appears the preconditioner is indefinite (has both positive and negative eigenvalues) and this Krylov method (*KSPCG*) requires it to be positive definite.

`__str__()`

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_IteratingConvergence(solver, iterations, residual,
                                                                      criterion,
                                                                      actual_code=None,
                                                                      **kwargs)
```

Bases: *IteratingConvergence*

This flag is returned if you call *KSPGetConvergedReason()* while the *KSPSolve()* is still running.

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_IterationConvergence(solver, iterations, residual,
                                                                    criterion,
                                                                    actual_code=None,
                                                                    **kwargs)
```

Bases: [*IterationConvergence*](#)

Used by the *KSPPREONLY* solver after the single iteration of the preconditioner is applied. Also used when the *KSPConvergedSkip()* convergence test routine is set in *KSP*.

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_IterationDivergence(solver, iterations, residual,
                                                                    criterion, actual_code=None,
                                                                    **kwargs)
```

Bases: [*IterationDivergence*](#)

Ran out of iterations before any convergence criteria was reached

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_NanOrInfDivergence(solver, iterations, residual,
                                                                    criterion, actual_code=None,
                                                                    **kwargs)
```

Bases: [*OutOfRangeDivergence*](#)

Residual norm became Not-a-number or Inf likely due to 0/0.

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_NonSymmetricDivergence(solver, iterations,
                                                                        residual, criterion,
                                                                        actual_code=None,
                                                                        **kwargs)
```

Bases: [*IllConditionedDivergence*](#)

It appears the operator or preconditioner is not symmetric and this Krylov method (*KSPCG*, *KSPMINRES*, *KSPCR*) requires symmetry

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_NormalAbsoluteToleranceConvergence(solver,
                                                                                    iterations,
                                                                                    residual,
                                                                                    criterion,
                                                                                    ac-
                                                                                    tual_code=None,
                                                                                    **kwargs)
```

Bases: [*KSP_AbsoluteToleranceConvergence*](#)

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_NormalRelativeToleranceConvergence(solver,
                                                                                   iterations,
                                                                                   residual,
                                                                                   criterion,
                                                                                   ac-
                                                                                   tual_code=None,
                                                                                   **kwargs)
```

Bases: *KSP_RelativeToleranceConvergence*

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_NullDivergence(solver, iterations, residual, criterion,
                                                             actual_code=None, **kwargs)
```

Bases: *NullDivergence*

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_PreconditioningDivergence(solver, iterations,
                                                                           residual, criterion,
                                                                           actual_code=None,
                                                                           **kwargs)
```

Bases: *PreconditioningDivergence*

It was not possible to build or use the requested preconditioner. This is usually due to a zero pivot in a factorization. It can also result from a failure in a subpreconditioner inside a nested preconditioner such as *PCFIELD-SPLIT*.

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_RelativeToleranceConvergence(solver, iterations,
                                                                              residual, criterion,
                                                                              ac-
                                                                              tual_code=None,
                                                                              **kwargs)
```

Bases: *RelativeToleranceConvergence*

Residual 2-norm decreased by a factor of *rtol*, from 2-norm of right hand side.

```
__str__()
```

Return str(self).

```
class fipy.solvers.petsc.petscConvergence.KSP_ToleranceDivergence(solver, iterations, residual,
                                                                    criterion, actual_code=None,
                                                                    **kwargs)
```

Bases: *ToleranceDivergence*

Residual norm increased by a factor of *divtol*.

```
__str__()
```

Return str(self).

fipy.solvers.petsc.petscKrylovSolver

Classes

PETScKrylovSolver([tolerance, ...])

Attention:

This class is abstract, always create one of its subclasses.

```
class fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver(tolerance='default',
                                                             absolute_tolerance=None,
                                                             divergence_tolerance=None,
                                                             criterion='default',
                                                             iterations='default',
                                                             precon='default')
```

Bases: *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** (*float*) – Required relative error tolerance.
- **absolute_tolerance** (*float*) – Required absolute error tolerance.
- **divergence_tolerance** (*float*) – Required divergence error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'preconditioned', 'natural', 'legacy'}) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*PETScPreconditioner* or *str*, *optional*) – Preconditioner to apply to the matrix. A value of None means to perform an unpreconditioned solve. While, for consistency with other solver suites, *FiPy* does implement some preconditioner objects for PETSc, you can also simply pass one of the *PCType* strings. (default: *DefaultPreconditioner*).

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of *DefaultPreconditioner*

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.petsc.petscSolver

Classes

PETScSolver([tolerance, criterion, ...])

Attention:

This class is abstract. Always create one of its subclasses.

```
class fipy.solvers.petsc.petscSolver.PETScSolver(tolerance='default', criterion='default',
                                              iterations='default', precon='default')
```

Bases: *Solver*

Attention: This class is abstract. Always create one of its subclasses.

Create a *Solver* object.

Parameters

- **tolerance** (`float`) – Required residual tolerance.
- **criterion** (`{'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}`, *optional*) – Interpretation of tolerance. See [Convergence](#) for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** ([Preconditioner](#)) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fiPy.solvers.petsc.preconditioners

class fiPy.solvers.petsc.preconditioners.DefaultPreconditioner

Bases: [PETScPreconditioner](#)

Apply PETSc's default preconditioning to [PETScSolver](#).

"The default preconditioner for sparse matrices is *PCILU* or *PCICC* with 0 fill on one process and block Jacobi (*PCBJACOBI*) with *PCILU* or *PCICC* in parallel."¹

class fiPy.solvers.petsc.preconditioners.HYPREPreconditioner

Bases: [PETScPreconditioner](#)

HYPRE preconditioner for [PETScSolver](#).

class fiPy.solvers.petsc.preconditioners.ICPreconditioner

Bases: [PETScPreconditioner](#)

Incomplete Choleski preconditioner for [PETScSolver](#).

class fiPy.solvers.petsc.preconditioners.ILUPreconditioner

Bases: [PETScPreconditioner](#)

Incomplete LU preconditioner for [PETScSolver](#).

class fiPy.solvers.petsc.preconditioners.JacobiPreconditioner

Bases: [PETScPreconditioner](#)

Jacobi preconditioner for [PETScSolver](#).

¹ <https://petsc.org/main/manualpages/PC/PCCreate/#note>

class `fipy.solvers.petsc.preconditioners.LUPreconditioner`

Bases: *PETScPreconditioner*

LU preconditioner for *PETScSolver*.

class `fipy.solvers.petsc.preconditioners.SSORPreconditioner`

Bases: *PETScPreconditioner*

SSOR preconditioner for *PETScSolver*.

Modules

fipy.solvers.petsc.preconditioners.defaultPreconditioner
fipy.solvers.petsc.preconditioners.hyprePreconditioner
fipy.solvers.petsc.preconditioners.icPreconditioner
fipy.solvers.petsc.preconditioners.iluPreconditioner
fipy.solvers.petsc.preconditioners.jacobiPreconditioner
fipy.solvers.petsc.preconditioners.luPreconditioner
fipy.solvers.petsc.preconditioners.petscPreconditioner
fipy.solvers.petsc.preconditioners.ssorPreconditioner

`fipy.solvers.petsc.preconditioners.defaultPreconditioner`

Classes

<i>DefaultPreconditioner()</i>	Apply PETSc's default preconditioning to <i>PETScSolver</i> .

class `fipy.solvers.petsc.preconditioners.defaultPreconditioner.DefaultPreconditioner`

Bases: *PETScPreconditioner*

Apply PETSc's default preconditioning to *PETScSolver*.

“The default preconditioner for sparse matrices is *PCILU* or *PCICC* with 0 fill on one process and block Jacobi (*PCBJACOBI*) with *PCILU* or *PCICC* in parallel.”¹

¹ <https://petsc.org/main/manualpages/PC/PCCreate/#note>

fipy.solvers.petsc.preconditioners.hyPrePreconditioner**Classes**

<i>HYPREPreconditioner()</i>	HYPRE preconditioner for <i>PETScSolver</i> .

class fipy.solvers.petsc.preconditioners.hyPrePreconditioner.HYPREPreconditioner

Bases: *PETScPreconditioner*

HYPRE preconditioner for *PETScSolver*.

fipy.solvers.petsc.preconditioners.icPreconditioner**Classes**

<i>ICPreconditioner()</i>	Incomplete Choleski preconditioner for <i>PETScSolver</i> .

class fipy.solvers.petsc.preconditioners.icPreconditioner.ICPreconditioner

Bases: *PETScPreconditioner*

Incomplete Choleski preconditioner for *PETScSolver*.

fipy.solvers.petsc.preconditioners.iluPreconditioner**Classes**

<i>ILUPreconditioner()</i>	Incomplete LU preconditioner for <i>PETScSolver</i> .

class fipy.solvers.petsc.preconditioners.iluPreconditioner.ILUPreconditioner

Bases: *PETScPreconditioner*

Incomplete LU preconditioner for *PETScSolver*.

fipy.solvers.petsc.preconditioners.jacobiPreconditioner**Classes**

<i>JacobiPreconditioner()</i>	Jacobi preconditioner for <i>PETScSolver</i> .

class fipy.solvers.petsc.preconditioners.jacobiPreconditioner.JacobiPreconditioner

Bases: *PETScPreconditioner*

Jacobi preconditioner for *PETScSolver*.

`fipy.solvers.petsc.preconditioners.luPreconditioner`

Classes

<code>LUPreconditioner()</code>	LU preconditioner for <i>PETScSolver</i> .

class `fipy.solvers.petsc.preconditioners.luPreconditioner.LUPreconditioner`

Bases: *PETScPreconditioner*

LU preconditioner for *PETScSolver*.

`fipy.solvers.petsc.preconditioners.petscPreconditioner`

Classes

<code>PETScPreconditioner()</code>	Base class preconditioners of for <i>PETScSolver</i> .

class `fipy.solvers.petsc.preconditioners.petscPreconditioner.PETScPreconditioner`

Bases: *SolverModifyingPreconditioner*

Base class preconditioners of for *PETScSolver*.

Attention: This class is abstract. Always create one of its subclasses.

`fipy.solvers.petsc.preconditioners.ssorPreconditioner`

Classes

<code>SSORPreconditioner()</code>	SSOR preconditioner for <i>PETScSolver</i> .

class `fipy.solvers.petsc.preconditioners.ssorPreconditioner.SSORPreconditioner`

Bases: *PETScPreconditioner*

SSOR preconditioner for *PETScSolver*.

23.4.3 `fipy.solvers.preconditioner`

Classes

<code>MatrixModifyingPreconditioner()</code>	Base class for preconditioners that modify a <code>_SparseMatrix</code> .
<code>Preconditioner()</code>	Base class for solver preconditioners.
<code>SolverModifyingPreconditioner()</code>	Base class for preconditioners that modify a <i>Solver</i> .

class `fipy.solvers.preconditioner.MatrixModifyingPreconditioner`

Bases: *Preconditioner*

Base class for preconditioners that modify a `_SparseMatrix`.

class `fipy.solvers.preconditioner.Preconditioner`

Bases: *object*

Base class for solver preconditioners.

Attention: This class is abstract. Always create one of its subclasses.

class `fipy.solvers.preconditioner.SolverModifyingPreconditioner`

Bases: *Preconditioner*

Base class for preconditioners that modify a *Solver*.

23.4.4 fipy.solvers.pyAMG

Modules

fipy.solvers.pyAMG.preconditioners

fipy.solvers.pyAMG.preconditioners

Modules

*fipy.solvers.pyAMG.preconditioners.
smoothedAggregationPreconditioner*

fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner

Classes

<i>SmoothedAggregationPreconditioner()</i>	Preconditioner based on <i>PyAMG</i> <i>smoothed_aggregation_solver</i> for <i>ScipySolver</i> .

class `fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner.
SmoothedAggregationPreconditioner`

Bases: *ScipyPreconditioner*

Preconditioner based on *PyAMG* *smoothed_aggregation_solver* for *ScipySolver*.

23.4.5 fipy.solvers.pyamgx

Modules

*fipy.solvers.pyamgx.aggregationAMGSolver**fipy.solvers.pyamgx.classicalAMGSolver**fipy.solvers.pyamgx.linearBiCGStabSolver**fipy.solvers.pyamgx.linearCGSolver**fipy.solvers.pyamgx.linearFGMRESSolver**fipy.solvers.pyamgx.linearGMRESSolver**fipy.solvers.pyamgx.linearLUSolver**fipy.solvers.pyamgx.preconditioners**fipy.solvers.pyamgx.pyAMGXSolver**fipy.solvers.pyamgx.pyamgxConvergence**fipy.solvers.pyamgx.smoother*

fipy.solvers.pyamgx.aggregationAMGSolver

Classes

<i>AggregationAMGSolver</i> ([tolerance, criterion, ...])	Interface to the aggregation algebraic multigrid (<i>AMG</i>) solver in <i>pyamgx</i> .

```
class fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver(tolerance='default',
                                                                    criterion='default',
                                                                    iterations='default',
                                                                    precon='default',
                                                                    smoother='default',
                                                                    **kwargs)
```

Bases: *PyAMGXSolver*

Interface to the aggregation algebraic multigrid (*AMG*) solver in *pyamgx*.

Uses a *BlockJacobiSmoother* smoother by default.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy'}) – Interpretation of tolerance. See *Convergence* for more information.

- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** (`PyAMGXPreconditioner`, *optional*) –
- **smoother** (`Smoother`, *optional*) –
- ****kwargs** – Other AMGX solver options

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_SMOOTHER = <fipy.solvers.pyamgx.smoothers.Smoother object>

Default smoother to apply to the ???

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.pyamgx.classicalAMGSolver

Classes

<code>ClassicalAMGSolver</code> ([tolerance, criterion, ...])	Interface to the classical algebraic multigrid (<i>AMG</i>) solver in <i>pyamgx</i> .

```
class fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver(tolerance='default',
                                                                  criterion='default',
                                                                  iterations='default',
                                                                  precon='default',
                                                                  smoother='default', **kwargs)
```

Bases: `PyAMGXSolver`

Interface to the classical algebraic multigrid (*AMG*) solver in *pyamgx*.

Uses a BlockJacobiSmoother smoother by default.

Parameters

- **tolerance** (`float`) – Required error tolerance.
- **criterion** (`{'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy'}`) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** (`PyAMGXPreconditioner`, *optional*) –
- **smoother** (`Smoother`, *optional*) –

- ****kwargs** – Other AMGX solver options

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_SMOOTHER = <fipy.solvers.pyamgx.smoothers.Smoother object>

Default smoother to apply to the ???

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.pyamgx.linearBiCGStabSolver

Classes

<code>LinearBiCGStabSolver([tolerance, criterion, ...])</code>	Interface to the Biconjugate Gradient (Stabilized) (<i>BiCGSTAB</i>) solver in <i>pyamgx</i> .

```
class fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver(tolerance='default',
                                                                    criterion='default',
                                                                    iterations='default',
                                                                    precon='default',
                                                                    smoother='default',
                                                                    **kwargs)
```

Bases: [PyAMGXSolver](#)

Interface to the Biconjugate Gradient (Stabilized) (*BiCGSTAB*) solver in *pyamgx*.

Uses a [JacobiPreconditioner](#) by default.

Parameters

- **tolerance** (`float`) – Required error tolerance.
- **criterion** (`{'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy'}`) – Interpretation of tolerance. See [Convergence](#) for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** ([PyAMGXPreconditioner](#), *optional*) –
- **smoother** ([Smoother](#), *optional*) –
- ****kwargs** – Other AMGX solver options

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of [JacobiPreconditioner](#)

DEFAULT_SMOOTHER = None

Default smoother to apply to the ???

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.pyamgx.linearCGSolver

Classes

LinearCGSolver ([tolerance, criterion, ...])	Interface to the conjugate gradient (<i>CG</i>) solver in pyamgx .
LinearPCGSolver	alias of LinearCGSolver

```
class fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver(tolerance='default', criterion='default',
                                                         iterations='default', precon='default',
                                                         smoother='default', **kwargs)
```

Bases: [PyAMGXSolver](#)

Interface to the conjugate gradient (*CG*) solver in [pyamgx](#).

Uses [JacobiPreconditioner](#) by default.

Parameters

- **tolerance** ([float](#)) – Required error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy'}) – Interpretation of **tolerance**. See [Convergence](#) for more information.
- **iterations** ([int](#)) – Maximum number of iterative steps to perform.
- **precon** ([PyAMGXPreconditioner](#), optional) –
- **smoother** ([Smoother](#), optional) –
- ****kwargs** – Other AMGX solver options

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of [*JacobiPreconditioner*](#)

DEFAULT_SMOOTHER = None

Default smoother to apply to the ???

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

`fipy.solvers.pyamgx.linearCGSolver.LinearPCGSolver`

alias of [*LinearCGSolver*](#)

fipy.solvers.pyamgx.linearFGMRESSolver**Classes**

[*LinearFGMRESSolver*](#)([tolerance, criterion, ...])

Interface to the Flexible Generalized Minimum RESidual (*FGMRES*) solver in [*pyamgx*](#).

```
class fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver(tolerance='default',
                                                                  criterion='default',
                                                                  iterations='default',
                                                                  precon='default',
                                                                  smoother='default', **kwargs)
```

Bases: [*PyAMGXSolver*](#)

Interface to the Flexible Generalized Minimum RESidual (*FGMRES*) solver in [*pyamgx*](#).

Uses a [*JacobiPreconditioner*](#) by default.

Parameters

- **tolerance** ([*float*](#)) – Required error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy'}) – Interpretation of tolerance. See [*Convergence*](#) for more information.
- **iterations** ([*int*](#)) – Maximum number of iterative steps to perform.
- **precon** ([*PyAMGXPreconditioner*](#), optional) –
- **smoother** ([*Smoother*](#), optional) –
- ****kwargs** – Other AMGX solver options

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONERalias of [*JacobiPreconditioner*](#)**DEFAULT_SMOOTHER = None**

Default smoother to apply to the ???

DEFAULT_TOLERANCE = 1e-05Default tolerance for linear solves unless *criterion*="legacy"**LEGACY_TOLERANCE = 1e-10**Default tolerance for linear solves if *criterion*="legacy"**__repr__()**

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.pyamgx.linearGMRESSolver**Classes**[*LinearGMRESSolver*](#)([tolerance, criterion, ...])Interface to the Generalized Minimum RESidual (*GM-RES*) solver in *pyamgx*.

```
class fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver(tolerance='default',
                                                                criterion='default',
                                                                iterations='default',
                                                                precon='default',
                                                                smoother='default', **kwargs)
```

Bases: [*PyAMGXSolver*](#)Interface to the Generalized Minimum RESidual (*GMRES*) solver in *pyamgx*.Uses a [*JacobiPreconditioner*](#) by default.**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy'}) – Interpretation of tolerance. See [*Convergence*](#) for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** ([*PyAMGXPreconditioner*](#), *optional*) –
- **smoother** ([*Smoother*](#), *optional*) –
- ****kwargs** – Other AMG solver options

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONERalias of [*JacobiPreconditioner*](#)

DEFAULT_SMOOTHER = None

Default smoother to apply to the ???

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.pyamgx.linearLUSolver

```
class fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver(tolerance='default',
                                                         absolute_tolerance=0.0,
                                                         criterion='default', iterations='default',
                                                         precon='default')
```

Bases: [ScipySolver](#)

Interface to *LU*-factorization in *SciPy*.

Create a *Solver* object.

Parameters

- **tolerance** ([float](#)) – Required relative error tolerance.
- **absolute_tolerance** ([float](#)) – Required absolute error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy', }) – Interpretation of tolerance. See [Convergence](#) for more information.
- **iterations** ([int](#)) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.pyamgx.preconditioners**Classes**

<i>AMGPreconditioner</i>(**kwargs)	Adaptive Multigrid preconditioner for <i>PyAMGXSolver</i> .
<i>AggregationAMGPreconditioner</i>()	Aggregation Adaptive Multigrid preconditioner for <i>PyAMGXSolver</i> .
<i>BiCGStabPreconditioner</i>(**kwargs)	Biconjugate Gradient Stabilized preconditioner for <i>PyAMGXSolver</i> .
<i>CGPreconditioner</i>(**kwargs)	Conjugate Gradient preconditioner for <i>PyAMGXSolver</i> .
<i>DILUPreconditioner</i>(**kwargs)	Diagonal Incomplete LU preconditioner for <i>PyAMGXSolver</i> .
<i>FGMRESPreconditioner</i>(**kwargs)	Flexible Generalized Minimum Residual preconditioner for <i>PyAMGXSolver</i> .
<i>GaussSeidelPreconditioner</i>(**kwargs)	Gauss-Seidel preconditioner for <i>PyAMGXSolver</i> .
<i>ILUPreconditioner</i>(**kwargs)	Incomplete LU preconditioner for <i>PyAMGXSolver</i> .
<i>JacobiPreconditioner</i>(**kwargs)	Block Jacobi preconditioner for <i>PyAMGXSolver</i> .
<i>PolynomialPreconditioner</i>(**kwargs)	Polynomial preconditioner for <i>PyAMGXSolver</i> .
<i>PyAMGXPreconditioner</i>(**kwargs)	Interface to <i>pyamgx</i> preconditioner configuration for <i>PyAMGXSolver</i> .

class fipy.solvers.pyamgx.preconditioners.**AMGPreconditioner**(**kwargs)

Bases: [*PyAMGXPreconditioner*](#)

Adaptive Multigrid preconditioner for [*PyAMGXSolver*](#).

Parameters

****kwargs** (*dict*, *optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

class fipy.solvers.pyamgx.preconditioners.**AggregationAMGPreconditioner**

Bases: [*AMGPreconditioner*](#)

Aggregation Adaptive Multigrid preconditioner for [*PyAMGXSolver*](#).

Parameters

****kwargs** (*dict*, *optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

class fipy.solvers.pyamgx.preconditioners.**BiCGStabPreconditioner**(**kwargs)

Bases: [*PyAMGXPreconditioner*](#)

Biconjugate Gradient Stabilized preconditioner for [*PyAMGXSolver*](#).

Parameters

****kwargs** (*dict*, *optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

class fipy.solvers.pyamgx.preconditioners.**CGPreconditioner**(**kwargs)

Bases: [*PyAMGXPreconditioner*](#)

Conjugate Gradient preconditioner for [*PyAMGXSolver*](#).

Parameters

****kwargs** (*dict*, *optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

```
class fipy.solvers.pyamgx.preconditioners.DILUPreconditioner(**kwargs)
```

Bases: [PyAMGXPreconditioner](#)

Diagonal Incomplete LU preconditioner for [PyAMGXSolver](#).

Parameters

****kwargs** (*dict, optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

```
class fipy.solvers.pyamgx.preconditioners.FGMRESPreconditioner(**kwargs)
```

Bases: [PyAMGXPreconditioner](#)

Flexible Generalized Minimum Residual preconditioner for [PyAMGXSolver](#).

Parameters

****kwargs** (*dict, optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

```
class fipy.solvers.pyamgx.preconditioners.GaussSeidelPreconditioner(**kwargs)
```

Bases: [PyAMGXPreconditioner](#)

Gauss-Seidel preconditioner for [PyAMGXSolver](#).

Parameters

****kwargs** (*dict, optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

```
class fipy.solvers.pyamgx.preconditioners.ILUPreconditioner(**kwargs)
```

Bases: [PyAMGXPreconditioner](#)

Incomplete LU preconditioner for [PyAMGXSolver](#).

Parameters

****kwargs** (*dict, optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

```
class fipy.solvers.pyamgx.preconditioners.JacobiPreconditioner(**kwargs)
```

Bases: [PyAMGXPreconditioner](#)

Block Jacobi preconditioner for [PyAMGXSolver](#).

Parameters

****kwargs** (*dict, optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

```
class fipy.solvers.pyamgx.preconditioners.PolynomialPreconditioner(**kwargs)
```

Bases: [PyAMGXPreconditioner](#)

Polynomial preconditioner for [PyAMGXSolver](#).

Parameters

****kwargs** (*dict, optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

```
class fipy.solvers.pyamgx.preconditioners.PyAMGXPreconditioner(**kwargs)
```

Bases: [SolverModifyingPreconditioner](#)

Interface to pyamgx preconditioner configuration for [PyAMGXSolver](#).

Parameters

****kwargs** (*dict, optional*) – Extra arguments to preconditioner: refer to [preconditioner configuration](#) for information about possible arguments.

fipy.solvers.pyamgx.pyAMGXSolver**Classes***PyAMGXSolver*([tolerance, criterion, ...])**param tolerance**

Required error tolerance.

```
class fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver(tolerance='default', criterion='default',
                                                    iterations='default', precon='default',
                                                    smoother='default', **kwargs)
```

Bases: *Solver***Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy'}) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*PyAMGXPreconditioner*, optional) –
- **smoother** (*Smoother*, optional) –
- ****kwargs** – Other AMGX solver options

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_SMOOTHER = None

Default smoother to apply to the ???

DEFAULT_TOLERANCE = 1e-05Default tolerance for linear solves unless *criterion="legacy"***LEGACY_TOLERANCE = 1e-10**Default tolerance for linear solves if *criterion="legacy"***__repr__()**

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.pyamgx.pyamgxConvergence**Classes**

pyamgx_BreakdownDivergence(solver, ...[, ...])*pyamgx_Convergence*(solver, iterations, ...)*pyamgx_IterationDivergence*(solver, ...[, ...])

```
class fipy.solvers.pyamgx.pyamgxConvergence.pyamgx_BreakdownDivergence(solver, iterations,
                                                                           residual, criterion,
                                                                           actual_code=None,
                                                                           **kwargs)
```

Bases: *BreakdownDivergence*

```
__str__()
    Return str(self).
```

```
class fipy.solvers.pyamgx.pyamgxConvergence.pyamgx_Convergence(solver, iterations, residual,
                                                                  criterion, actual_code=None,
                                                                  **kwargs)
```

Bases: *Convergence*

```
__str__()
    Return str(self).
```

```
class fipy.solvers.pyamgx.pyamgxConvergence.pyamgx_IterationDivergence(solver, iterations,
                                                                           residual, criterion,
                                                                           actual_code=None,
                                                                           **kwargs)
```

Bases: *IterationDivergence*

```
__str__()
    Return str(self).
```

fipy.solvers.pyamgx.smoothers**Classes**

Smoother(smoother_type) Interface to *pyamgx smoother configuration*.

```
class fipy.solvers.pyamgx.smoothers.Smoother(smoother_type)
```

Bases: *object*

Interface to *pyamgx smoother configuration*.

23.4.6 fipy.solvers.scipy

Modules

`fipy.solvers.scipy.linearBicgstabSolver`

`fipy.solvers.scipy.linearCGSSolver`

`fipy.solvers.scipy.linearCGSolver`

`fipy.solvers.scipy.linearGMRESSolver`

`fipy.solvers.scipy.linearLUSolver`

`fipy.solvers.scipy.preconditioners`

`fipy.solvers.scipy.scipyConvergence`

`fipy.solvers.scipy.scipyKrylovSolver`

`fipy.solvers.scipy.scipySolver`

fipy.solvers.scipy.linearBicgstabSolver

Classes

<code>LinearBicgstabSolver([tolerance, ...])</code>	Interface to the Biconjugate Gradient (Stabilized) (<i>BiCGSTAB</i>) solver in <i>SciPy</i> .

```
class fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver(tolerance='default',
                                                                    absolute_tolerance=0.0,
                                                                    criterion='default',
                                                                    iterations='default',
                                                                    precon='default')
```

Bases: `ScipyKrylovSolver`

Interface to the Biconjugate Gradient (Stabilized) (*BiCGSTAB*) solver in *SciPy*.

No preconditioning by default.

Create a *Solver* object.

Parameters

- **tolerance** (`float`) – Required relative error tolerance.
- **absolute_tolerance** (`float`) – Required absolute error tolerance.
- **criterion** (`{'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy', }`) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.

- **precon** – Preconditioner to use.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.scipy.linearCGSSolver

Classes

LinearCGSSolver([tolerance, ...])

Interface to the conjugate gradient squared (*CGS*) solver in *SciPy*.

```
class fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver(tolerance='default',
                                                         absolute_tolerance=0.0,
                                                         criterion='default', iterations='default',
                                                         precon='default')
```

Bases: *ScipyKrylovSolver*

Interface to the conjugate gradient squared (*CGS*) solver in *SciPy*.

No preconditioning by default.

Create a *Solver* object.

Parameters

- **tolerance** (**float**) – Required relative error tolerance.
- **absolute_tolerance** (**float**) – Required absolute error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy', }) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (**int**) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.scipy.linearCGSolver

Classes

<i>LinearCGSolver</i> ([tolerance, ...])	Interface to the conjugate gradient (<i>CG</i>) solver in <i>SciPy</i> .
<i>LinearPCGSolver</i>	alias of <i>LinearCGSolver</i>

class fipy.solvers.scipy.linearCGSolver.**LinearCGSolver**(*tolerance*='default', *absolute_tolerance*=0.0, *criterion*='default', *iterations*='default', *precon*='default')

Bases: [*ScipyKrylovSolver*](#)

Interface to the conjugate gradient (*CG*) solver in *SciPy*.

No preconditioning by default.

Create a *Solver* object.

Parameters

- **tolerance** ([float](#)) – Required relative error tolerance.
- **absolute_tolerance** ([float](#)) – Required absolute error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy', }) – Interpretation of tolerance. See [Convergence](#) for more information.
- **iterations** ([int](#)) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

`__repr__()`

Return `repr(self)`.

property `default_tolerance`

Default tolerance for linear solve

`fipy.solvers.scipy.linearCGSolver.LinearPCGSolver`

alias of [LinearCGSolver](#)

`fipy.solvers.scipy.linearGMRESSolver`

Classes

[LinearGMRESSolver](#)([tolerance, ...])

Interface to the Generalized Minimum RESidual (*GM-RES*) solver in *SciPy*.

```
class fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver(tolerance='default',
                                                             absolute_tolerance=0.0,
                                                             criterion='default',
                                                             iterations='default',
                                                             precon='default')
```

Bases: [ScipyKrylovSolver](#)

Interface to the Generalized Minimum RESidual (*GMRES*) solver in *SciPy*.

No preconditioning by default.

Create a *Solver* object.

Parameters

- **tolerance** (`float`) – Required relative error tolerance.
- **absolute_tolerance** (`float`) – Required absolute error tolerance.
- **criterion** (`{'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy', }`) – Interpretation of tolerance. See [Convergence](#) for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

`__repr__()`

Return `repr(self)`.

property default_tolerance

Default tolerance for linear solve

fipy.solvers.scipy.linearLUSolver

Classes

LinearLUSolver([tolerance, ...])

Interface to *LU*-factorization in *SciPy*.

```
class fipy.solvers.scipy.linearLUSolver.LinearLUSolver(tolerance='default', absolute_tolerance=0.0,
                                                    criterion='default', iterations='default',
                                                    precon='default')
```

Bases: *ScipySolver*

Interface to *LU*-factorization in *SciPy*.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required relative error tolerance.
- **absolute_tolerance** (*float*) – Required absolute error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy', }) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.scipy.preconditioners

class fipy.solvers.scipy.preconditioners.**ILUPreconditioner**

Bases: *ScipyPreconditioner*

Incomplete LU preconditioner for *ScipySolver*.

Wrapper class for `scipy.sparse.linalg.spilu()`. Adapted from <https://stackoverflow.com/q/46876951/2019542>.

class fipy.solvers.scipy.preconditioners.**JacobiPreconditioner**

Bases: *ScipyPreconditioner*

Jacobi preconditioner for *ScipySolver*.

Wrapper class for `scipy.sparse.linalg.spsolve()` with *matrix* diagonal. Adapted from <https://stackoverflow.com/q/46876951/2019542>.

Modules

```
fipy.solvers.scipy.preconditioners.
iluPreconditioner
fipy.solvers.scipy.preconditioners.
jacobiPreconditioner
fipy.solvers.scipy.preconditioners.
scipyPreconditioner
```

fipy.solvers.scipy.preconditioners.iluPreconditioner

Classes

<i>ILUPreconditioner()</i>	Incomplete LU preconditioner for <i>ScipySolver</i> .

class fipy.solvers.scipy.preconditioners.iluPreconditioner.**ILUPreconditioner**

Bases: *ScipyPreconditioner*

Incomplete LU preconditioner for *ScipySolver*.

Wrapper class for `scipy.sparse.linalg.spilu()`. Adapted from <https://stackoverflow.com/q/46876951/2019542>.

fipy.solvers.scipy.preconditioners.jacobiPreconditioner

Classes

<i>JacobiPreconditioner()</i>	Jacobi preconditioner for <i>ScipySolver</i> .

class fipy.solvers.scipy.preconditioners.jacobiPreconditioner.JacobiPreconditioner

Bases: *ScipyPreconditioner*

Jacobi preconditioner for *ScipySolver*.

Wrapper class for `scipy.sparse.linalg.spsolve()` with *matrix* diagonal. Adapted from <https://stackoverflow.com/q/46876951/2019542>.

fipy.solvers.scipy.preconditioners.scipyPreconditioner

Classes

ScipyPreconditioner()

Base class for preconditioners for *ScipySolver*.

class fipy.solvers.scipy.preconditioners.scipyPreconditioner.ScipyPreconditioner

Bases: *MatrixModifyingPreconditioner*

Base class for preconditioners for *ScipySolver*.

Attention: This class is abstract. Always create one of its subclasses.

fipy.solvers.scipy.scipyConvergence

Classes

SciPy_BreakdownDivergence(solver, ...[, ...])

SciPy_Convergence(solver, iterations, ...[, ...])

SciPy_IterationDivergence(solver, ...[, ...])

class fipy.solvers.scipy.scipyConvergence.SciPy_BreakdownDivergence(*solver*, *iterations*, *residual*,
criterion,
actual_code=None,
***kwargs*)

Bases: *BreakdownDivergence*

__str__()

Return str(self).

class fipy.solvers.scipy.scipyConvergence.SciPy_Convergence(*solver*, *iterations*, *residual*, *criterion*,
actual_code=None, ***kwargs*)

Bases: *Convergence*

__str__()

Return str(self).

```
class fipy.solvers.scipy.scipyConvergence.SciPy_IterationDivergence(solver, iterations, residual,
                                                                    criterion,
                                                                    actual_code=None,
                                                                    **kwargs)
```

Bases: *IterationDivergence*

```
__str__()
```

Return str(self).

fipy.solvers.scipy.scipyKrylovSolver

Classes

<i>ScipyKrylovSolver</i> ([tolerance, ...])	The base <i>ScipyKrylovSolver</i> class.

```
class fipy.solvers.scipy.scipyKrylovSolver.SciPyKrylovSolver(tolerance='default',
                                                             absolute_tolerance=0.0,
                                                             criterion='default',
                                                             iterations='default',
                                                             precon='default')
```

Bases: *ScipySolver*

The base *ScipyKrylovSolver* class.

Attention: This class is abstract. Always create one of its subclasses.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required relative error tolerance.
- **absolute_tolerance** (*float*) – Required absolute error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy', }) – Interpretation of *tolerance*. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.scipy.scipySolver

Classes

<i>ScipySolver</i> ([tolerance, absolute_tolerance, ...])	The base <i>ScipySolver</i> class.

```
class fipy.solvers.scipy.scipySolver.ScipySolver(tolerance='default', absolute_tolerance=0.0,
                                                  criterion='default', iterations='default',
                                                  precon='default')
```

Bases: *Solver*

The base *ScipySolver* class.

Attention: This class is abstract. Always create one of its subclasses.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required relative error tolerance.
- **absolute_tolerance** (*float*) – Required absolute error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy', }) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

23.4.7 fipy.solvers.solver

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
```

Classes

<code><i>Solver</i>([tolerance, criterion, iterations, ...])</code>	The base <i>LinearXSolver</i> class.

Exceptions

<code><i>BreakdownWarning</i>(solver, iter, relres)</code>
<code><i>IllConditionedPreconditionerWarning</i>(solver, ...)</code>
<code><i>IllegalInputOrBreakdownWarning</i>(solver, iter, ...)</code>
<code><i>LossOfPrecisionWarning</i>(solver, iter, relres)</code>
<code><i>MatrixIllConditionedWarning</i>(solver, iter, relres)</code>
<code><i>MaximumIterationWarning</i>(solver, iter, relres)</code>
<code><i>NormalConvergence</i>(solver, iter, relres)</code>
<code><i>ParameterWarning</i>(solver, iter, relres)</code>
<code><i>PreconditionerNotPositiveDefiniteWarning</i>(...)</code>
<code><i>PreconditionerWarning</i>(solver, iter, relres)</code>
<code><i>ScalarQuantityOutOfRangeWarning</i>(solver, ...)</code>
<code><i>SolverConvergenceWarning</i>(solver, iter, relres)</code>
<code><i>StagnatedSolverWarning</i>(solver, iter, relres)</code>

exception `fipy.solvers.solver.BreakdownWarning(solver, iter, relres)`

Bases: `SolverConvergenceWarning`

__cause__
exception cause

__context__
exception context

__delattr__(*name*, /)
Implement delattr(self, name).

__getattr__(*name*, /)
Return getattr(self, name).

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__setattr__(*name*, *value*, /)
Implement setattr(self, name, value).

__str__()
Return str(self).

add_note()
Exception.add_note(note) – add a note to the exception

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.solvers.solver.**IllConditionedPreconditionerWarning**(*solver*, *iter*, *relres*)

Bases: [*PreconditionerWarning*](#)

__cause__
exception cause

__context__
exception context

__delattr__(*name*, /)
Implement delattr(self, name).

__getattr__(*name*, /)
Return getattr(self, name).

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__setattr__(*name*, *value*, /)
Implement setattr(self, name, value).

__str__()
Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.solvers.solver.**IllegalInputOrBreakdownWarning**(*solver, iter, relres*)

Bases: [SolverConvergenceWarning](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name, /*)

Implement delattr(self, name).

__getattr__(*name, /*)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(*name, value, /*)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.solvers.solver.**LossOfPrecisionWarning**(*solver, iter, relres*)

Bases: [SolverConvergenceWarning](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name, /*)

Implement delattr(self, name).

__getattr__(*name, /*)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(*name, value, /*)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.solvers.solver.**MatrixIllConditionedWarning**(*solver, iter, relres*)

Bases: [SolverConvergenceWarning](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name, /*)

Implement delattr(self, name).

__getattr__(*name, /*)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(*name, value, /*)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.solvers.solver.**MaximumIterationWarning**(*solver, iter, relres*)

Bases: [SolverConvergenceWarning](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name*, /)

Implement delattr(self, name).

__getattribute__(*name*, /)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(*name*, *value*, /)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.solvers.solver.**NormalConvergence**(*solver*, *iter*, *relres*)

Bases: [SolverConvergenceWarning](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name*, /)

Implement delattr(self, name).

__getattribute__(*name*, /)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(*name*, *value*, /)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception `fipy.solvers.solver.ParameterWarning(solver, iter, relres)`

Bases: [*SolverConvergenceWarning*](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name, /*)

Implement `delattr(self, name)`.

__getattr__(*name, /*)

Return `getattr(self, name)`.

__reduce__()

Helper for pickle.

__repr__()

Return `repr(self)`.

__setattr__(*name, value, /*)

Implement `setattr(self, name, value)`.

__str__()

Return `str(self)`.

add_note()

`Exception.add_note(note)` – add a note to the exception

with_traceback()

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

exception `fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning(solver, iter, relres)`

Bases: [*PreconditionerWarning*](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name, /*)

Implement `delattr(self, name)`.

__getattr__(*name, /*)

Return `getattr(self, name)`.

__reduce__()

Helper for pickle.

__repr__()

Return `repr(self)`.

__setattr__(*name, value, /*)

Implement `setattr(self, name, value)`.

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.solvers.solver.PreconditionerWarning(solver, iter, relres)

Bases: [SolverConvergenceWarning](#)

__cause__

exception cause

__context__

exception context

__delattr__(name, /)

Implement delattr(self, name).

__getattr__(name, /)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(name, value, /)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.solvers.solver.ScalarQuantityOutOfRangeWarning(solver, iter, relres)

Bases: [SolverConvergenceWarning](#)

__cause__

exception cause

__context__

exception context

__delattr__(name, /)

Implement delattr(self, name).

__getattr__(name, /)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(name, value, /)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class fipy.solvers.solver.**Solver**(tolerance='default', criterion='default', iterations='default',
precon='default')

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 residual tolerance.
- **criterion** ({'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}, *optional*) – Interpretation of tolerance. See [Convergence](#) for more information.
- **iterations** ([int](#)) – Maximum number of iterative steps to perform.
- **precon** ([Preconditioner](#)) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

exception `fipy.solvers.solver.SolverConvergenceWarning(solver, iter, relres)`

Bases: [Warning](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name, /*)

Implement `delattr(self, name)`.

__getattr__(*name, /*)

Return `getattr(self, name)`.

__reduce__()

Helper for pickle.

__repr__()

Return `repr(self)`.

__setattr__(*name, value, /*)

Implement `setattr(self, name, value)`.

__str__()

Return `str(self)`.

add_note()

`Exception.add_note(note)` – add a note to the exception

with_traceback()

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return `self`.

exception `fipy.solvers.solver.StagnatedSolverWarning(solver, iter, relres)`

Bases: [SolverConvergenceWarning](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name, /*)

Implement `delattr(self, name)`.

__getattr__(*name, /*)

Return `getattr(self, name)`.

__reduce__()

Helper for pickle.

__repr__()

Return `repr(self)`.

__setattr__(*name, value, /*)

Implement `setattr(self, name, value)`.

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

23.4.8 fipy.solvers.test

23.4.9 fipy.solvers.trilinos

Modules

fipy.solvers.trilinos.aztecConvergence

fipy.solvers.trilinos.comms

fipy.solvers.trilinos.linearBicgstabSolver

fipy.solvers.trilinos.linearCGSSolver

fipy.solvers.trilinos.linearCGSolver

fipy.solvers.trilinos.linearGMRESSolver

fipy.solvers.trilinos.linearLUSolver

fipy.solvers.trilinos.preconditioners

*fipy.solvers.trilinos.
trilinosAztec00Solver*

fipy.solvers.trilinos.trilinosMLTest

*fipy.solvers.trilinos.
trilinosNonlinearSolver*

fipy.solvers.trilinos.trilinosSolver

fipy.solvers.trilinos.aztecConvergence

Classes

AZ_BreakdownDivergence(solver, iterations, ...)

AZ_IllConditionedDivergence(solver, ...[, ...])

AZ_IterationDivergence(solver, iterations, ...)

AZ_LossOfAccuracyConvergence(solver, ...[, ...]) Numerical loss of precision occurred.

AZ_NormalConvergence(solver, iterations, ...)

AZ_ParameterDivergence(solver, iterations, ...)

```
class fipy.solvers.trilinos.aztecConvergence.AZ_BreakdownDivergence(solver, iterations, residual,
                                                                    criterion,
                                                                    actual_code=None,
                                                                    **kwargs)
```

Bases: *BreakdownDivergence*

```
__str__()
```

Return str(self).

```
class fipy.solvers.trilinos.aztecConvergence.AZ_IllConditionedDivergence(solver, iterations,
                                                                    residual, criterion,
                                                                    actual_code=None,
                                                                    **kwargs)
```

Bases: *IllConditionedDivergence*

```
__str__()
```

Return str(self).

```
class fipy.solvers.trilinos.aztecConvergence.AZ_IterationDivergence(solver, iterations, residual,
                                                                    criterion,
                                                                    actual_code=None,
                                                                    **kwargs)
```

Bases: *IterationDivergence*

```
__str__()
```

Return str(self).

```
class fipy.solvers.trilinos.aztecConvergence.AZ_LossOfAccuracyConvergence(solver, iterations,
                                                                    residual, criterion,
                                                                    actual_code=None,
                                                                    **kwargs)
```

Bases: *LossOfAccuracyConvergence*

Numerical loss of precision occurred.

```
__str__()
```

Return str(self).

```
class fipy.solvers.trilinos.aztecConvergence.AZ_NormalConvergence(solver, iterations, residual,
                                                                    criterion, actual_code=None,
                                                                    **kwargs)
```

Bases: *Convergence*

`__str__()`

Return str(self).

```
class fipy.solvers.trilinos.aztecConvergence.AZ_ParameterDivergence(solver, iterations, residual,
                                                                criterion,
                                                                actual_code=None,
                                                                **kwargs)
```

Bases: *Divergence*

`__str__()`

Return str(self).

fipy.solvers.trilinos.comms

Modules

```
fipy.solvers.trilinos.comms.
epetraCommWrapper
fipy.solvers.trilinos.comms.
parallelEpetraCommWrapper
fipy.solvers.trilinos.comms.
serialEpetraCommWrapper
```

fipy.solvers.trilinos.comms.epetraCommWrapper

Classes

<i>EpetraCommWrapper()</i>	MPI Communicator wrapper

```
class fipy.solvers.trilinos.comms.epetraCommWrapper.EpetraCommWrapper
```

Bases: *CommWrapper*

MPI Communicator wrapper

Encapsulates capabilities needed for Epetra. Some capabilities are not parallel.

`__getstate__()`

Helper for pickle.

`__repr__()`

Return repr(self).

fipy.solvers.trilinos.comms.parallelEpetraCommWrapper**Classes**

*ParallelEpetraCommWrapper()*MPI Communicator wrapper

class fipy.solvers.trilinos.comms.parallelEpetraCommWrapper.**ParallelEpetraCommWrapper**Bases: *EpetraCommWrapper*

MPI Communicator wrapper

Encapsulates capabilities needed for both Epetra and mpi4py.

MaxAll(*obj*)

return max across all processes

MinAll(*obj*)

return min across all processes

__getstate__()

Helper for pickle.

__repr__()

Return repr(self).

allgather(*obj*)mpi4py *allgather*Communicates copies of each *sendobj* to every rank in the comm, creating a rank-dimensional list of *sendobj* objects.

```
>>> m4count = self.mpi4py_comm.allgather(self.mpi4py_comm.Get_rank())
>>> for i in range(self.mpi4py_comm.Get_size()):
...     assert m4count[i] == i
```

fipy.solvers.trilinos.comms.serialEpetraCommWrapper**Classes**

SerialEpetraCommWrapper()

class fipy.solvers.trilinos.comms.serialEpetraCommWrapper.**SerialEpetraCommWrapper**Bases: *EpetraCommWrapper***__getstate__**()

Helper for pickle.

__repr__()

Return repr(self).

fipy.solvers.trilinos.linearBicgstabSolver**Classes**

<code>LinearBicgstabSolver([tolerance, criterion, ...])</code>	Interface to the Biconjugate Gradient (Stabilized) (<i>BiCGSTAB</i>) solver in <i>Trilinos</i> .

```
class fipy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver(tolerance='default',
                                                                    criterion='default',
                                                                    iterations='default',
                                                                    precon='default')
```

Bases: *TrilinosAztec00Solver*

Interface to the Biconjugate Gradient (Stabilized) (*BiCGSTAB*) solver in *Trilinos*.

Uses the *JacobiPreconditioner* by default.

Create a *Solver* object.

Parameters

- **tolerance** (`float`) – Required residual tolerance.
- **criterion** ({'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}, *optional*) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of *JacobiPreconditioner*

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.trilinos.linearCGSSolver**Classes**

<i>LinearCGSSolver</i> ([tolerance, criterion, ...])	Interface to the conjugate gradient squared (<i>CGS</i>) solver in <i>Trilinos</i>

```
class fipy.solvers.trilinos.linearCGSSolver.LinearCGSSolver(tolerance='default',  
                                                         criterion='default',  
                                                         iterations='default', precon='default')
```

Bases: *TrilinosAztec00Solver*

Interface to the conjugate gradient squared (*CGS*) solver in *Trilinos*

Uses the *MultilevelDDPreconditioner* by default.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required residual tolerance.
- **criterion** ({'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}, *optional*) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of *MultilevelDDPreconditioner*

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.trilinos.linearCGSolver**Classes**

<i>LinearCGSolver</i> ([tolerance, criterion, ...])	Interface to the conjugate gradient (<i>CG</i>) solver in <i>Trilinos</i> .
<i>LinearPCGSolver</i>	alias of <i>LinearCGSolver</i>

class fipy.solvers.trilinos.linearCGSolver.**LinearCGSolver**(*tolerance*='default', *criterion*='default', *iterations*='default', *precon*='default')

Bases: *TrilinosAztec00Solver*

Interface to the conjugate gradient (*CG*) solver in *Trilinos*.

Uses the *MultilevelDDPreconditioner* by default.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required residual tolerance.
- **criterion** ({'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}, *optional*) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of *MultilevelDDPreconditioner*

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.trilinos.linearCGSolver.**LinearPCGSolver**

alias of *LinearCGSolver*

fipy.solvers.trilinos.linearGMRESSolver**Classes**

<code>LinearGMRESSolver</code> ([tolerance, criterion, ...])	Interface to the generalized minimal residual (<i>GMRES</i>) solver in <i>Trilinos</i> .

```
class fipy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver(tolerance='default',
                                                                criterion='default',
                                                                iterations='default',
                                                                precon='default')
```

Bases: *TrilinosAztec00Solver*

Interface to the generalized minimal residual (*GMRES*) solver in *Trilinos*.

Uses the *MultilevelDDPreconditioner* by default.

Create a *Solver* object.

Parameters

- **tolerance** (`float`) – Required residual tolerance.
- **criterion** ({'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}, *optional*) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of *MultilevelDDPreconditioner*

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.trilinos.linearLUSolver**Classes**

<i>LinearLUSolver</i> ([tolerance, criterion, ...])	Interface to the Amesos KLU solver in <i>Trilinos</i> .

class fipy.solvers.trilinos.linearLUSolver.**LinearLUSolver**(*tolerance='default', criterion='default', precon=None, iterations=10*)

Bases: *TrilinosSolver*

Interface to the Amesos KLU solver in *Trilinos*.

KLU is a direct, serial *LU* solver.

Parameters

- **tolerance** (*float*) – Required error tolerance.
- **criterion** ({'default', 'unscaled', 'RHS', 'matrix', 'initial', 'legacy'}) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – *ignored*

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion="legacy"*

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion="legacy"*

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.trilinos.preconditioners

class fipy.solvers.trilinos.preconditioners.**DomDecompPreconditioner**

Bases: *TrilinosPreconditioner*

Domain Decomposition preconditioner for *TrilinosSolver*.

class fipy.solvers.trilinos.preconditioners.**ICPreconditioner**

Bases: *TrilinosPreconditioner*

Incomplete Cholesky Preconditioner from IFPACK for *TrilinosSolver*.

class fipy.solvers.trilinos.preconditioners.**ILUPreconditioner**

Bases: *DomDecompPreconditioner*

Incomplete LU Domain Decomposition preconditioner for *TrilinosSolver*.

class fipy.solvers.trilinos.preconditioners.**JacobiPreconditioner**

Bases: *TrilinosPreconditioner*

Jacobi preconditioner for *TrilinosSolver*.

class fipy.solvers.trilinos.preconditioners.**MultilevelDDMLPreconditioner**(*levels=3*)

Bases: *MultilevelPreconditioner*

3-level algebraic domain decomposition multilevel preconditioner for *TrilinosSolver*.

Parameters

levels (*int*) – Maximum number of levels

class fipy.solvers.trilinos.preconditioners.**MultilevelDDPreconditioner**(*levels=2*)

Bases: *MultilevelPreconditioner*

Classical smoothed aggregation-based 2-level domain decomposition preconditioner for *TrilinosSolver*.

Parameters

levels (*int*) – Maximum number of levels

class fipy.solvers.trilinos.preconditioners.**MultilevelNSSAPreconditioner**(*levels=10*)

Bases: *MultilevelPreconditioner*

Energy-based minimizing smoothed aggregation preconditioner for *TrilinosSolver*.

Suitable for highly convective non-symmetric fluid flow problems.

Parameters

levels (*int*) – Maximum number of levels

class fipy.solvers.trilinos.preconditioners.**MultilevelSAPreconditioner**(*levels=10*)

Bases: *MultilevelPreconditioner*

Classical smoothed aggregation multilevel preconditioner for *TrilinosSolver*.

Suitable for symmetric positive definite or nearly symmetric positive definite systems.

Parameters

levels (*int*) – Maximum number of levels

class fipy.solvers.trilinos.preconditioners.**MultilevelSGSPreconditioner**(*levels=10*)

Bases: *MultilevelPreconditioner*

Multilevel preconditioner using Symmetric Gauss-Seidel smoothing for *TrilinosSolver*.

Parameters

levels (*int*) – Maximum number of levels

class fipy.solvers.trilinos.preconditioners.**MultilevelSolverSmootherPreconditioner**(*levels=10*)

Bases: *MultilevelPreconditioner*

Multilevel preconditioner using Aztec solvers as smoothers for *TrilinosSolver*.

Parameters

levels (*int*) – Maximum number of levels

Modules

<code>fipy.solvers.trilinos.preconditioners.domDecompPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.icPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.iluPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.jacobiPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.multilevelPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner</code>
<code>fipy.solvers.trilinos.preconditioners.trilinosPreconditioner</code>

`fipy.solvers.trilinos.preconditioners.domDecompPreconditioner`

Classes

<code>DomDecompPreconditioner()</code>	Domain Decomposition preconditioner for <i>TrilinosSolver</i> .

class

`fipy.solvers.trilinos.preconditioners.domDecompPreconditioner.DomDecompPreconditioner`

Bases: *TrilinosPreconditioner*

Domain Decomposition preconditioner for *TrilinosSolver*.

fipy.solvers.trilinos.preconditioners.icPreconditioner**Classes**

<i>ICPreconditioner()</i>	Incomplete Cholesky Preconditioner from IFPACK for <i>TrilinosSolver</i> .

```
class fipy.solvers.trilinos.preconditioners.icPreconditioner.ICPreconditioner
```

Bases: *TrilinosPreconditioner*

Incomplete Cholesky Preconditioner from IFPACK for *TrilinosSolver*.

fipy.solvers.trilinos.preconditioners.iluPreconditioner**Classes**

<i>ILUPreconditioner()</i>	Incomplete LU Domain Decomposition preconditioner for <i>TrilinosSolver</i> .

```
class fipy.solvers.trilinos.preconditioners.iluPreconditioner.ILUPreconditioner
```

Bases: *DomDecompPreconditioner*

Incomplete LU Domain Decomposition preconditioner for *TrilinosSolver*.

fipy.solvers.trilinos.preconditioners.jacobiPreconditioner**Classes**

<i>JacobiPreconditioner()</i>	Jacobi preconditioner for <i>TrilinosSolver</i> .

```
class fipy.solvers.trilinos.preconditioners.jacobiPreconditioner.JacobiPreconditioner
```

Bases: *TrilinosPreconditioner*

Jacobi preconditioner for *TrilinosSolver*.

fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner**Classes**

<i>MultilevelDDMLPreconditioner([levels])</i>	3-level algebraic domain decomposition multilevel preconditioner for <i>TrilinosSolver</i> .

class fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner.MultilevelDDMLPreconditioner(*levels*=

Bases: *MultilevelPreconditioner*

3-level algebraic domain decomposition multilevel preconditioner for *TrilinosSolver*.

Parameters

levels (*int*) – Maximum number of levels

fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner

Classes

<i>MultilevelDDPreconditioner</i> ([<i>levels</i>])	Classical smoothed aggregation-based 2-level domain decomposition preconditioner for <i>TrilinosSolver</i> .

class fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner.MultilevelDDPreconditioner(*levels*=

Bases: *MultilevelPreconditioner*

Classical smoothed aggregation-based 2-level domain decomposition preconditioner for *TrilinosSolver*.

Parameters

levels (*int*) – Maximum number of levels

fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner

Classes

<i>MultilevelNSSAPreconditioner</i> ([<i>levels</i>])	Energy-based minimizing smoothed aggregation preconditioner for <i>TrilinosSolver</i> .

class fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner.MultilevelNSSAPreconditioner(*levels*=

Bases: *MultilevelPreconditioner*

Energy-based minimizing smoothed aggregation preconditioner for *TrilinosSolver*.

Suitable for highly convective non-symmetric fluid flow problems.

Parameters

levels (*int*) – Maximum number of levels

fipy.solvers.trilinos.preconditioners.multilevelPreconditioner

Classes

<i>MultilevelPreconditioner</i> ([<i>levels</i>])	Base class for multilevel preconditioners for <i>TrilinosSolver</i> .

```
class fipy.solvers.trilinos.preconditioners.multilevelPreconditioner.MultilevelPreconditioner(levels=10)
```

Bases: *TrilinosPreconditioner*

Base class for multilevel preconditioners for *TrilinosSolver*.

Parameters

levels (*int*) – Maximum number of levels

fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner

Classes

<i>MultilevelSAPreconditioner</i> ([levels])	Classical smoothed aggregation multilevel preconditioner for <i>TrilinosSolver</i> .

```
class fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner.MultilevelSAPreconditioner(levels=
```

Bases: *MultilevelPreconditioner*

Classical smoothed aggregation multilevel preconditioner for *TrilinosSolver*.

Suitable for symmetric positive definite or nearly symmetric positive definite systems.

Parameters

levels (*int*) – Maximum number of levels

fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner

Classes

<i>MultilevelSGSPreconditioner</i> ([levels])	Multilevel preconditioner using Symmetric Gauss-Seidel smoothing for <i>TrilinosSolver</i> .

```
class fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner.MultilevelSGSPreconditioner(lev
```

Bases: *MultilevelPreconditioner*

Multilevel preconditioner using Symmetric Gauss-Seidel smoothing for *TrilinosSolver*.

Parameters

levels (*int*) – Maximum number of levels

fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner

Classes

<i>MultilevelSolverSmootherPreconditioner</i> ([levels])	Multilevel preconditioner using Aztec solvers as smoothers for <i>TrilinosSolver</i> .

class `fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner.MultilevelSolverSmootherPreconditioner`

Bases: *MultilevelPreconditioner*

Multilevel preconditioner using Aztec solvers as smoothers for *TrilinosSolver*.

Parameters

levels (*int*) – Maximum number of levels

fipy.solvers.trilinos.preconditioners.trilinosPreconditioner

Classes

TrilinosPreconditioner()

Base class of preconditioners for *TrilinosSolver*.

class `fipy.solvers.trilinos.preconditioners.trilinosPreconditioner.TrilinosPreconditioner`

Bases: *SolverModifyingPreconditioner*

Base class of preconditioners for *TrilinosSolver*.

Attention: This class is abstract. Always create one of its subclasses.

fipy.solvers.trilinos.trilinosAztecOOSolver

Classes

TrilinosAztecOOSolver([tolerance, ...])

Attention:

This class is abstract, always create one of its subclasses.

class `fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver`(*tolerance*='default', *criterion*='default', *iterations*='default', *precon*='default')

Bases: *TrilinosSolver*

Attention: This class is abstract, always create one of its subclasses. It provides the code to call all solvers from the Trilinos AztecOO package.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required residual tolerance.
- **criterion** ({'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}, *optional*) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER

alias of *JacobiPreconditioner*

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.trilinos.trilinosMLTest

Classes

<i>TrilinosMLTest</i> ([tolerance, iterations, ...])	This solver class does not actually solve the system, but outputs information about what ML preconditioner settings will work best.

```
class fipy.solvers.trilinos.trilinosMLTest.TrilinosMLTest(tolerance='default', iterations=5,
                                                           MLOptions={}, testUnsupported=False)
```

Bases: *TrilinosSolver*

This solver class does not actually solve the system, but outputs information about what ML preconditioner settings will work best.

For detailed information on the possible parameters for ML, see <https://trilinos.github.io/ml.html>

Currently, passing options to Aztec through ML is not supported.

Parameters

- **tolerance** (`float`) – Required error tolerance.
- **iterations** (`int`) – Maximum number of iterative steps to perform.
- **MLOptions** (`dict`) – Options to pass to ML. This will be passed to `ML.SetParameterList`.
- **testUnsupported** (`bool`) – Test smoothers that are not currently implemented in preconditioner objects.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fiPy.solvers.trilinos.trilinosNonlinearSolver

Classes

<code>TrilinosNonlinearSolver</code> (equation[, ...])	Create a <i>Solver</i> object.

```
class fipy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver(equation,
                                                                              jacobian=None,
                                                                              toler-
                                                                              ance='default',
                                                                              itera-
                                                                              tions='default',
                                                                              printingOp-
                                                                              tions=None,
                                                                              solverOp-
                                                                              tions=None,
                                                                              linearSolverOp-
                                                                              tions=None,
                                                                              lineSearchOp-
                                                                              tions=None,
                                                                              directionOp-
                                                                              tions=None,
                                                                              newtonOp-
                                                                              tions=None)
```

Bases: [*TrilinosSolver*](#)

Create a *Solver* object.

Parameters

- **tolerance** ([*float*](#)) – Required residual tolerance.
- **criterion** ({'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}, *optional*) – Interpretation of tolerance. See [*Convergence*](#) for more information.
- **iterations** ([*int*](#)) – Maximum number of iterative steps to perform.
- **precon** ([*Preconditioner*](#)) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10

Default tolerance for linear solves if *criterion*="legacy"

__repr__()

Return repr(self).

property default_tolerance

Default tolerance for linear solve

fipy.solvers.trilinos.trilinosSolver**Classes***TrilinosSolver*([tolerance, criterion, ...])**Attention:**

This class is abstract. Always create one of its subclasses.

```
class fipy.solvers.trilinos.trilinosSolver.TrilinosSolver(tolerance='default', criterion='default',
                                                         iterations='default', precon='default')
```

Bases: *Solver*

Attention: This class is abstract. Always create one of its subclasses.

Create a *Solver* object.

Parameters

- **tolerance** (*float*) – Required residual tolerance.
- **criterion** ({'default', 'initial', 'unscaled', 'RHS', 'matrix', 'solution', 'preconditioned', 'natural', 'legacy'}, *optional*) – Interpretation of tolerance. See *Convergence* for more information.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) – Preconditioner to use. Not all solver suites support preconditioners.

DEFAULT_ITERATIONS = 1000

Default maximum number of iterative steps to perform

DEFAULT_PRECONDITIONER = None

Default preconditioner to apply to the matrix

DEFAULT_TOLERANCE = 1e-05

Default tolerance for linear solves unless *criterion*="legacy"

LEGACY_TOLERANCE = 1e-10Default tolerance for linear solves if *criterion*="legacy"**__repr__()**

Return repr(self).

property default_tolerance

Default tolerance for linear solve

23.5 fipy.steps

(Obsolete) utilities for iterating time steps

Use [steppyngstounes](#) instead.**fipy.steps.L1error(*args, **kws)**Deprecated since version 4.0: Calculate error explicitly with [L1norm](#).

$$\frac{\|\text{var} - \text{var}^{\text{old}}\|_1}{\|\text{var}^{\text{old}}\|_1}$$

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(*args, **kws)Deprecated since version 4.0: Calculate error explicitly with [L2norm](#).

$$\frac{\|\text{var} - \text{var}^{\text{old}}\|_2}{\|\text{var}^{\text{old}}\|_2}$$

where $\|\vec{x}\|_2$ is the L^2 norm of $\vec{x}$.**Parameters**

- **var** ([CellVariable](#)) – The *CellVariable* in question.
- **matrix** – (ignored)
- **RHSvector** – (ignored)

fipy.steps.LINFerror(*args, **kws)Deprecated since version 4.0: Calculate error explicitly with [LINFnorm](#).

$$\frac{\|\text{var} - \text{var}^{\text{old}}\|_\infty}{\|\text{var}^{\text{old}}\|_\infty}$$

where $\|\vec{x}\|_\infty$ is the L^∞ norm of $\vec{x}$.**Parameters**

- **var** ([CellVariable](#)) – The *CellVariable* in question.
- **matrix** – (ignored)
- **RHSvector** – (ignored)

`fipy.steppers.error(*args, **kws)`

Deprecated since version 4.0: Calculate error explicitly with [numerix](#).

$$\frac{\|\text{var} - \text{var}^{\text{old}}\|_?}{\|\text{var}^{\text{old}}\|_?}$$

where $\|\vec{x}\|_?$ is the normalization of $\vec{x}$ provided by `norm()`.

Parameters

- **var** ([CellVariable](#)) – The *CellVariable* in question.
- **matrix** – (*ignored*)
- **RHSvector** – (*ignored*)
- **norm** (function) – A function that will normalize its *array* argument and return a single number (default: [L1norm\(\)](#)).

`fipy.steppers.residual(*args, **kws)`

Deprecated since version 4.0: Pass a `residualFn` to [sweep\(\)](#).

Determines the residual for the current solution matrix and variable.

$$\|\mathbf{L}\vec{x} - \vec{b}\|_\infty$$

where $\|\vec{\xi}\|_\infty$ is the L^∞ norm of $\vec{\xi}$.

Parameters

- **var** ([CellVariable](#)) – The *CellVariable* in question, *prior* to solution.
- **matrix** (`_SparseMatrix`) – The coefficient matrix at this step/sweep
- **RHSvector** (`ndarray`) – The right hand side vector

`fipy.steppers.sweepMonotonic(*args, **kws)`

Deprecated since version 4.0: Use the [steppyngstounes](#) package instead.

Repeatedly calls `fn(*args, **kwargs)()` until the residual returned by `fn()` is no longer decreasing.

Parameters

- **fn** (function) – The function to call
- ***args** –
- ****kwargs** –

Return type

`float`

Modules

[`fipy.steppers.pidStepper`](#)

[`fipy.steppers.pseudoRKQSStepper`](#)

[`fipy.steppers.stepper`](#)

23.5.1 fipy.steppers.pidStepper

`fipy.steppers.pidStepper.PIDStepper(*args, **kws)`

Deprecated since version 4.0: Use the *steppyngstounes* package instead.

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},
}
```

23.5.2 fipy.steppers.pseudoRKQSStepper

`fipy.steppers.pseudoRKQSStepper.PseudoRKQSStepper(*args, **kws)`

Deprecated since version 4.0: Use the *steppyngstounes* package instead.

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.

23.5.3 fipy.steppers.stepper

Classes

Stepper([vardata])

Rudimentary utility class for iterating time steps

class `fipy.steppers.stepper.Stepper(vardata=())`

Bases: `object`

Rudimentary utility class for iterating time steps

Use *steppyngstounes* instead.

23.6 fipy.terms

Discretizations of partial differential equation expressions

Exceptions

AbstractBaseClassError([s])

ExplicitVariableError([s])

IncorrectSolutionVariable([s])

SolutionVariableNumberError([s])

SolutionVariableRequiredError([s])

TermMultiplyError([s])

TransientTermError([s])

VectorCoeffError([s])

exception `fipy.terms.AbstractBaseClassError`(*s*="can't instantiate abstract base class")

Bases: `NotImplementedError`

__cause__

exception cause

__context__

exception context

__delattr__(*name*, /)

Implement `delattr`(self, name).

__getattr__(*name*, /)

Return `getattr`(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return `repr`(self).

__setattr__(*name*, *value*, /)

Implement `setattr`(self, name, value).

__str__()

Return `str`(self).

add_note()

`Exception.add_note(note)` – add a note to the exception

with_traceback()

`Exception.with_traceback(tb)` – set `self.__traceback__` to `tb` and return self.

exception `fipy.terms.ExplicitVariableError`(*s*="Terms with explicit Variables cannot mix with Terms with implicit Variables.")

Bases: `Exception`

__cause__
exception cause

__context__
exception context

__delattr__(*name*, /)
Implement delattr(self, name).

__getattr__(*name*, /)
Return getattr(self, name).

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__setattr__(*name*, *value*, /)
Implement setattr(self, name, value).

__str__()
Return str(self).

add_note()
Exception.add_note(note) – add a note to the exception

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.terms.**IncorrectSolutionVariable**(*s*='The solution variable is incorrect.')

Bases: [Exception](#)

__cause__
exception cause

__context__
exception context

__delattr__(*name*, /)
Implement delattr(self, name).

__getattr__(*name*, /)
Return getattr(self, name).

__reduce__()
Helper for pickle.

__repr__()
Return repr(self).

__setattr__(*name*, *value*, /)
Implement setattr(self, name, value).

__str__()
Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.terms.**SolutionVariableNumberError**(*s='Different number of solution variables and equations.'*)

Bases: [Exception](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name, /*)

Implement delattr(self, name).

__getattr__(*name, /*)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(*name, value, /*)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.terms.**SolutionVariableRequiredError**(*s='The solution variable needs to be specified.'*)

Bases: [Exception](#)

__cause__

exception cause

__context__

exception context

__delattr__(*name, /*)

Implement delattr(self, name).

__getattr__(*name, /*)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(name, value, /)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.terms.**TermMultiplyError**(s='Must multiply terms by int or float.')

Bases: [Exception](#)

__cause__

exception cause

__context__

exception context

__delattr__(name, /)

Implement delattr(self, name).

__getattr__(name, /)

Return getattr(self, name).

__reduce__()

Helper for pickle.

__repr__()

Return repr(self).

__setattr__(name, value, /)

Implement setattr(self, name, value).

__str__()

Return str(self).

add_note()

Exception.add_note(note) – add a note to the exception

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception fipy.terms.**TransientTermError**(s='The equation requires a TransientTerm with explicit convection.')

Bases: [Exception](#)

__cause__

exception cause

__context__

exception context

```

__delattr__(name, /)
    Implement delattr(self, name).

__getattribute__(name, /)
    Return getattr(self, name).

__reduce__()
    Helper for pickle.

__repr__()
    Return repr(self).

__setattr__(name, value, /)
    Implement setattr(self, name, value).

__str__()
    Return str(self).

add_note()
    Exception.add_note(note) – add a note to the exception

with_traceback()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

```

exception fipy.terms.**VectorCoeffError**(s=*The coefficient must be a vector value.*)

Bases: [TypeError](#)

```

__cause__
    exception cause

__context__
    exception context

__delattr__(name, /)
    Implement delattr(self, name).

__getattribute__(name, /)
    Return getattr(self, name).

__reduce__()
    Helper for pickle.

__repr__()
    Return repr(self).

__setattr__(name, value, /)
    Implement setattr(self, name, value).

__str__()
    Return str(self).

add_note()
    Exception.add_note(note) – add a note to the exception

with_traceback()
    Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

```

Modules

<code>fipy.terms.abstractBinaryTerm</code>
<code>fipy.terms.abstractConvectionTerm</code>
<code>fipy.terms.abstractDiffusionTerm</code>
<code>fipy.terms.abstractUpwindConvectionTerm</code>
<code>fipy.terms.advectionTerm</code>
<code>fipy.terms.asymmetricConvectionTerm</code>
<code>fipy.terms.binaryTerm</code>
<code>fipy.terms.cellTerm</code>
<code>fipy.terms.centralDiffConvectionTerm</code>
<code>fipy.terms.coupledBinaryTerm</code>
<code>fipy.terms.diffusionTerm</code>
<code>fipy.terms.diffusionTermCorrection</code>
<code>fipy.terms.diffusionTermNoCorrection</code>
<code>fipy.terms.explicitDiffusionTerm</code>
<code>fipy.terms.explicitSourceTerm</code>
<code>fipy.terms.explicitUpwindConvectionTerm</code>
<code>fipy.terms.exponentialConvectionTerm</code>
<code>fipy.terms.faceTerm</code>
<code>fipy.terms.firstOrderAdvectionTerm</code>
<code>fipy.terms.hybridConvectionTerm</code>
<code>fipy.terms.implicitDiffusionTerm</code>
<code>fipy.terms.implicitSourceTerm</code>
<code>fipy.terms.nonDiffusionTerm</code>
<code>fipy.terms.powerLawConvectionTerm</code>
<code>fipy.terms.residualTerm</code>

continues on next page

Table 2 – continued from previous page

<code>fiPy.terms.sourceTerm</code>
<code>fiPy.terms.term</code>
<code>fiPy.terms.test</code>
<code>fiPy.terms.transientTerm</code>
<code>fiPy.terms.unaryTerm</code>
<code>fiPy.terms.upwindConvectionTerm</code>
<code>fiPy.terms.vanLeerConvectionTerm</code>

23.6.1 `fiPy.terms.abstractBinaryTerm`

23.6.2 `fiPy.terms.abstractConvectionTerm`

23.6.3 `fiPy.terms.abstractDiffusionTerm`

23.6.4 `fiPy.terms.abstractUpwindConvectionTerm`

23.6.5 `fiPy.terms.advectionTerm`

Classes

<code>AdvectionTerm</code> ([coeff])	The <i>AdvectionTerm</i> object constructs the <i>b</i> vector contribution for the advection term given by

class `fiPy.terms.advectionTerm.AdvectionTerm`(coeff=None)

Bases: `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:

$$-|\nabla\phi| = -\left|\frac{\partial\phi}{\partial x}\right| = -2|x|$$

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.

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.

__eq__(*other*)

Return self==value.

__hash__()

Return hash(self).

__mul__(*other*)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.]])
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(*other*)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.]])
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (`CellVariable`) – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$
- **norm** (`float`) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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*

Returns

residual – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$

Return type

`CellVariable`

23.6.6 fipy.terms.asymmetricConvectionTerm

23.6.7 fipy.terms.binaryTerm

23.6.8 fipy.terms.cellTerm

Classes

`CellTerm([coeff, var])`

Attention:

This class is abstract. Always create one of its subclasses.

```
class fipy.terms.cellTerm.CellTerm(coeff=1.0, var=None)
```

Bases: `_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.

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.

__eq__(*other*)

Return self==value.

__hash__()

Return hash(self).

__mul__(*other*)

Multiply a term


```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.
```

(continues on next page)

(continued from previous page)

```

↪ 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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (float) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (*bool*) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.9 fipy.terms.centralDiffConvectionTerm

Classes

CentralDifferenceConvectionTerm([coeff, var]) This *Term* represents

class fipy.terms.centralDiffConvectionTerm.**CentralDifferenceConvectionTerm**(coeff=1.0, var=None)

Bases: *_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):
```

(continues on next page)

(continued from previous page)

```

...
VectorCoefficientError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoefficientError: 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):
...
VectorCoefficientError: 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.]]), 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 (*MeshVariable*) – The *Term*'s coefficient value.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (`float`) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (bool) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.10 fipy.terms.coupledBinaryTerm

23.6.11 fipy.terms.diffusionTerm

Classes

<i>DiffusionTerm</i> ([coeff, var])	This term represents a higher order diffusion term. The order of the term is determined by the number of <i>coeffs</i> , such that::

class fipy.terms.diffusionTerm.**DiffusionTerm**(*coeff=(1.0,)*, *var=None*)

Bases: *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.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (*float*) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (bool) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.12 fipy.terms.diffusionTermCorrection

Classes

<i>DiffusionTermCorrection</i> ([coeff, var])	Create a <i>Term</i> .

class fipy.terms.diffusionTermCorrection.**DiffusionTermCorrection**(coeff=(1.0,), var=None)

Bases: *_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.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{old} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (float) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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*

Returns

residual – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$

Return type

CellVariable

23.6.13 fipy.terms.diffusionTermNoCorrection

Classes

<i>DiffusionTermNoCorrection</i> ([coeff, var])	Create a <i>Term</i> .

```
class fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection(coeff=(1.0), var=None)
```

Bases: *_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.

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.

`__eq__(other)`

Return self==value.

`__hash__()`

Return hash(self).

`__repr__()`

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (`CellVariable`) – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$
- **norm** (`float`) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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*

Returns

residual – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$

Return type

`CellVariable`

23.6.14 fipy.terms.explicitDiffusionTerm

Classes

<code>ExplicitDiffusionTerm([coeff, var])</code>	The discretization for the <i>ExplicitDiffusionTerm</i> is given by

class fipy.terms.explicitDiffusionTerm.**ExplicitDiffusionTerm**(*coeff*=(1.0), *var*=None)

Bases: `_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.

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.

`__eq__`(*other*)

Return self==value.

`__hash__`()

Return hash(self).

`__repr__`()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{old} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (`float`) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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*

Returns

residual – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$

Return type

CellVariable

23.6.15 fipy.terms.explicitSourceTerm

23.6.16 fipy.terms.explicitUpwindConvectionTerm

Classes

<i>ExplicitUpwindConvectionTerm</i> ([coeff, var])	The discretization for this <i>Term</i> is given by

```
class fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm(coeff=1.0,
                                                                              var=None)
```

Bases: *_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)))
>>> (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 (*MeshVariable*) – The *Term*’s coefficient value.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (float) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (bool) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.17 fipy.terms.exponentialConvectionTerm

Classes

ExponentialConvectionTerm([coeff, var])

The discretization for this *Term* is given by

class fipy.terms.exponentialConvectionTerm.**ExponentialConvectionTerm**(*coeff=1.0, var=None*)

Bases: *_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):
...
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 (*MeshVariable*) – The *Term*’s coefficient value.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (*float*) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (bool) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.18 fipy.terms.faceTerm

Classes

FaceTerm([coeff, var])

Attention:

This class is abstract. Always create one of its subclasses.

```
class fipy.terms.faceTerm.FaceTerm(coeff=1.0, var=None)
```

Bases: `_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.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * _NonDiffusionTerm(coeff=0.5)
_NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
      [ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
      [ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (*float*) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (`bool`) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.19 fipy.terms.firstOrderAdvectionTerm

Classes

FirstOrderAdvectionTerm([coeff])

The *FirstOrderAdvectionTerm* object constructs the b vector contribution for the advection term given by

class fipy.terms.firstOrderAdvectionTerm.**FirstOrderAdvectionTerm**(coeff=None)

Bases: *_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.

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.

`__eq__(other)`

Return self==value.

`__hash__()`

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.**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)
```

(continues on next page)

(continued from previous page)

```
>>> 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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (float) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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*

Returns

residual – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$

Return type

CellVariable

23.6.20 fipy.terms.hybridConvectionTerm

Classes

HybridConvectionTerm([coeff, var])

The discretization for this *Term* is given by

class fipy.terms.hybridConvectionTerm.**HybridConvectionTerm**(coeff=1.0, var=None)

Bases: *_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):
```

(continues on next page)

(continued from previous page)

```

...
VectorCoefficientError: The coefficient must be a vector value.
>>> __ConvectionTerm(coeff = fv)
Traceback (most recent call last):
...
VectorCoefficientError: 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):
...
VectorCoefficientError: 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.]]), 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.]]), 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 (*MeshVariable*) – The *Term*'s coefficient value.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

`CellVariable`

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.

Returns

- **residual** (`CellVariable`) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (`float`) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (bool) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.21 fipy.terms.implicitDiffusionTerm

fipy.terms.implicitDiffusionTerm.**ImplicitDiffusionTerm**

alias of *DiffusionTerm*

23.6.22 fipy.terms.implicitSourceTerm

Classes

ImplicitSourceTerm([coeff, var])

The *ImplicitSourceTerm* represents

class fipy.terms.implicitSourceTerm.**ImplicitSourceTerm**(*coeff=1.0, var=None*)

Bases: *SourceTerm*

The *ImplicitSourceTerm* represents

$$\int_V \phi S dV \simeq \phi_P S_P V_P$$

where *S* is the *coeff* value.

Parameters

- **coeff** (`float` or `CellVariable`) – Proportionality coefficient S (default: 1)
- **var** (`CellVariable`) – Variable ϕ that `ImplicitSourceTerm` is implicit in.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (float) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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*

Returns

residual – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$

Return type

CellVariable

23.6.23 fipy.terms.nonDiffusionTerm

23.6.24 fipy.terms.powerLawConvectionTerm

Classes

PowerLawConvectionTerm([coeff, var])

The discretization for this *Term* is given by

```
class fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm(coeff=1.0, var=None)
```

Bases: *_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):
...
VectorCoefficientError: 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):
...
VectorCoefficientError: 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,)))).
```

(continues on next page)

(continued from previous page)

```

→solve(var=cv2, solver=DummySolver(), dt=1.)
>>> (TransientTerm() - ExplicitUpwindConvectionTerm(coeff = (0, 0))).solve(var=cv2,
→solver=DummySolver(), dt=1.)

```

Parameters

coeff (*MeshVariable*) – The *Term*’s coefficient value.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```

>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)

```

Test for ticket:291.

```

>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))

```

__neg__()

Negate a *Term*.

```

>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)

```

__repr__()

The representation of a *Term* object is given by,

```

>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)

```

__rmul__(other)

Multiply a term

```

>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)

```

Test for ticket:291.

```

>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0

```

(continues on next page)

(continued from previous page)

```
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (*float*) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (*bool*) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.25 fipy.terms.residualTerm

Classes

<code>ResidualTerm(equation[, underRelaxation])</code>	The <i>ResidualTerm</i> is a special form of explicit <i>SourceTerm</i> that adds the residual of one equation to another equation.

class fipy.terms.residualTerm.**ResidualTerm**(*equation*, *underRelaxation*=1.0)

Bases: `_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.

Parameters

- **coeff** (`float` or *CellVariable*) – Coefficient of source (default: 0)
- **var** (*CellVariable*) – Variable ϕ that *SourceTerm* is implicit in.

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.

`__eq__(other)`

Return self==value.

`__hash__()`

Return hash(self).

`__mul__(other)`

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

`__neg__()`

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

`__repr__()`

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(*other*)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.]])
```

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.
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.

Returns**error** – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$ **Return type***CellVariable*

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (`CellVariable`) – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$
- **norm** (`float`) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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*

Returns

residual – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$

Return type

`CellVariable`

23.6.26 fipy.terms.sourceTerm

Classes

SourceTerm([coeff, var])

Attention:

This class is abstract. Always create one of its subclasses.

```
class fipy.terms.sourceTerm.SourceTerm(coeff=0.0, var=None)
```

Bases: *CellTerm*

Attention: This class is abstract. Always create one of its subclasses.

Parameters

- **coeff** (*float* or *CellVariable*) – Coefficient of source (default: 0)
- **var** (*CellVariable*) – Variable ϕ that *SourceTerm* is implicit in.

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.

__eq__(*other*)

Return self==value.

__hash__()

Return hash(self).

__mul__(*other*)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{old} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (*float*) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (`bool`) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.27 fipy.terms.term

Classes

Term([coeff, var])

Base class for elements of a partial differential equation.

class fipy.terms.term.**Term**(*coeff*=1.0, *var*=None)

Bases: `object`

Base class for elements of a partial differential equation.

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.

__eq__(*other*)

Return self==value.

__hash__()

Return hash(self).

__repr__()

Return repr(self).

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (float) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (bool) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.28 fipy.terms.test

23.6.29 fipy.terms.transientTerm

Classes

TransientTerm([coeff, var])

The *TransientTerm* represents

class fipy.terms.transientTerm.**TransientTerm**(coeff=1.0, var=None)

Bases: *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.

```
>>> 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), rtol=2e-5))
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.

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.

__eq__(*other*)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

`CellVariable`

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.

Returns

- **residual** (`CellVariable`) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (`float`) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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*

Returns

residual – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$

Return type

CellVariable

23.6.30 fipy.terms.unaryTerm

23.6.31 fipy.terms.upwindConvectionTerm

Classes

UpwindConvectionTerm([coeff, var])

The discretization for this *Term* is given by

class fipy.terms.upwindConvectionTerm.**UpwindConvectionTerm**(*coeff=1.0, var=None*)

Bases: *_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)))
>>> (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 (*MeshVariable*) – The *Term*’s coefficient value.

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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (*float*) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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 $\vec{r} = L\vec{x} - \vec{b}$ in the *residualVector* member of *Term*
- **cacheError** (bool) – If *True*, use the residual vector $\vec{r}$ to solve $L\vec{e} = \vec{r}$ for the error vector $\vec{e}$ and store it in the *errorVector* member of *Term*

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

23.6.32 fipy.terms.vanLeerConvectionTerm

Classes

<i>VanLeerConvectionTerm</i> ([coeff, var])	Create a <i>_AbstractConvectionTerm</i> object.

class fipy.terms.vanLeerConvectionTerm.**VanLeerConvectionTerm**(*coeff=1.0, var=None*)

Bases: *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)
```

(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.]]), 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** (*MeshVariable*) – The *Term*'s coefficient value.**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.

__eq__(other)

Return self==value.

__hash__()

Return hash(self).

__mul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

__neg__()

Negate a *Term*.

```
>>> -__NonDiffusionTerm(coeff=1.)
__NonDiffusionTerm(coeff=-1.0)
```

__repr__()

The representation of a *Term* object is given by,

```
>>> print(__UnaryTerm(123.456))
__UnaryTerm(coeff=123.456)
```

__rmul__(other)

Multiply a term

```
>>> 2. * __NonDiffusionTerm(coeff=0.5)
__NonDiffusionTerm(coeff=1.0)
```

Test for ticket:291.

```
>>> from fipy import PowerLawConvectionTerm
>>> PowerLawConvectionTerm(coeff=[[1], [0]]) * 1.0
PowerLawConvectionTerm(coeff=array([[ 1.],
[ 0.])))
```

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.

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.

Returns

error – The residual vector $\vec{e} = L\vec{x}_{\text{old}} - \vec{b}$

Return type

CellVariable

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.

Returns

residual – The residual vector $\vec{r} = L\vec{x} - \vec{b}$

Return type

CellVariable

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.

Returns

- **residual** (*CellVariable*) – The residual vector $\vec{r} = L\vec{x} - \vec{b}$
- **norm** (float) – The L2 norm of *residual*, $\|\vec{r}\|_2$

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*

Returns

residual – The residual vector $\vec{r} = \mathbf{L}\vec{x} - \vec{b}$

Return type

CellVariable

23.7 fipy.testFiPy

Test suite for *FiPy* modules

23.8 fipy.tests

Unit testing scripts

Modules

<code>fipy.tests.doctestPlus</code>	
<code>fipy.tests.lateImportTest</code>	Classes to enable accumulating tests without importing them
<code>fipy.tests.test</code>	
<code>fipy.tests.testProgram</code>	

23.8.1 fipy.tests.doctestPlus

Functions

<code>execButNoTest([name])</code>	Execute the doctests in the module without testing.
<code>register_skipper(flag, test, why[, skipWarning])</code>	Create a new doctest option flag for skipping tests
<code>report_skips()</code>	Print out how many doctest examples were skipped due to flags
<code>testmod([m, name, globs, verbose, report, ...])</code>	Test examples in the given module.

`fipy.tests.doctestPlus.execButNoTest(name='__main__')`

Execute the doctests in the module without testing.

`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.

23.8.2 fipy.tests.lateImportTest

Classes to enable accumulating tests without importing them

Prevent failure to import one test from stopping execution of other tests.

23.8.3 fipy.tests.test

Functions

```
main([module_or_suite, inline, ...])
```

Classes

<code>DeprecationErroringTestProgram([module, ...])</code>	<i>TestProgram</i> that overrides inability of standard <i>TestProgram</i> to throw errors on <i>DeprecationWarning</i>
<code>TestCommand(module_or_suite, inline, ...)</code>	

23.8.4 fipy.tests.testProgram

Classes

<code>TestProgram([module, defaultTest, argv, ...])</code>	A command-line program that runs a set of tests
<code>main</code>	alias of <i>TestProgram</i>

```
class fipy.tests.testProgram.TestProgram(module='__main__', defaultTest=None, argv=None,
                                           testRunner=None, testLoader=<unittest.loader.TestLoader
                                           object>, exit=True, verbosity=1, failfast=None,
                                           catchbreak=None, buffer=None, warnings=None, *,
                                           tb_locals=False)
```

Bases: *TestProgram*

A command-line program that runs a set of tests

This is primarily for making test modules conveniently executable.

`fipy.tests.testProgram.main`

alias of *TestProgram*

23.9 fipy.tools

Utility modules, functions, and values

`fipy.tools.serialComm`: [*CommWrapper*](#)

Serial MPI communicator when running in parallel.

`fipy.tools.parallelComm`: [*CommWrapper*](#)

Parallel MPI communicator when running in parallel.

class `fipy.tools.PhysicalField`(*value*, *unit=None*, *array=None*)

Bases: `object`

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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)`

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples 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?

__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).

__le__(other)

Return self<=value.

__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
```

__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?

__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
```

__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 *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
```

`__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 *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
```

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:

$$| \text{self} - \text{other} | < \text{atol} + \text{rtol} * | \text{other} |$$

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):
```

(continues on next page)

(continued from previous page)

```
...
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
```

property dtype

Returns the NumPy sctype of the underlying array.

```
>>> issubclass(PhysicalField(1, 'm').dtype.type, numerix.integer)
True
>>> issubclass(PhysicalField(1., 'm').dtype.type, numerix.floating)
```

(continues on next page)

(continued from previous page)

```
True
>>> issubclass(PhysicalField((1, 1.), 'm').dtype.type, numerix.floating)
True
```

floor()

Return the largest integer less than or equal to the *PhysicalField*.

```
>>> print(PhysicalField(2.2, "m").floor())
2.0 m
```

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')
>>> 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
```

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 *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
```

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
```

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>
```

fipy.tools.SharedTemporaryFile(*mode*='w+b', *buffering*=-1, *encoding*=None, *newline*=None, *suffix*="", *prefix*='tmp', *dir*=None, *delete*=True, *communicator*=DummyComm())

Create a temporary file shared by all MPI ranks.

The file is created as *NamedTemporaryFile* would do it. The name of the returned file-like object is accessible as its *name* attribute. The file will be automatically deleted when it is closed unless the *delete* argument is set to False.

```
>>> from fipy.tools import SharedTemporaryFile, parallelComm
>>> with SharedTemporaryFile(mode='w+', suffix=".tmp") as tmpFile:
...     # write on processor 0
...     if parallelComm.procID == 0:
...         _ = tmpFile.write("shared text")
...
...     parallelComm.Barrier()
...
...     # read on all processors
...     _ = tmpFile.seek(0)
...     txt = tmpFile.read()
>>> print(txt)
shared text
```

Parameters

- **prefix** (*str*) – As for mkstemp
- **suffix** (*str*) – As for mkstemp
- **dir** (*str*) – As for mkstemp
- **mode** (*str*) – The mode argument to io.open (default “w+b”)
- **buffering** (*int*) – The buffer size argument to io.open (default -1)
- **encoding** (*str* or None) – The encoding argument to io.open (default None)
- **newline** (*str* or None) – The newline argument to io.open (default None)
- **delete** (*bool*) – Whether the file is deleted on close (default True)

- **communicator** (*CommWrapper*) – MPI communicator describing ranks to share with. A duck-typed object with *procID* and *Nproc* attributes is sufficient.

Return type

file-like object

See also:`tempfile.NamedTemporaryFile`, `tempfile.mkstemp`, `io.open`**Modules**

<code>fiPy.tools.comms</code>	
<code>fiPy.tools.copy_script</code>	
<code>fiPy.tools.debug</code>	
<code>fiPy.tools.decorators</code>	
<code>fiPy.tools.dimensions</code>	
<code>fiPy.tools.dump</code>	
<code>fiPy.tools.inline</code>	
<code>fiPy.tools.logging</code>	
<code>fiPy.tools.numerix</code>	Replacement module for NumPy
<code>fiPy.tools.parser</code>	
<code>fiPy.tools.sharedtempfile</code>	This module provides a generic, high-level interface for creating shared temporary files.
<code>fiPy.tools.test</code>	
<code>fiPy.tools.timer</code>	
<code>fiPy.tools.vector</code>	Vector utility functions that are inexplicably absent from Numeric

23.9.1 fipy.tools.comms

Modules

<code>fiPy.tools.comms.commWrapper</code>
<code>fiPy.tools.comms.dummyComm</code>

fipy.tools.comms.commWrapper**Classes***CommWrapper()*

MPI Communicator wrapper

class fipy.tools.comms.commWrapper.**CommWrapper**Bases: *object*

MPI Communicator wrapper

Encapsulates capabilities needed for possibly parallel operations. Some capabilities are not parallel.

__getstate__()

Helper for pickle.

__repr__()

Return repr(self).

fipy.tools.comms.dummyComm**Classes***DummyComm()***class** fipy.tools.comms.dummyComm.**DummyComm**Bases: *CommWrapper***__getstate__()**

Helper for pickle.

__repr__()

Return repr(self).

23.9.2 fipy.tools.copy_script**Functions***copy_script(copy_from, copy_to)**import_from_path(module_name, file_path)*

23.9.3 fipy.tools.debug

Functions

<code>PRINT(label[, arg, stall])</code>	Display <i>label</i> and <i>arg</i> on each MPI rank.

`fipy.tools.debug.PRINT(label, arg="", stall=True)`

Display *label* and *arg* on each MPI rank.

Annotate with rank number. If *stall* is true, ensure all ranks print before proceeding.

23.9.4 fipy.tools.decorators

Functions

<code>deprecate(*args, **kwargs)</code>	Issues a generic <i>DeprecationWarning</i> .

`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 doc-string after the warning.

Returns

old_func – The deprecated function.

Return type

function

23.9.5 fipy.tools.dimensions

Modules

<code>fipy.tools.dimensions.DictWithDefault</code>	
<code>fipy.tools.dimensions.NumberDict</code>	
<code>fipy.tools.dimensions.physicalField</code>	Physical quantities with units.

fipy.tools.dimensions.DictWithDefault**fipy.tools.dimensions.NumberDict****fipy.tools.dimensions.physicalField**

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
```

(continues on next page)

(continued from previous page)

```

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
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 = 0.00378541209600000006 m**3
1 gn = 9.80665 m/s**2
1 Grav = 6.6742e-11 m**3/kg/s**2
1 h = 3600.0 s

```

(continues on next page)

(continued from previous page)

```

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 tbsp = 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
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

```

Classes

<i>PhysicalField</i> (value[, unit, array])	Field or quantity with units.
<i>PhysicalUnit</i> (names, factor, powers[, offset])	The units of a <i>PhysicalField</i> .

class fipy.tools.dimensions.physicalField.**PhysicalField**(value, unit=None, array=None)

Bases: `object`

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'))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: Incompatible units
```

__array__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples 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?

__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).

__le__(other)

Return self<=value.

__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
```

__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?

__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
```

__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 *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
```

__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 *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
```

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:

$$| \text{self} - \text{other} | < \text{atol} + \text{rtol} * | \text{other} |$$

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 *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
```

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
```

property dtype

Returns the NumPy sctype of the underlying array.

```
>>> isinstance(PhysicalField(1, 'm').dtype.type, numerix.integer)
True
>>> isinstance(PhysicalField(1., 'm').dtype.type, numerix.floating)
True
>>> isinstance(PhysicalField((1, 1.), 'm').dtype.type, numerix.floating)
True
```

floor()

Return the largest integer less than or equal to the *PhysicalField*.

```
>>> print(PhysicalField(2.2, "m").floor())
2.0 m
```

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')
>>> 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
```

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 *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
```

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
```

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`

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.

__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")
```

(continues on next page)

(continued from previous page)

```
>>> 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

__le__(other)

Return self<=value.

__lt__(other)

Return self<value.

__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):
...
```

(continues on next page)

(continued from previous page)

```

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>
```

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
>>> [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]))
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>
```

23.9.6 fipy.tools.dump

Functions

<code>read(filename[, fileobject, communicator, ...])</code>	Read a pickled object from a file.
<code>write(data[, filename, extension, communicator])</code>	Pickle an object and write it to a file.

`fipy.tools.dump.read(filename, fileobject=None, communicator=DummyComm(), 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. If the filename extension is *.gz*, the file is first decompressed.
- **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)

`fipy.tools.dump.write(data, filename=None, extension="", communicator=DummyComm())`

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. If the filename ends in *.gz*, the file is automatically saved in compressed *gzip* format.
- **extension** (*str*) – File extension to append if *filename* is not given. If set to *.gz*, the file is automatically saved in compressed *gzip* format.
- **communicator** (*CommWrapper*) – A duck-typed object with *procID* and *Nproc* attributes is sufficient

23.9.7 fipy.tools.inline

23.9.8 fipy.tools.logging

Modules

fipy.tools.logging.environment

fipy.tools.logging.environment**Functions**

<code>conda_info([conda])</code>	Collect information about conda environment.
<code>nix_info()</code>	Collect information about nix environment.
<code>package_info()</code>	Collect information about installed packages FiPy uses.
<code>pip_info([python])</code>	Collect information about pip environment.
<code>platform_info()</code>	Collect information about platform Python is running in.

`fipy.tools.logging.environment.conda_info(conda='conda')`

Collect information about conda environment.

Parameters

conda (`str`) – Name of conda executable (default: “conda”).

Returns

Result of *conda info* and *conda env export* for active conda environment.

Return type

`dict`

`fipy.tools.logging.environment.nix_info()`

Collect information about nix environment.

Returns

Result of:

```
nix derivation show .#fipy
```

Return type

`dict`

`fipy.tools.logging.environment.package_info()`

Collect information about installed packages FiPy uses.

Returns

Versions of important Python packages.

Return type

`dict`

`fipy.tools.logging.environment.pip_info(python='python')`

Collect information about pip environment.

Parameters

python (`str`) – Name of Python executable (default: “python”).

Returns

Result of *pip list --format json*.

Return type

`list of dict`

`fipy.tools.logging.environment.platform_info()`

Collect information about platform Python is running in.

Returns

Data extracted from *platform* package.

Return type

`dict`

23.9.9 fipy.tools.numerix

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.]
```

Functions

<code>L1norm(arr)</code>	Taxicab or Manhattan norm of <i>arr</i>
<code>L2norm(arr)</code>	Euclidean norm of <i>arr</i>
<code>LINFnorm(arr)</code>	Infinity norm of <i>arr</i>
<code>all(a[, axis, out])</code>	Test whether all array elements along a given axis evaluate to True.
<code>allclose(first, second[, rtol, atol])</code>	Tests whether or not <i>first</i> and <i>second</i> are equal, subject to the given relative and absolute tolerances, such that.
<code>allequal(first, second)</code>	Returns <i>true</i> if every element of <i>first</i> is equal to the corresponding element of <i>second</i> .
<code>dot(a1, a2[, axis])</code>	return array of vector dot-products of <i>v1</i> and <i>v2</i> for arrays <i>a1</i> and <i>a2</i> of vectors <i>v1</i> and <i>v2</i>
<code>getShape(arr)</code>	Return the shape of <i>arr</i>
<code>getUnit(arr)</code>	Return the unit of <i>arr</i> .
<code>isclose(first, second[, rtol, atol])</code>	Returns which elements of <i>first</i> and <i>second</i> are equal, subject to the given relative and absolute tolerances, such that.
<code>nearest(data, points[, max_mem])</code>	find the indices of <i>data</i> that are closest to <i>points</i>
<code>put(arr, ids, values)</code>	The opposite of <i>take</i> .
<code>rank(a)</code>	Get the rank of sequence <i>a</i> (the number of dimensions, not a matrix rank) The rank of a scalar is zero.
<code>reshape(arr, shape)</code>	Change the shape of <i>arr</i> to <i>shape</i> , as long as the product of all the lengths of all the axes is constant (the total number of elements does not change).
<code>sqrtdot(a1, a2)</code>	Return array of square roots of vector dot-products for arrays <i>a1</i> and <i>a2</i> of vectors <i>v1</i> and <i>v2</i>
<code>sum(arr[, axis])</code>	The sum of all the elements of <i>arr</i> along the specified axis.
<code>take(a, indices[, axis, fill_value])</code>	Selects the elements of <i>a</i> corresponding to <i>indices</i> .
<code>tostring(arr[, max_line_width, precision, ...])</code>	Returns a textual representation of a number or field of numbers.

`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.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:

$$|\text{first} - \text{second}| < \text{atol} + \text{rtol} * |\text{second}|$$

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 vectorsTest 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)`

Return the unit of *arr*.

If *arr* has no units, returns 1.

`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.nearest(data, points, max_mem=100000000.0)`

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, '1')
>>> 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))
```

(continues on next page)

(continued from previous page)

```

[ 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*

23.9.10 fipy.tools.parser**Functions***parse*(*larg*[, *action*, *type*, *default*])This is a wrapper function for the python *optparse* 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

23.9.11 fipy.tools.sharedtempfile

This module provides a generic, high-level interface for creating shared temporary files. All of the interfaces provided by this module can be used without fear of race conditions.

Functions

<i>SharedTemporaryFile</i> ([mode, buffering, ...])	Create a temporary file shared by all MPI ranks.

`fipy.tools.sharedtempfile.SharedTemporaryFile(mode='w+b', buffering=-1, encoding=None, newline=None, suffix='', prefix='tmp', dir=None, delete=True, communicator=DummyComm())`

Create a temporary file shared by all MPI ranks.

The file is created as *NamedTemporaryFile* would do it. The name of the returned file-like object is accessible as its **name** attribute. The file will be automatically deleted when it is closed unless the *delete* argument is set to *False*.

```
>>> from fipy.tools import SharedTemporaryFile, parallelComm
>>> with SharedTemporaryFile(mode='w+', suffix=".tmp") as tmpFile:
...     # write on processor 0
...     if parallelComm.procID == 0:
...         _ = tmpFile.write("shared text")
...
...     parallelComm.Barrier()
...
...     # read on all processors
...     _ = tmpFile.seek(0)
...     txt = tmpFile.read()
>>> print(txt)
shared text
```

Parameters

- **prefix** (*str*) – As for `mkstemp`
- **suffix** (*str*) – As for `mkstemp`
- **dir** (*str*) – As for `mkstemp`
- **mode** (*str*) – The mode argument to `io.open` (default “w+b”)
- **buffering** (*int*) – The buffer size argument to `io.open` (default -1)
- **encoding** (*str* or *None*) – The encoding argument to `io.open` (default *None*)
- **newline** (*str* or *None*) – The newline argument to `io.open` (default *None*)
- **delete** (*bool*) – Whether the file is deleted on close (default *True*)
- **communicator** (*CommWrapper*) – MPI communicator describing ranks to share with. A duck-typed object with *procID* and *Nproc* attributes is sufficient.

Return type

file-like object

See also:`tempfile.NamedTemporaryFile`, `tempfile.mkstemp`, `io.open`

23.9.12 fipy.tools.test

23.9.13 fipy.tools.timer

Classes

<code>Timer([timer])</code>	Context manager that measures time elapsed in context

class `fipy.tools.timer.Timer(timer=None)`Bases: `object`

Context manager that measures time elapsed in context

Defaults to nanosecond precision (although probably only microsecond or even millisecond accuracy).

```
>>> with Timer() as timer:
...     pass
>>> print("elapsed: {elapsed} ns".format(elapsed=timer.elapsed))
elapsed: ... ns
```

Parameters**timer** (callable, *optional*) – Function that returns a time`timer()` -> int or floatThe difference between successive calls to `timer()` should give the elapsed time at the desired resolution and type. (default: `perf_counter_ns()`)**static clock_ns()**

Substitute “nanosecond” timer

property elapsed

Time measured so far

23.9.14 fipy.tools.vector

Vector utility functions that are inexplicably absent from Numeric

Functions

<code>prune(array, shift[, start, axis])</code>	removes elements with indices $i = \text{start} + \text{shift} * n$ where $n = 0, 1, 2, \dots$
<code>putAdd(vector, ids, additionVector)</code>	This is a temporary replacement for <i>Numeric.put</i> 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])
```

`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.

23.10 fipy.variables

Collections of values supporting lazy evaluation

Modules

<code>fipy.variables.addOverFacesVariable</code>
<code>fipy.variables.arithmeticCellToFaceVariable</code>
<code>fipy.variables.betaNoiseVariable</code>
<code>fipy.variables.binaryOperatorVariable</code>
<code>fipy.variables.cellToFaceVariable</code>
<code>fipy.variables.cellVariable</code>
<code>fipy.variables.constant</code>
<code>fipy.variables.constraintMask</code>
<code>fipy.variables.coupledCellVariable</code>
<code>fipy.variables.distanceVariable</code>

continues on next page

Table 3 – continued from previous page

<code>fipy.variables.exponentialNoiseVariable</code>	
<code>fipy.variables.faceGradContributionsVariable</code>	
<code>fipy.variables.faceGradVariable</code>	
<code>fipy.variables.faceVariable</code>	
<code>fipy.variables.gammaNoiseVariable</code>	
<code>fipy.variables.gaussCellGradVariable</code>	
<code>fipy.variables.gaussianNoiseVariable</code>	
<code>fipy.variables.harmonicCellToFaceVariable</code>	
<code>fipy.variables.histogramVariable</code>	
<code>fipy.variables.interfaceAreaVariable</code>	
<code>fipy.variables.interfaceFlagVariable</code>	
<code>fipy.variables.leastSquaresCellGradVariable</code>	
<code>fipy.variables.levelSetDiffusionVariable</code>	
<code>fipy.variables.meshVariable</code>	
<code>fipy.variables.minmodCellToFaceVariable</code>	
<code>fipy.variables.modCellGradVariable</code>	
<code>fipy.variables.modCellToFaceVariable</code>	
<code>fipy.variables.modFaceGradVariable</code>	
<code>fipy.variables.modPhysicalField</code>	
<code>fipy.variables.modularVariable</code>	
<code>fipy.variables.noiseVariable</code>	
<code>fipy.variables.operatorVariable</code>	
<code>fipy.variables.scharfetterGummelFaceVariable</code>	
<code>fipy.variables.surfactantConvectionVariable</code>	
<code>fipy.variables.surfactantVariable</code>	
<code>fipy.variables.test</code>	Test numeric implementation of the mesh

continues on next page

Table 3 – continued from previous page

*fiPy.variables.unaryOperatorVariable**fiPy.variables.uniformNoiseVariable**fiPy.variables.variable*

23.10.1 *fiPy.variables.addOverFacesVariable*

23.10.2 *fiPy.variables.arithmeticCellToFaceVariable*

23.10.3 *fiPy.variables.betaNoiseVariable*

Classes

BetaNoiseVariable(*args, **kwargs)

Represents a beta distribution of random numbers with the probability distribution

class *fiPy.variables.betaNoiseVariable.BetaNoiseVariable*(*args, **kwargs)Bases: *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

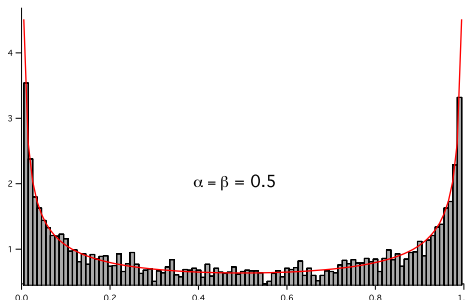
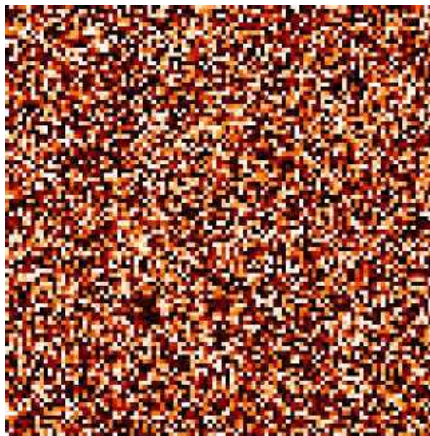

```
>>> from fipy.variables.cellVariable import CellVariable
>>> betadist = CellVariable(mesh = histogram.mesh)
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> betadist = ((Gamma(alpha + beta) / (Gamma(alpha) * Gamma(beta)))
...             * 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 .

__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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__(*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

__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

```

__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 *CellVariable* 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).

__invert__()

Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

__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(10000000000000000000 * 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)
```

__ne__(other)

Test if a *Variable* is not equal to another quantity

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

(continues on next page)

(continued from previous page)

```
(Variable(value=array(3)) != 4)
>>> b()
1
```

static `__new__(cls, *args, **kws)`

__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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(dict)

Used internally to create a new *CellVariable* from pickled persistent storage.

__str__()

Return str(self).

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
```

any(*axis*=None)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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.]
```

property `constraintMask`

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]
```

property `copy()`

Copy the value of the *NoiseVariable* to a static *CellVariable*.

property `dot(other, opShape=None, operatorClass=None)`

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

property `dtype`

Returns the Numpy *dtype* of the underlying array.

```

>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True

```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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 $\frac{1}{V_p} \sum_f \vec{n}_f \phi_f A_f$ 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
```

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')
>>> 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
```

property leastSquaresGrad

Return $\nabla\phi$, which is determined by solving for $\nabla\phi$ in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n}_{AP} \cdot \nabla\phi)_{AP}$$

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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]
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$$\text{other} \cdot \text{self}$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

release(*constraint*)

Remove *constraint* from *self*

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
```

(continues on next page)

(continued from previous page)

```

>>> 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]

```

scramble()

Generate a new random distribution.

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

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))

```

(continues on next page)

(continued from previous page)

```
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()*2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```

23.10.4 fipy.variables.binaryOperatorVariable

23.10.5 fipy.variables.cellToFaceVariable

23.10.6 fipy.variables.cellVariable

Classes

<code>CellVariable(*args, **kwargs)</code>	Represents the field of values of a variable on a <i>Mesh</i> .

class fipy.variables.cellVariable.**CellVariable**(*args, **kwargs)

Bases: *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

`__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
```

Check representation works with different versions of numpy


```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)`

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))))
```

(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()`**__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

__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
```

__ge__(*other*)Test if a *Variable* is greater than or equal to another quantity

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

(continues on next page)

(continued from previous page)

```

>>> 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 *CellVariable* 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).

__invert__()

Returns logical “not” of the *Variable*

```

>>> a = Variable(value=True)
>>> print(~a)
False

```

__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

```

__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(10000000000000000000 * 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)

```

__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

```

static __new__(cls, *args, **kwargs)**__nonzero__()**

```

>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...
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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(*dict*)

Used internally to create a new *CellVariable* from pickled persistent storage.

__str__()

Return str(self).

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
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

any(axis=None)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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.]
```

property **constraintMask**

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *

>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]
```

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]))
```


dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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
```

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')
>>> 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
```

property leastSquaresGrad

Return $\nabla\phi$, which is determined by solving for $\nabla\phi$ in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n}_{AP} \cdot \nabla\phi)_{AP}$$

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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]
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$$\text{other} \cdot \text{self}$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

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

```

property shape

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True

```

(continues on next page)

(continued from previous page)

```
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()*2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```

23.10.7 fipy.variables.constant

23.10.8 fipy.variables.constraintMask

23.10.9 fipy.variables.coupledCellVariable

23.10.10 fipy.variables.distanceVariable

Classes

DistanceVariable(*args, **kws)*A DistanceVariable object calculates ϕ so it satisfies,*

class fipy.variables.distanceVariable.DistanceVariable(*args, **kws)Bases: *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.

__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

__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__ (*dtype=None, copy=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_wrap__ (*arr, context=None, return_scalar=False*)

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__(*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

__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

```

__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 *CellVariable* 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).

__invert__()

Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

__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(10000000000000000000 * 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)
```

__ne__(other)

Test if a *Variable* is not equal to another quantity

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

(continues on next page)

(continued from previous page)

```
(Variable(value=array(3)) != 4)
>>> b()
1
```

static `__new__(cls, *args, **kws)`

__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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(dict)

Used internally to create a new *CellVariable* from pickled persistent storage.

__str__()

Return str(self).

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
```

any(*axis*=None)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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
```

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

```

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)

```

(continues on next page)

(continued from previous page)

```

[[-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.]

```

property `constraintMask`

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```

>>> from fipy import *

>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]

```

method `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]))
```

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True
```

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.

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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]
```

(continues on next page)

(continued from previous page)

```
>>> 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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
```

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')
>>> 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
```

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, \
```

(continues on next page)

(continued from previous page)

```
... [[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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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))
```

(continues on next page)

(continued from previous page)

```

>>> 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]

```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{other} \cdot \text{self}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

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]

```

(continues on next page)

(continued from previous page)

```
>>> 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

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(*axis=None*, *kwargs*)**

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()**2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.


```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```

23.10.11 fipy.variables.exponentialNoiseVariable**Classes**

<i>ExponentialNoiseVariable</i> (*args, **kws)	Represents an exponential distribution of random numbers with the probability distribution

class fipy.variables.exponentialNoiseVariable.**ExponentialNoiseVariable**(*args, **kws)

Bases: *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

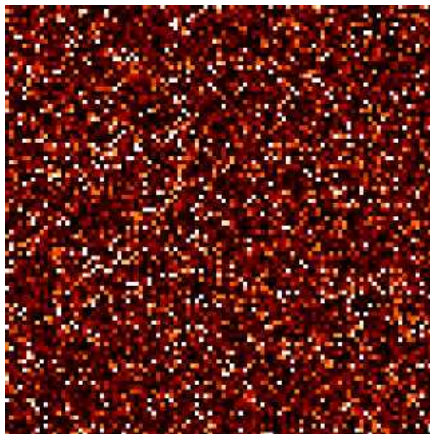
```
>>> from fipy.variables.cellVariable import CellVariable
>>> expdist = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
```

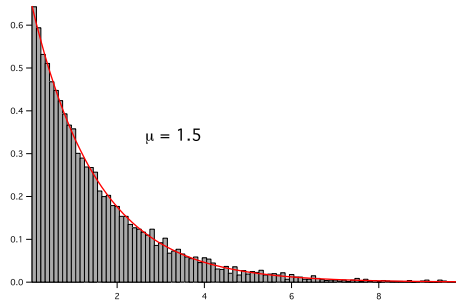
```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=5)
...     histoplot = Viewer(vars=(histogram, expdist),
...                           datamin=0, datamax=1.5)
```

```
>>> 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 .

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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__(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

__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
```

__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 *CellVariable* 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())
```

(continues on next page)

(continued from previous page)

```

0
>>> a.value = 5
>>> print(b())
1

```

__hash__()

Return hash(self).

__invert__()Returns logical “not” of the *Variable*

```

>>> a = Variable(value=True)
>>> print(~a)
False

```

__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

```

__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(10000000000000000000 * 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)

```

__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
```

static __new__(cls, *args, **kws)**__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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(dict)Used internally to create a new *CellVariable* from pickled persistent storage.

__str__()

Return str(self).

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
```

any(axis=None)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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
```

(continues on next page)

(continued from previous page)

```
>>> 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 cellVolumeAverageReturn 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.]
```

property constraintMask

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False True False False True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False True False True True False True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False True]
>>> print(v1)
[ 1.  0.  0.  3.]
```

copy()

Copy the value of the *NoiseVariable* to a static *CellVariable*.

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

`self · other`

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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
```

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')
>>> 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
```

property leastSquaresGrad

Return $\nabla\phi$, which is determined by solving for $\nabla\phi$ in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n}_{AP} \cdot \nabla\phi)_{AP}$$

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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]
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$$\text{other} \cdot \text{self}$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

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]

```

scramble()

Generate a new random distribution.

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

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))

```

(continues on next page)

(continued from previous page)

```
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()*2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```


23.10.12 fipy.variables.faceGradContributionsVariable

23.10.13 fipy.variables.faceGradVariable

23.10.14 fipy.variables.faceVariable

Classes

```
FaceVariable(*args, **kwargs)
```

param mesh

the mesh that defines the geometry of this *Variable*

```
class fipy.variables.faceVariable.FaceVariable(*args, **kwargs)
```

Bases: *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

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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)
```

(continues on next page)

(continued from previous page)

```
>>> 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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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)
```

(continues on next page)

(continued from previous page)

```
>>> b()
7
```

__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
```

__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 *MeshVariable* to persistent storage.

__gt__(other)

Test if a *Variable* is greater than another quantity

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

(continues on next page)

(continued from previous page)

```
(Variable(value=array(3)) > 4)
>>> print(b())
0
>>> a.value = 5
>>> print(b())
1
```

__hash__()

Return hash(self).

__invert__()Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

__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)
```

__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
```

static __new__(cls, *args, **kwargs)**__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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(*dict*)Used internally to create a new *Variable* from pickled persistent storage.**__str__**()

Return str(self).

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(100000))
>>> print(var.allclose(numerix.zeros(100000, 'l')))
False
```

any(*axis=None*)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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 constraintMask

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
```

(continues on next page)

(continued from previous page)

```
>>> print(v1)
[ 1.  0.  0.  3.]
```

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)
>>> var = CellVariable(mesh=mesh, value=list(range(3*2)))
>>> print(var.faceGrad.divergence)
[ 4.  3.  2. -2. -3. -4.]
```

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

self · *other*

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
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')
>>> 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
```

property mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{other} \cdot \text{self}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

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

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True

```

Have to account for zero length arrays

```

>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True

```

std(*axis=None, **kwargs*)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```

>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()2).allclose(1.25))
True

```

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

```

23.10.15 fipy.variables.gammaNoiseVariable

Classes

<code>GammaNoiseVariable(*args, **kws)</code>	Represents a gamma distribution of random numbers with the probability distribution

class `fipy.variables.gammaNoiseVariable.GammaNoiseVariable(*args, **kws)`

Bases: `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

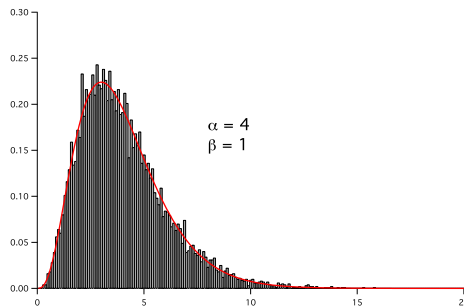
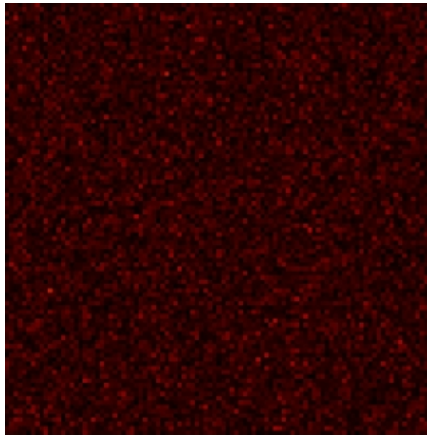
```
>>> from fipy.variables.cellVariable import CellVariable
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> from fipy.tools.numerix import exp
>>> gammadist = (x**(alpha - 1) * (beta**alpha * exp(-beta * x)) / Gamma(alpha))
```

```
>>> 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()
...             histplot.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 .

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)`

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))))
```

(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()`**__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

__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
```

__ge__(*other*)Test if a *Variable* is greater than or equal to another quantity

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

(continues on next page)

(continued from previous page)

```

>>> 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 *CellVariable* 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).

__invert__()

Returns logical “not” of the *Variable*

```

>>> a = Variable(value=True)
>>> print(~a)
False

```

__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

```

__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(10000000000000000000 * 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)

```

__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

```

static __new__(cls, *args, **kws)**__nonzero__()**

```

>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...
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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(*dict*)

Used internally to create a new *CellVariable* from pickled persistent storage.

__str__()

Return str(self).

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
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

any(axis=None)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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.]

```

property *constraintMask*

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]
```

copy()

Copy the value of the *NoiseVariable* to a static *CellVariable*.

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

self · *other*

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1)).dtype.type, numerix.floating)
True
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

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')
>>> 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
```

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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]
```

(continues on next page)

(continued from previous page)

```
>>> 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]
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{other} \cdot \text{self}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

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]
```

scramble()

Generate a new random distribution.

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

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
```

(continues on next page)

(continued from previous page)

```
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()*2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```

23.10.16 fipy.variables.gaussCellGradVariable

23.10.17 fipy.variables.gaussianNoiseVariable

Classes

`GaussianNoiseVariable(*args, **kwargs)`

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

class fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable(*args, **kwargs)

Bases: `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:

```
sigmaSqrd = Mobility * kBoltzmann * Temperature / (mesh.cellVolumes * timeStep)
GaussianNoiseVariable(mesh = mesh, variance = sigmaSqrd)
```

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)
```

and compare to a Gaussian distribution

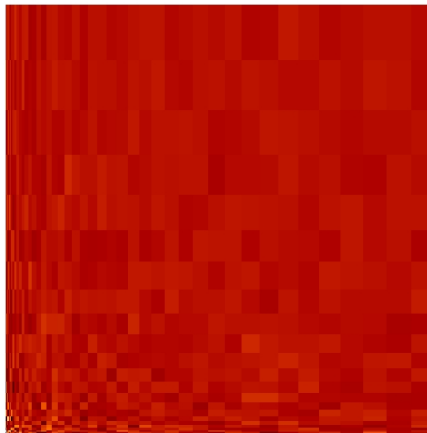
```
>>> gauss = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
>>> gauss.value = ((1/(sqrt(variance * 2 * pi))) * exp(-(x - mean)**2 / (2 *
↪ variance))))
```

```
>>> if __name__ == '__main__':
...     from fipy.viewers import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=-5, datamax=5)
...     histoplot = Viewer(vars=(histogram, gauss))
```

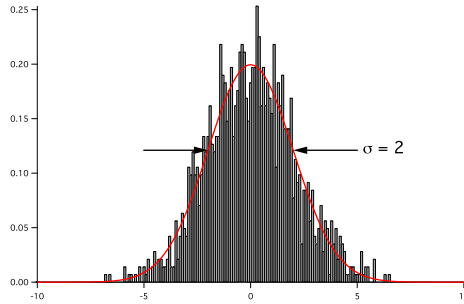
```
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()
```

```
>>> 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 .

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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__(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

`__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
```

`__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 *CellVariable* to persistent storage.

`__gt__(other)`

Test if a *Variable* is greater than another quantity

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

(continues on next page)

(continued from previous page)

```
(Variable(value=array(3)) > 4)
>>> print(b())
0
>>> a.value = 5
>>> print(b())
1
```

__hash__()

Return hash(self).

__invert__()Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

__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)
```

__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
```

static __new__(cls, *args, **kwargs)**__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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(dict)

Used internally to create a new *CellVariable* from pickled persistent storage.

__str__()

Return str(self).

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
```

any(axis=None)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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)
```

(continues on next page)

(continued from previous page)

```

>>> 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.]

```

property constraintMask

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```

>>> from fipy import *

```

```

>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]

```

copy()

Copy the value of the *NoiseVariable* to a static *CellVariable*.

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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
```

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')
>>> 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
```

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```


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]
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$$\text{other} \cdot \text{self}$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

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]

```

scramble()

Generate a new random distribution.

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

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))

```

(continues on next page)

(continued from previous page)

```
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()*2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```

23.10.18 fipy.variables.harmonicCellToFaceVariable

23.10.19 fipy.variables.histogramVariable

Classes

<code>HistogramVariable(*args, **kwargs)</code>	Produces a histogram of the values of the supplied distribution.

class fipy.variables.histogramVariable.HistogramVariable(*args, **kwargs)

Bases: `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

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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__` (*dtype=None, copy=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_wrap__` (*arr, context=None, return_scalar=False*)

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__` (*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
```

(continues on next page)

(continued from previous page)

```

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

`__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

```

`__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 *CellVariable* 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).

__invert__()

Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

__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
```

(continues on next page)

(continued from previous page)

```
>>> print(10000000000000000000 * 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)
```

`__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
```

`static __new__(cls, *args, **kws)`

`__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]
```

`__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>
```

__repr__()

Return repr(self).

__setstate__(dict)

Used internally to create a new *CellVariable* from pickled persistent storage.

__str__()

Return str(self).

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
```

any(axis=None)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```

[[-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.]

```

property constraintMask

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```

>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)

```

(continues on next page)

(continued from previous page)

```
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]
```

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]))
```

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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
```

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')
>>> 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
```

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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)
```

(continues on next page)

(continued from previous page)

```

>>> 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]

```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{other} \cdot \text{self}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

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)

```

(continues on next page)

(continued from previous page)

```
[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

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(*axis=None*, ***kwargs*)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()**2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```

23.10.20 fipy.variables.interfaceAreaVariable

23.10.21 fipy.variables.interfaceFlagVariable

23.10.22 fipy.variables.leastSquaresCellGradVariable

23.10.23 fipy.variables.levelSetDiffusionVariable

23.10.24 fipy.variables.meshVariable

Classes

MeshVariable(*args, **kwargs)

Abstract base class for a *Variable* that is defined on a mesh

class fipy.variables.meshVariable.**MeshVariable**(*args, **kwargs)

Bases: *Variable*

Abstract base class for a *Variable* that is defined on a mesh

Attention: This class is abstract. Always create one of its subclasses.

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

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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
```

__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
```

__ge__(other)

Test if a *Variable* is greater than or equal to another quantity

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

(continues on next page)

(continued from previous page)

```

>>> 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 *MeshVariable* 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).

__invert__()

Returns logical “not” of the *Variable*

```

>>> a = Variable(value=True)
>>> print(~a)
False

```

__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

```

__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(10000000000000000000 * 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)

```

__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

```

static __new__(cls, *args, **kwargs)**__nonzero__()**

```

>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...
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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(*dict*)

Used internally to create a new *Variable* from pickled persistent storage.

__str__()

Return str(self).

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
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

any(*axis=None*)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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 constraintMask

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]
```

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]))

```

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

property dtype

Returns the Numpy *dtype* of the underlying array.

```

>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
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 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')
>>> 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

```

property mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(*axis=None*)

Return the maximum along a given axis.

min(*axis=None*)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

rdot(*other, opShape=None, operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{other} \cdot \text{self}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

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]
```

(continues on next page)

(continued from previous page)

```
>>> 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

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(*axis=None*, *kwargs*)**

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()2).allclose(1.25))
True
```

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
```

23.10.25 fipy.variables.minmodCellToFaceVariable

23.10.26 fipy.variables.modCellGradVariable

23.10.27 fipy.variables.modCellToFaceVariable

23.10.28 fipy.variables.modFaceGradVariable

23.10.29 fipy.variables.modPhysicalField

23.10.30 fipy.variables.modularVariable

Classes

ModularVariable(*args, **kwargs)

The *ModularVariable* defines a variable that exists on the circle between $-\pi$ and π

class fipy.variables.modularVariable.**ModularVariable**(*args, **kwargs)

Bases: *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

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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)
```

(continues on next page)

(continued from previous page)

```
>>> 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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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__(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.))))
```

(continues on next page)

(continued from previous page)

```
[ 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

__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
```

__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 *CellVariable* 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).

`__invert__()`

Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

`__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
```

__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)

```

__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

```

static __new__(cls, *args, **kwargs)**__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

```

(continues on next page)

(continued from previous page)

```

>>> 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]

```

__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>

```

__repr__()

Return repr(self).

__setstate__(*dict*)Used internally to create a new *CellVariable* from pickled persistent storage.**__str__**()

Return str(self).

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(100000))
>>> print(var.allclose(numerix.zeros(10000, 'l')))
False

```

any(*axis=None*)

```

>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1

```

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 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.]
```

property `constraintMask`

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]
```

method `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]))
```

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True
```

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 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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). Adjusted for a *ModularVariable*

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

```

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')
>>> 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
```

property leastSquaresGrad

Return $\nabla\phi$, which is determined by solving for $\nabla\phi$ in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n}_{AP} \cdot \nabla\phi)_{AP}$$

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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]
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{other} \cdot \text{self}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

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
```

property shape

```
>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()**2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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
```

(continues on next page)

(continued from previous page)

```
>>> answer = CellVariable(mesh=mesh, value=1.)
>>> print(var.old.allclose(answer))
True
```

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
```

23.10.31 fipy.variables.noiseVariable**Classes**

NoiseVariable(*args, **kwargs)

Attention:

This class is abstract. Always create one of its subclasses.

class fipy.variables.noiseVariable.**NoiseVariable**(*args, **kwargs)

Bases: *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

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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__(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

`__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
```

`__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 *CellVariable* to persistent storage.

`__gt__`(*other*)

Test if a *Variable* is greater than another quantity

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

(continues on next page)

(continued from previous page)

```
(Variable(value=array(3)) > 4)
>>> print(b())
0
>>> a.value = 5
>>> print(b())
1
```

__hash__()

Return hash(self).

__invert__()Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

__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(10000000000000000000 * 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)
```

__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
```

static __new__(cls, *args, **kwargs)**__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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(dict)Used internally to create a new *CellVariable* from pickled persistent storage.**__str__**()

Return str(self).

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
```

any(axis=None)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

property arithmeticFaceValueReturns 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)
```

(continues on next page)

(continued from previous page)

```

>>> 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.]

```

property constraintMask

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```

>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]

```

copy()

Copy the value of the *NoiseVariable* to a static *CellVariable*.

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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
```

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')
>>> 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
```

property leastSquaresGrad

Return $\nabla\phi$, which is determined by solving for $\nabla\phi$ in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n}_{AP} \cdot \nabla\phi)_{AP}$$

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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]
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$$\text{other} \cdot \text{self}$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

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]

```

scramble()

Generate a new random distribution.

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

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))

```

(continues on next page)

(continued from previous page)

```
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()*2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```

23.10.32 fipy.variables.operatorVariable

23.10.33 fipy.variables.scharfetterGummelFaceVariable

Classes

```
ScharfetterGummelFaceVariable(*args, **kws)
```

param mesh

the mesh that defines the geometry of this *Variable*

```
class fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable(*args,
                                                                                   **kws)
```

Bases: `_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

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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
```

(continues on next page)

(continued from previous page)

```
>>> 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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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
```

__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
```

__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 *MeshVariable* 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
```

(continues on next page)

(continued from previous page)

```
>>> print(b())
1
```

__hash__()

Return hash(self).

__invert__()Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

__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(10000000000000000000 * 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)
```

__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

```

static `__new__(cls, *args, **kwargs)`

`__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]

```

`__pow__(other)`

return $\text{self}^{**}\text{other}$, 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>

```

`__repr__()`

Return `repr(self)`.

`__setstate__(dict)`

Used internally to create a new *Variable* from pickled persistent storage.

`__str__()`

Return `str(self)`.

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(100000))
>>> print(var.allclose(numerix.zeros(100000, 'l')))
False
```

any(*axis=None*)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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 constraintMask

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```

>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]

```

copy()

Make an duplicate of the *Variable*

```

>>> a = Variable(value=3)
>>> b = a.copy()

```

(continues on next page)

(continued from previous page)

```
>>> 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)
>>> var = CellVariable(mesh=mesh, value=list(range(3*2)))
>>> print(var.faceGrad.divergence)
[ 4.  3.  2. -2. -3. -4.]
```

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
```

(continues on next page)

(continued from previous page)

```
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
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')
>>> 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
```

property mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

rdot(other, opShape=None, operatorClass=None)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{other} \cdot \text{self}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

release(*constraint*)

Remove *constraint* from *self*

```
>>> from fipy import *
>>> m = Grid1D(nx=3)
>>> v = CellVariable(mesh=m, value=m.cellCenters[0])
>>> c0 = Constraint(0., where=m.facesLeft)
>>> v.constrain(c0)
>>> c1 = Constraint(3., where=m.facesRight)
>>> v.faceValue.constrain(c1)
>>> print(v.faceValue)
[ 0.  1.  2.  3.]
>>> v.faceValue.release(constraint=c0)
>>> print(v.faceValue)
[ 0.5  1.  2.  3. ]
>>> v.faceValue.release(constraint=c1)
>>> 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
```

property shape

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True

```

Have to account for zero length arrays

```

>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True

```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```

>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()**2).allclose(1.25))
True

```

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

```

23.10.34 fipy.variables.surfactantConvectionVariable

Classes

<code>SurfactantConvectionVariable(*args, **kwargs)</code>	Convection coefficient for the <i>ConservativeSurfactantEquation</i> .

class fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable(*args, **kwargs)

Bases: `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

```

`__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

```

Check representation works with different versions of numpy

```

>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))

```

`__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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
```

__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
```

__ge__(other)

Test if a *Variable* is greater than or equal to another quantity

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

(continues on next page)

(continued from previous page)

```

>>> 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 *MeshVariable* 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).

__invert__()

Returns logical “not” of the *Variable*

```

>>> a = Variable(value=True)
>>> print(~a)
False

```

__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

```

__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(10000000000000000000 * 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)

```

__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

```

static __new__(cls, *args, **kwargs)**__nonzero__()**

```

>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...
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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(*dict*)

Used internally to create a new *Variable* from pickled persistent storage.

__str__()

Return str(self).

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
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

any(*axis=None*)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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 constraintMask

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```
>>> from fipy import *
```

```
>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]
```

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)
>>> var = CellVariable(mesh=mesh, value=list(range(3*2)))
>>> print(var.faceGrad.divergence)
[ 4.  3.  2. -2. -3. -4.]
```

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

self · *other*

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
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')
>>> 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
```

property mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

rdot(other, opShape=None, operatorClass=None)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

other · self

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

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
```

(continues on next page)

(continued from previous page)

```

>>> 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

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True

```

Have to account for zero length arrays


```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allegal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()*2).allclose(1.25))
True
```

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
```

23.10.35 fipy.variables.surfactantVariable

Classes

SurfactantVariable(*args, **kwargs)

The *SurfactantVariable* maintains a conserved volumetric concentration on cells adjacent to, but in front of, the interface.

class fipy.variables.surfactantVariable.*SurfactantVariable*(*args, **kwargs)

Bases: *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.

`__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

```

Check representation works with different versions of numpy

```

>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))

```

`__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__ (*dtype=None, copy=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_wrap__ (*arr, context=None, return_scalar=False*)

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__ (*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

`__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
```

`__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
```

(continues on next page)

(continued from previous page)

```
>>> 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 *CellVariable* 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).

__invert__()

Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

(continues on next page)

(continued from previous page)

```
>>> print(b())
1
>>> a.value = 5
>>> print(b())
0
```

__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)
```

__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
```

static __new__(cls, *args, **kws)**__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]

```

__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>

```

__repr__()

Return repr(self).

__setstate__(*dict*)

Used internally to create a new *CellVariable* from pickled persistent storage.

__str__()

Return str(self).

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

```

`any(axis=None)`

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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.]

```

property `constraintMask`

Test that `constraintMask` returns a `Variable` that updates itself whenever the constraints change.

```

>>> from fipy import *

>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)

```

(continues on next page)

(continued from previous page)

```
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]
```

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]))
```

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

self · *other*

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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
```

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')
>>> 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
```

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.

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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]
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$$\text{other} \cdot \text{self}$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

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

```

property shape

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True

```

(continues on next page)

(continued from previous page)

```
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()*2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```


23.10.36 fipy.variables.test

Test numeric implementation of the mesh

23.10.37 fipy.variables.unaryOperatorVariable

23.10.38 fipy.variables.uniformNoiseVariable

Classes

<code>UniformNoiseVariable(*args, **kws)</code>	Represents a uniform distribution of random numbers.

class fipy.variables.uniformNoiseVariable.UniformNoiseVariable(*args, **kws)

Bases: *NoiseVariable*

Represents a uniform distribution of random numbers.

We generate noise on a uniform Cartesian mesh

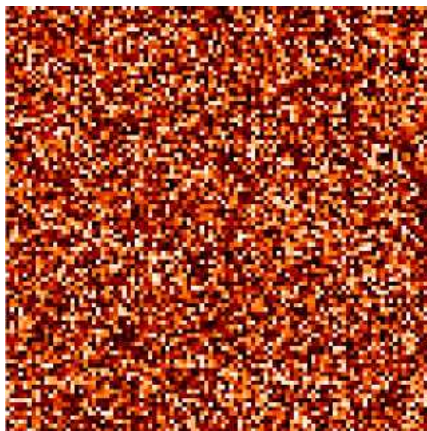
```
>>> from fipy.meshes import Grid2D
>>> noise = UniformNoiseVariable(mesh=Grid2D(nx=100, ny=100))
```

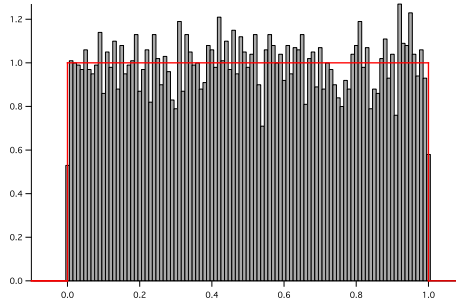
and histogram the noise

```
>>> 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)
```

```
>>> 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.

`__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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)

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__(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

`__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
```

`__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 *CellVariable* to persistent storage.

`__gt__`(*other*)

Test if a *Variable* is greater than another quantity

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

(continues on next page)

(continued from previous page)

```
(Variable(value=array(3)) > 4)
>>> print(b())
0
>>> a.value = 5
>>> print(b())
1
```

__hash__()

Return hash(self).

__invert__()Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

__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(10000000000000000000 * 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)
```

__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
```

static __new__(cls, *args, **kwargs)**__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]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(dict)

Used internally to create a new *CellVariable* from pickled persistent storage.

__str__()

Return str(self).

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
```

any(axis=None)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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)
```

(continues on next page)

(continued from previous page)

```

>>> 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.]

```

property constraintMask

Test that *constraintMask* returns a *Variable* that updates itself whenever the constraints change.

```

>>> from fipy import *

```

```

>>> m = Grid2D(nx=2, ny=2)
>>> x, y = m.cellCenters
>>> v0 = CellVariable(mesh=m)
>>> v0.constrain(1., where=m.facesLeft)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False False  True False False]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  0.  1.  0.  0.]
>>> v0.constrain(3., where=m.facesRight)
>>> print(v0.faceValue.constraintMask)
[False False False False False False  True False  True  True False  True]
>>> print(v0.faceValue)
[ 0.  0.  0.  0.  0.  0.  1.  0.  3.  1.  0.  3.]
>>> v1 = CellVariable(mesh=m)
>>> v1.constrain(1., where=(x < 1) & (y < 1))
>>> print(v1.constraintMask)
[ True False False False]
>>> print(v1)
[ 1.  0.  0.  0.]
>>> v1.constrain(3., where=(x > 1) & (y > 1))
>>> print(v1.constraintMask)
[ True False False  True]
>>> print(v1)
[ 1.  0.  0.  3.]

```

copy()

Copy the value of the *NoiseVariable* to a static *CellVariable*.

dot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$\text{self} \cdot \text{other}$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a [MeshVariable](#).

property dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
True
```

property faceGrad

Return $\nabla\phi$ as a rank-1 *FaceVariable* using differencing 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 $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$ 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
```

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 one element per unit such that the sum of all quantities in the tuple equals 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')
>>> 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
```

property leastSquaresGrad

Return $\nabla\phi$, which is determined by solving for $\nabla\phi$ in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n}_{AP} \cdot \nabla\phi)_{AP}$$

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 mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(axis=None)

```
>>> from fipy import Grid2D, CellVariable
>>> mesh = Grid2D(nx=5, ny=5)
>>> x, y = mesh.cellCenters
>>> v = CellVariable(mesh=mesh, value=x*y)
>>> print(v.min())
0.25
```

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]
```

rdot(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$$\text{other} \cdot \text{self}$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

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]

```

scramble()

Generate a new random distribution.

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

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=2, ny=3)
>>> var = CellVariable(mesh=mesh)
>>> print(numerix.allequal(var.shape, (6,)))
True
>>> print(numerix.allequal(var.arithmeticFaceValue.shape, (17,)))

```

(continues on next page)

(continued from previous page)

```
True
>>> print(numerix.allequal(var.grad.shape, (2, 6)))
True
>>> print(numerix.allequal(var.faceGrad.shape, (2, 17)))
True
```

Have to account for zero length arrays

```
>>> from fipy import Grid1D
>>> m = Grid1D(nx=0)
>>> v = CellVariable(mesh=m, elementshape=(2,))
>>> numerix.allequal((v * 1).shape, (2, 0))
True
```

std(axis=None, **kwargs)

Evaluate standard deviation of all the elements of a *MeshVariable*.

Adapted from <http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=2, ny=2, dx=2., dy=5.)
>>> var = fp.CellVariable(value=(1., 2., 3., 4.), mesh=mesh)
>>> print((var.std()*2).allclose(1.25))
True
```

property unit

Return the unit object of *self*.

```
>>> Variable(value="1 m").unit
<PhysicalUnit m>
```

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.
```

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
```

23.10.39 fipy.variables.variable

Classes

Variable(*args, **kwargs)

Lazily evaluated quantity with units.

class fipy.variables.variable.**Variable**(*args, **kwargs)

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),
...                                                dtype=int))
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
```

Check representation works with different versions of numpy

```
>>> print(repr(abs(Variable(2.3))))
numerix.fabs(Variable(value=array(2.3)))
```

`__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__(dtype=None, copy=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_wrap__(arr, context=None, return_scalar=False)`

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
```

__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
```

__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).

__invert__()

Returns logical “not” of the *Variable*

```
>>> a = Variable(value=True)
>>> print(~a)
False
```

__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
```

__lt__(other)

Test if a *Variable* is less than another quantity

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

(continues on next page)

(continued from previous page)

```

(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)

```

`__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

```

`static __new__(cls, *args, **kws)`

`__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)

```

(continues on next page)

(continued from previous page)

```
>>> print(numerix.allegal((a == 0) | (b == 1), [True, True, False, True]))
True
>>> print(a | b)
[0 1 1 1]
```

__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>
```

__repr__()

Return repr(self).

__setstate__(*dict*)Used internally to create a new *Variable* from pickled persistent storage.**__str__**()

Return str(self).

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
```

any(*axis=None*)

```
>>> print(Variable(value=0).any())
0
>>> print(Variable(value=(0, 0, 1, 1)).any())
1
```

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

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 dtype

Returns the Numpy *dtype* of the underlying array.

```
>>> issubclass(Variable(1).dtype.type, numerix.integer)
True
>>> issubclass(Variable(1.).dtype.type, numerix.floating)
True
>>> issubclass(Variable((1, 1.)).dtype.type, numerix.floating)
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')
>>> 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
```

property mag

The magnitude of the *Variable*, e.g., $|\vec{\psi}| = \sqrt{\vec{\psi} \cdot \vec{\psi}}$.

max(axis=None)

Return the maximum along a given axis.

min(*axis=None*)

Return the minimum along a given axis.

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.allequal(Variable(value=(3,)).shape, (1,))
True
>>> numerix.allequal(Variable(value=(3, 4)).shape, (2,))
True
```

```
>>> Variable(value="3 m").shape
()
>>> numerix.allequal(Variable(value=(3,), unit="m").shape, (1,))
True
>>> numerix.allequal(Variable(value=(3, 4), unit="m").shape, (2,))
True
```

std(axis=None, **kwargs)

Return the standard deviation along a given axis.

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
```

23.11 fipy.viewers

Tools for displaying the values of *Variable* objects

Functions

Viewer(vars[, title, limits, FIPY_VIEWER])

Generic function for creating a *Viewer*.

Classes

<i>DummyViewer</i> (vars[, title])	Substitute viewer that doesn't do anything

Exceptions

<i>MeshDimensionError</i>

class fipy.viewers.**DummyViewer**(vars, title=None, **kwlimits)

Bases: *AbstractViewer*

Substitute viewer that doesn't do anything

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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

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** (`float`, *optional*) – 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.
- **xmax** (`float`, *optional*) – 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.
- **ymin** (`float`, *optional*) – 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.
- **ymax** (`float`, *optional*) – 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.
- **zmin** (`float`, *optional*) – 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.
- **zmax** (`float`, *optional*) – 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.
- **datamin** (`float`, *optional*) – 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.
- **datamax** (`float`, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

exception `fipy.viewers.MeshDimensionError`

Bases: `IndexError`

__cause__

exception cause

__context__

exception context

__delattr__(*name*, /)

Implement `delattr(self, name)`.

__getattr__(*name*, /)

Return `getattr(self, name)`.

`__reduce__()`

Helper for pickle.

`__repr__()`

Return repr(self).

`__setattr__(name, value, /)`

Implement setattr(self, name, value).

`__str__()`

Return str(self).

`add_note()`

Exception.add_note(note) – add a note to the exception

`with_traceback()`

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

`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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.

- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

Modules

fipy.viewers.matplotlibViewer

fipy.viewers.mayaviViewer

fipy.viewers.multiViewer

fipy.viewers.test

Test implementation of the viewers

fipy.viewers.testinteractive

Interactively test the viewers

fipy.viewers.tsvViewer

fipy.viewers.viewer

fipy.viewers.vtkViewer

23.11.1 fipy.viewers.matplotlibViewer

Functions

MatplotlibViewer(vars[, title, limits, ...])

Generic function for creating a *MatplotlibViewer*.

```
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 *Variable* instances 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))
>>> 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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.
- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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

Modules

<code>fipy.viewers.matplotlibViewer.</code>	
<code>abstractMatplotlib2DViewer</code>	
<code>fipy.viewers.matplotlibViewer.</code>	
<code>abstractMatplotlibViewer</code>	
<code>fipy.viewers.matplotlibViewer.</code>	
<code>matplotlib1DViewer</code>	
<code>fipy.viewers.matplotlibViewer.</code>	
<code>matplotlib2DContourViewer</code>	
<code>fipy.viewers.matplotlibViewer.</code>	
<code>matplotlib2DGridContourViewer</code>	
<code>fipy.viewers.matplotlibViewer.</code>	
<code>matplotlib2DGridViewer</code>	
<code>fipy.viewers.matplotlibViewer.</code>	
<code>matplotlib2DViewer</code>	
<code>fipy.viewers.matplotlibViewer.</code>	
<code>matplotlibSparseMatrixViewer</code>	
<code>fipy.viewers.matplotlibViewer.</code>	
<code>matplotlibStreamViewer</code>	
<code>fipy.viewers.matplotlibViewer.</code>	
<code>matplotlibVectorViewer</code>	
<code>fipy.viewers.matplotlibViewer.test</code>	Test numeric implementation of the mesh

fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer

Classes

<code>AbstractMatplotlib2DViewer</code> (vars[, title, ...])	Base class for plotting 2D <code>MeshVariable</code> objects with <code>Matplotlib</code> .

```
class fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer(vars,
                                                                                       ti-
                                                                                       tle=None,
                                                                                       fi-
                                                                                       gaspect=1.0,
                                                                                       cmap=None,
                                                                                       col-
                                                                                       or-
                                                                                       bar=None,
                                                                                       axes=None,
                                                                                       log=False,
                                                                                       **kwlim-
                                                                                       its)
```

Bases: `AbstractMatplotlibViewer`

Base class for plotting 2D `MeshVariable` objects with `Matplotlib`.

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** (`float`, *optional*) – 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.
- **xmax** (`float`, *optional*) – 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.
- **ymin** (`float`, *optional*) – 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.
- **ymax** (`float`, *optional*) – 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.
- **datamin** (`float`, *optional*) – 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.
- **datamax** (`float`, *optional*) – 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

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib Figure* number.

property log

Whether data has logarithmic scaling (*bool*).

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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.
- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.

- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.matplotlibViewer.abstractMatplotlibViewer

Classes

<code>AbstractMatplotlibViewer(vars[, title, ...])</code>	Base class for the viewers that use the <i>Matplotlib</i> plotting package.

```
class fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer(vars,
                                                                                       ti-
                                                                                       tle=None,
                                                                                       fi-
                                                                                       gaspect=1.0,
                                                                                       cmap=None,
                                                                                       col-
                                                                                       or-
                                                                                       bar=None,
                                                                                       axes=None,
                                                                                       log=False,
                                                                                       **kwlim-
                                                                                       its)
```

Bases: *AbstractViewer*

Base class for the viewers that use the *Matplotlib* plotting package.

Attention: This class is abstract. Always create one of its subclasses.

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib Figure* number.

property log

Whether data has logarithmic scaling (*bool*).

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** (float, optional) – 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.
- **xmax** (float, optional) – 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.
- **ymin** (float, optional) – 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.
- **ymax** (float, optional) – 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.
- **zmin** (float, optional) – 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.
- **zmax** (float, optional) – 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.
- **datamin** (float, optional) – 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.
- **datamax** (float, optional) – 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.

property title

The text appearing at the top center.

(default: if len(self.vars) == 1, the name of the only *Variable*, otherwise "")

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.matplotlibViewer.matplotlib1DViewer**Classes**

<code>Matplotlib1DViewer</code> (vars[, title, xlog, ...])	Displays a y vs.

```
class fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer(vars, title=None,
                                                                    xlog=False,
                                                                    ylog=False,
                                                                    limits={},
                                                                    legend='upper left',
                                                                    axes=None,
                                                                    **kwlimits)
```

Bases: *AbstractMatplotlibViewer*

Displays a y vs. x plot of one or more 1D *CellVariable* objects using *Matplotlib*.

```
>>> import fipy as fp
>>> mesh = fp.Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = fp.CellVariable(mesh=mesh, name="x", value=x)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = Matplotlib1DViewer(vars=(fp.numerix.sin(k * xVar) + 2,
...                                fp.numerix.cos(k * xVar / fp.numerix.pi) + 2),
...                             xmin=10, xmax=90,
...                             datamin=1.1, datamax=4.0,
...                             title="Matplotlib1DViewer test")
>>> for kval in fp.numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer.cmap = "ocean"
>>> viewer.log = True
```

```
>>> viewer.title = "Matplotlib1DViewer changed"
>>> 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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).

- **xmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **datamin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **datamax** (*float, optional*) – 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

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib Figure* number.

property lines

The collection of *Matplotlib Line2D* objects representing the plotted data.

property log

Whether data has logarithmic 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.

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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.

- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.matplotlibViewer.matplotlib2DContourViewer**Classes**

<i>Matplotlib2DContourViewer</i> (vars[, title, ...])	Displays a contour plot of a 2D <i>CellVariable</i> object.

```
class fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer(vars,
                                                                                       ti-
                                                                                       tle=None,
                                                                                       lim-
                                                                                       its={},
                                                                                       cmap=None,
                                                                                       col-
                                                                                       or-
                                                                                       bar='vertical',
                                                                                       axes=None,
                                                                                       num-
                                                                                       ber=None,
                                                                                       lev-
                                                                                       els=None,
                                                                                       fi-
                                                                                       gaspect='auto',
                                                                                       **kwlim-
                                                                                       its)
```

Bases: [AbstractMatplotlib2DViewer](#)

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DContourViewer* plots a 2D *CellVariable* using [Matplotlib](#).

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** ([float](#), *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** ([float](#), *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** ([float](#), *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** ([float](#), *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** ([float](#), *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** ([float](#), *optional*) – 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*) – Determines the number and positions of the contour lines / regions. (deprecated, use *levels=*).
- **levels** ([int](#) or [array_like](#), *optional*) – Determines the number and positions of the contour lines / regions. If an *int n*, tries to automatically choose no more than *n+1* “nice” contour levels over the range of *vars*. If *array_like*, draw contour lines at the specified levels. The values must be in increasing order. 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.

property axes

The [Matplotlib Axes](#).

property cmap

The [Matplotlib Colormap](#).

property colorbar

The [Matplotlib Colorbar](#).

property fig

The [Matplotlib Figure](#).

property id

The *Matplotlib* Figure number.

property levels

The number of automatically-chosen contours or their values.

property log

Whether data has logarithmic scaling (*bool*).

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.
- **zmax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fiPy.viewers.matplotlibViewer.matplotlib2DGridContourViewer**Classes**

<i>Matplotlib2DGridContourViewer</i> (vars[, title, ...])	Displays a contour plot of a 2D <i>CellVariable</i> object.

```
class fiPy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer(vars,
                                                    title=None, limits=None, cmap=None, colorbar=None,
                                                    axes=None, levels=None, figsize=None, aspect=None,
                                                    **kwargs)
    """
    A 2D contour plot of a CellVariable object using Matplotlib.
    """
```

Bases: *AbstractMatplotlib2DViewer*

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DGridContourViewer* plots a 2D *CellVariable* using Matplotlib.

```
>>> import fiPy as fp
>>> mesh = fp.Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridContourViewer(vars=fp.numerix.sin(k * xyVar) * 1000 +
    ↪ 1002,
...           ymin=0.1, ymax=0.9,
...           # datamin=1.1, datamax=4.0,
...           title="Matplotlib2DGridContourViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer.levels = 2
```

```
>>> viewer.cmap = "ocean"
>>> viewer.log = True
```

```
>>> viewer.title = "Matplotlib2DGridContourViewer changed"
>>> 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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float*, *optional*) – 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
- **levels** (*int* or *array_like*, *optional*) – Determines the number and positions of the contour lines / regions. If an *int n*, tries to automatically choose no more than *n+1* “nice” contour levels over the range of *vars*. If *array_like*, draw contour lines at the specified levels. The values must be in increasing order. E.g. to draw just the zero contour pass *levels=[0]*.
- **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.

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib* Figure number.

property levels

The number of automatically-chosen contours or their values.

property log

Whether data has logarithmic scaling (*bool*).

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.
- **zmax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.matplotlibViewer.matplotlib2DGridViewer**Classes**

<code>Matplotlib2DGridViewer(vars[, title, ...])</code>	Displays an image plot of a 2D <i>CellVariable</i> object using Matplotlib.

```
class fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer(vars, title=None, limits={}, cmap=None, colorbar='vertical', axes=None, figsize=None, aspect='auto', **kwargs):
```

Bases: *AbstractMatplotlib2DViewer*

Displays an image plot of a 2D *CellVariable* object using Matplotlib.

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = Matplotlib2DGridViewer(vars=fp.numerix.sin(k * xyVar) * 1000 + 1002,
...                               ymin=0.1, ymax=0.9,
...                               # datamin=1.1, datamax=4.0,
...                               title="Matplotlib2DGridViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer.cmap = "ocean"
>>> viewer.log = True
```

```
>>> viewer.title = "Matplotlib2DGridViewer changed"
>>> 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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float*, *optional*) – 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.

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib Figure* number.

property log

Whether data has logarithmic scaling (*bool*).

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.
- **zmax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.matplotlibViewer.matplotlib2DViewer**Classes**

<code>Matplotlib2DViewer</code> (vars[, title, limits, ...])	Displays a contour plot of a 2D <i>CellVariable</i> object.

```
class fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer(vars, title=None,
                                                                           limits={},
                                                                           cmap=None,
                                                                           colorbar='vertical',
                                                                           axes=None,
                                                                           figaspect='auto',
                                                                           **kwlimits)
```

Bases: *AbstractMatplotlib2DViewer*

Displays a contour plot of a 2D *CellVariable* object.

The *Matplotlib2DViewer* plots a 2D *CellVariable* using *Matplotlib*.

```
>>> import fipy as fp
>>> mesh = (fp.Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (fp.Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...         + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = Matplotlib2DViewer(vars=fp.numerix.sin(k * xyVar) * 1000 + 1002,
...                             ymin=0.1, ymax=0.9,
...                             # datamin=1.1, datamax=4.0,
...                             title="Matplotlib2DViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer.cmap = "ocean"
>>> viewer.log = True
```

```
>>> viewer.title = "Matplotlib2DViewer changed"
>>> 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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

- **xmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float, optional*) – 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.

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property collection

The *Matplotlib PolyCollection* representing the cells.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib Figure* number.

property log

Whether data has logarithmic scaling (*bool*).

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** ([float](#), *optional*) – 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.
- **xmax** ([float](#), *optional*) – 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.
- **ymin** ([float](#), *optional*) – 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.
- **ymax** ([float](#), *optional*) – 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.
- **zmin** ([float](#), *optional*) – 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.
- **zmax** ([float](#), *optional*) – 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.
- **datamin** ([float](#), *optional*) – 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.
- **datamax** ([float](#), *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only [Variable](#), otherwise `""`.)

property vars

The [Variable](#) or list of [Variable](#) objects to display.

fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer**Classes**

MatplotlibSparseMatrixViewer ([title])	Displays <code>_SparseMatrix</code> objects using Matplotlib .

```
class fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.MatplotlibSparseMatrixViewer(title='Sparse')
```

Bases: [object](#)

Displays `_SparseMatrix` objects using [Matplotlib](#).

fipy.viewers.matplotlibViewer.matplotlibStreamViewer**Classes**

<code>MatplotlibStreamViewer</code> (vars[, title, log, ...])	Displays a stream plot of a 2D rank-1 <i>CellVariable</i> or <i>FaceVariable</i> object using Matplotlib

```
class fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer(vars, title=None, log=False, limits={}, axes=None, figsize=None, aspect='auto', density=1, linewidth=None, color=None, cmap=None, norm=None, arrowsize=1, arrowstyle='>', minlength=0.1, **kwargs)
```

Bases: [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.

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=1.)
>>> viewer = MatplotlibStreamViewer(vars=fp.numerix.sin(k * xyVar).grad,
...                               title="MatplotlibStreamViewer test")
>>> for kval in fp.numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibStreamViewer(vars=fp.numerix.sin(k * xyVar).faceGrad,
...                               title="MatplotlibStreamViewer test")
```

(continues on next page)

(continued from previous page)

```
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer.cmap = "ocean"
>>> viewer.log = True
```

```
>>> viewer.title = "MatplotlibStreamViewer changed"
>>> viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> import fipy as fp
>>> mesh = (fp.Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (fp.Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=1.)
>>> viewer = MatplotlibStreamViewer(vars=fp.numerix.sin(k * xyVar).grad,
...                                 title="MatplotlibStreamViewer test")
>>> for kval in fp.numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibStreamViewer(vars=fp.numerix.sin(k * xyVar).faceGrad,
...                                 title="MatplotlibStreamViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer.cmap = "ocean"
>>> viewer.log = True
```

```
>>> viewer.title = "MatplotlibStreamViewer changed"
>>> 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** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

- **xmax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **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 *tuple* of *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.

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib Figure* number.

property kwargs

keyword arguments to pass to `streamplot()`.

property log

Whether data has logarithmic scaling (`bool`).

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** (`float`, *optional*) – 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.
- **xmax** (`float`, *optional*) – 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.
- **ymin** (`float`, *optional*) – 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.
- **ymax** (`float`, *optional*) – 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.
- **zmin** (`float`, *optional*) – 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.
- **zmax** (`float`, *optional*) – 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.
- **datamin** (`float`, *optional*) – 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.
- **datamax** (`float`, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.matplotlibViewer.matplotlibVectorViewer**Classes**

<i>MatplotlibVectorViewer</i> (vars[, title, scale, ...])	Displays a vector plot of a 2D rank-1 <i>MeshVariable</i> using <i>Matplotlib</i>

```
class fipy.viewers.matplotlibViewer.matplotlibVectorViewer(vars, ti-
                                                         tle=None,
                                                         scale=None,
                                                         spar-
                                                         sity=None,
                                                         log=False,
                                                         lim-
                                                         its={},
                                                         axes=None,
                                                         fi-
                                                         gaspect='auto',
                                                         **kwlim-
                                                         its)
```

Bases: *AbstractMatplotlib2DViewer*

Displays a vector plot of a 2D rank-1 *MeshVariable* using *Matplotlib*

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=fp.numerix.sin(k * xyVar).grad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in fp.numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=fp.numerix.sin(k * xyVar).faceGrad,
...                               title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> for sparsity in numerix.arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer.cmap = "ocean"
>>> viewer.log = True
```

```
>>> viewer.title = "MatplotlibVectorViewer changed"
>>> viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> import fipy as fp
>>> mesh = (fp.Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (fp.Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=1.)
>>> viewer = MatplotlibVectorViewer(vars=fp.numerix.sin(k * xyVar).grad,
...                                 title="MatplotlibVectorViewer test")
>>> for kval in fp.numerix.arange(1, 10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer = MatplotlibVectorViewer(vars=fp.numerix.sin(k * xyVar).faceGrad,
...                                 title="MatplotlibVectorViewer test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> for sparsity in numerix.arange(5000, 0, -500):
...     viewer.quiver(sparsity=sparsity)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer.cmap = "ocean"
>>> viewer.log = True
```

```
>>> viewer.title = "MatplotlibVectorViewer changed"
>>> 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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float*, *optional*) – 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.

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib Figure* number.

property log

Whether data has logarithmic scaling (*bool*).

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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.
- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.matplotlibViewer.test

Test numeric implementation of the mesh

23.11.2 fipy.viewers.mayaviViewer

```
class fipy.viewers.mayaviViewer.MayaviClient(vars, title=None, daemon_file=None, fps=1.0,
                                             **kwlimits)
```

Bases: *AbstractViewer*

The *MayaviClient* uses the *Mayavi* python plotting package.

```
>>> import fipy as fp
>>> mesh = fp.Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = fp.CellVariable(mesh=mesh, name="x", value=x)
```

(continues on next page)

(continued from previous page)

```

>>> k = fp.Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=(fp.numerix.sin(k * xVar) + 2,
...                             fp.numerix.cos(k * xVar / fp.numerix.pi) + 2),
...                       xmin=10, xmax=90,
...                       datamin=1.1, datamax=4.0,
...                       title="MayaviClient test")
>>> for kval in fp.numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

```

>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=fp.numerix.sin(k * xyVar) * 1000 + 1002,
...                       ymin=0.1, ymax=0.9,
...                       # datamin=1.1, datamax=4.0,
...                       title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

```

>>> import fipy as fp
>>> mesh = (fp.Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (fp.Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5, ), (0.2, ))))
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=fp.numerix.sin(k * xyVar) * 1000 + 1002,
...                       ymin=0.1, ymax=0.9,
...                       # datamin=1.1, datamax=4.0,
...                       title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

```

>>> import fipy as fp
>>> mesh = fp.Grid3D(nx=50, ny=100, nz=10, dx=0.1, dy=0.01, dz=0.1)
>>> x, y, z = mesh.cellCenters
>>> xyzVar = fp.CellVariable(mesh=mesh, name=r"x y z", value=x * y * z)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=fp.numerix.sin(k * xyzVar) + 2,
...                       ymin=0.1, ymax=0.9,
...                       datamin=1.1, datamax=4.0,
...                       title="MayaviClient test")
>>> for kval in range(10):

```

(continues on next page)

(continued from previous page)

```
...     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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.
- **zmax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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

property **fps**

The frames per second to attempt to display.

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.
- **zmax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

Modules

<code>fipy.viewers.mayaviViewer.mayaviClient</code>	
<code>fipy.viewers.mayaviViewer.mayaviDaemon</code>	A simple script that polls a data file for changes and then updates the Mayavi pipeline automatically.
<code>fipy.viewers.mayaviViewer.test</code>	Test numeric implementation of the mesh

fipy.viewers.mayaviViewer.mayaviClient

Classes

<code>MayaviClient</code> (vars[, title, daemon_file, fps])	The <i>MayaviClient</i> uses the <i>Mayavi</i> python plotting package.

class fipy.viewers.mayaviViewer.mayaviClient.**MayaviClient**(vars, title=None, daemon_file=None, fps=1.0, **kwlimits)

Bases: *AbstractViewer*

The *MayaviClient* uses the *Mayavi* python plotting package.

```
>>> import fipy as fp
>>> mesh = fp.Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = fp.CellVariable(mesh=mesh, name="x", value=x)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=(fp.numerix.sin(k * xVar) + 2,
...                               fp.numerix.cos(k * xVar / fp.numerix.pi) + 2),
...                        xmin=10, xmax=90,
...                        datamin=1.1, datamax=4.0,
...                        title="MayaviClient test")
>>> for kval in fp.numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> import fipy as fp
>>> mesh = fp.Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=fp.numerix.sin(k * xyVar) * 1000 + 1002,
...                        ymin=0.1, ymax=0.9,
...                        # datamin=1.1, datamax=4.0,
...                        title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> import fipy as fp
>>> mesh = (fp.Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...         + (fp.Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...           + ((0.5,), (0.2,))))
>>> x, y = mesh.cellCenters
>>> xyVar = fp.CellVariable(mesh=mesh, name="x y", value=x * y)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=fp.numerix.sin(k * xyVar) * 1000 + 1002,
...                       ymin=0.1, ymax=0.9,
...                       # datamin=1.1, datamax=4.0,
...                       title="MayaviClient test")
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> import fipy as fp
>>> mesh = fp.Grid3D(nx=50, ny=100, nz=10, dx=0.1, dy=0.01, dz=0.1)
>>> x, y, z = mesh.cellCenters
>>> xyzVar = fp.CellVariable(mesh=mesh, name=r"x y z", value=x * y * z)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = MayaviClient(vars=fp.numerix.sin(k * xyzVar) + 2,
...                       ymin=0.1, ymax=0.9,
...                       datamin=1.1, datamax=4.0,
...                       title="MayaviClient test")
>>> 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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.

- **zmax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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

property fps

The frames per second to attempt to display.

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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.
- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.

- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.mayaviViewer.mayaviDaemon

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.

Functions

<code>main([argv])</code>	Simple helper to start up the mayavi application.

Classes

<code>MayaviDaemon(*args, **kwargs)</code>	Given a file name and a mayavi2 data reader object, this class polls the file for any changes and automatically updates the mayavi pipeline.

class `fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon` (**args: Any, **kwargs: Any*)

Bases: `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.

parse_command_line(*argv*)

Parse command line options.

Parameters

argv (list of str) – The command line arguments

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.mayaviDaemon.main(argv=None)`

Simple helper to start up the mayavi application.

This returns the running application.

fipy.viewers.mayaviViewer.test

Test numeric implementation of the mesh

23.11.3 fipy.viewers.multiViewer

Classes

<i>MultiViewer</i> (viewers)	Treat a collection of different viewers (such for different 2D plots or 1D plots with different axes) as a single viewer that will <i>plot()</i> all subviewers simultaneously.

class `fipy.viewers.multiViewer.MultiViewer`(*viewers*)

Bases: *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 Viewer) – the viewers to bind together

plot()

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.
- **zmax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

23.11.4 fipy.viewers.test

Test implementation of the viewers

23.11.5 fipy.viewers.testinteractive

Interactively test the viewers

23.11.6 fipy.viewers.tsvViewer

Classes

<code>TSVViewer(vars[, title, limits])</code>	"Views" one or more variables in tab-separated-value format.

class fipy.viewers.tsvViewer.TSVViewer(*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: `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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

- **zmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

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       -3.33333333333333
0.15   0.15   2        10        5
0.05   0.45  -2        35       -3.33333333333333
0.15   0.45   5        35        5
```

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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.

- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

23.11.7 fipy.viewers.viewer

Classes

<i>AbstractViewer</i> (vars[, title])	Base class for FiPy Viewers

class `fipy.viewers.viewer.AbstractViewer`(vars, title=None, **kwlimits)

Bases: `object`

Base class for FiPy Viewers

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** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

- **xmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.
- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.

- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

23.11.8 fipy.viewers.vtkViewer

Functions

VTKViewer(vars[, title, limits])

Generic function for creating a *VTKViewer*.

class fipy.viewers.vtkViewer.VTKCellViewer(vars, title=None, limits={}, **kwlimits)

Bases: *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** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

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** (float, optional) – 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.
- **xmax** (float, optional) – 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.
- **ymin** (float, optional) – 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.
- **ymax** (float, optional) – 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.
- **zmin** (float, optional) – 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.
- **zmax** (float, optional) – 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.
- **datamin** (float, optional) – 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.
- **datamax** (float, optional) – 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.

property title

The text appearing at the top center.

(default: if len(self.vars) == 1, the name of the only *Variable*, otherwise "").)

property vars

The *Variable* or list of *Variable* objects to display.

class fipy.viewers.vtkViewer.VTKFaceViewer(*vars*, *title=None*, *limits={}*, ***kwlimits*)

Bases: *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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.

- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

`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** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

- **datamin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

Modules

<code>fipy.viewers.vtkViewer.test</code>	Test numeric implementation of the mesh
<code>fipy.viewers.vtkViewer.vtkCellViewer</code>	
<code>fipy.viewers.vtkViewer.vtkFaceViewer</code>	
<code>fipy.viewers.vtkViewer.vtkViewer</code>	

fipy.viewers.vtkViewer.test

Test numeric implementation of the mesh

fipy.viewers.vtkViewer.vtkCellViewer

Classes

<code>VTKCellViewer</code> (vars[, title, limits])	Renders <i>CellVariable</i> data in VTK format

class fipy.viewers.vtkViewer.vtkCellViewer.**VTKCellViewer**(vars, title=None, limits={}, **kwlimits)

Bases: `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** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

- **zmax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float, optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.
- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.vtkViewer.vtkFaceViewer**Classes**

VTKFaceViewer(vars[, title, limits])Renders *MeshVariable* data in VTK format

class fipy.viewers.vtkViewer.vtkFaceViewer.**VTKFaceViewer**(vars, title=None, limits={}, **kwlimits)

Bases: *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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.
- **zmax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

fipy.viewers.vtkViewer.vtkViewer**Classes****VTKViewer**(vars[, title, limits])Renders *MeshVariable* data in VTK format**class** fipy.viewers.vtkViewer.vtkViewer.**VTKViewer**(vars, title=None, limits={}, **kwlimits)Bases: *AbstractViewer*Renders *MeshVariable* data in VTK formatCreates 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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **xmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **ymax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **zmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.
- **datamax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale.

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.
- **zmax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

Chapter 24

examples

Demonstration scripts and high-level tests of the *fipy* package

Modules

`examples.benchmarking`

`examples.cahnHilliard`

`examples.chemotaxis`

`examples.convection`

`examples.diffusion`

`examples.elphf`

The following examples exhibit various parts of a model to study electrochemical interfaces.

`examples.flow`

`examples.levelSet`

`examples.meshing`

`examples.parallel`

`examples.phase`

`examples.reactiveWetting`

`examples.riemann`

`examples.test`

Run all the test cases in examples/

`examples.updating`

24.1 examples.benchmarking

Modules

<code>examples.benchmarking.benchmarker</code>
<code>examples.benchmarking.size</code>
<code>examples.benchmarking.steps</code>
<code>examples.benchmarking.utils</code>
<code>examples.benchmarking.versions</code>

24.1.1 examples.benchmarking.benchmarker

24.1.2 examples.benchmarking.size

24.1.3 examples.benchmarking.steps

24.1.4 examples.benchmarking.utils

24.1.5 examples.benchmarking.versions

24.2 examples.cahnHilliard

Modules

<code>examples.cahnHilliard.mesh2D</code>	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).
<code>examples.cahnHilliard.mesh2DCoupled</code>	Solve the Cahn-Hilliard problem in two dimensions.
<code>examples.cahnHilliard.mesh3D</code>	Solves the Cahn-Hilliard problem in a 3D cube
<code>examples.cahnHilliard.sphere</code>	Solves the Cahn-Hilliard problem on the surface of a sphere.
<code>examples.cahnHilliard.sphereDaemon</code>	
<code>examples.cahnHilliard.tanh1D</code>	This example solves the Cahn-Hilliard equation given by,
<code>examples.cahnHilliard.test</code>	

24.2.1 examples.cahnHilliard.mesh2D

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, TransientTerm, \
DiffusionTerm, ImplicitSourceTerm, LinearLUSolver, Viewer, DefaultSolver
>>> 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 = 1000
... else:
...     nx = ny = 20
>>> mesh = Grid2D(nx=nx, ny=ny, dx=0.25, dy=0.25)
>>> 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 __name__ == "__main__":
...     viewer = Viewer(vars=(phi,), datamin=0., datamax=1.)
```

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 ϕ 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)))
```

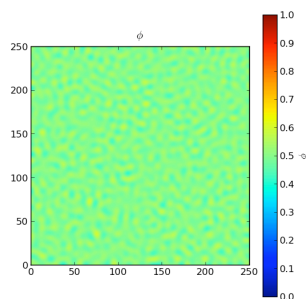
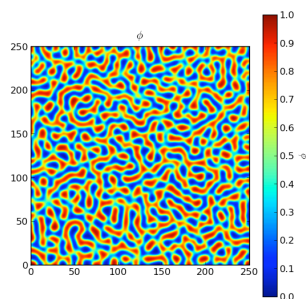
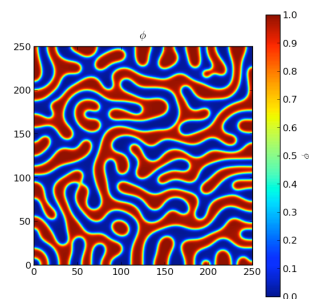
```
>>> import fipy.solvers.solver
>>> if fipy.solvers.solver_suite in ['petsc']:
...     solver = DefaultSolver(precon=None)
... elif fipy.solvers.solver_suite in ['trilinos']:
...     solver = LinearLUSolver()
... else:
...     solver = DefaultSolver()
```

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.
>>> duration = 1000.
```

```
>>> while elapsed < duration:
...     dt = min(100, numerix.exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(phi, dt=dt, solver=solver)
...     if __name__ == "__main__":
...         viewer.plot()
...     elif (max(phi.globalValue) > 0.7) and (min(phi.globalValue) < 0.3) and elapsed > 10.:
...         break
```

```
>>> print((max(phi.globalValue) > 0.7) and (min(phi.globalValue) < 0.3))
True
```

(a) $t = 30$ (b) $t = 100$ (c) $t = 1000$

24.2.2 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, (obsolete) Pysparse required 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))
```

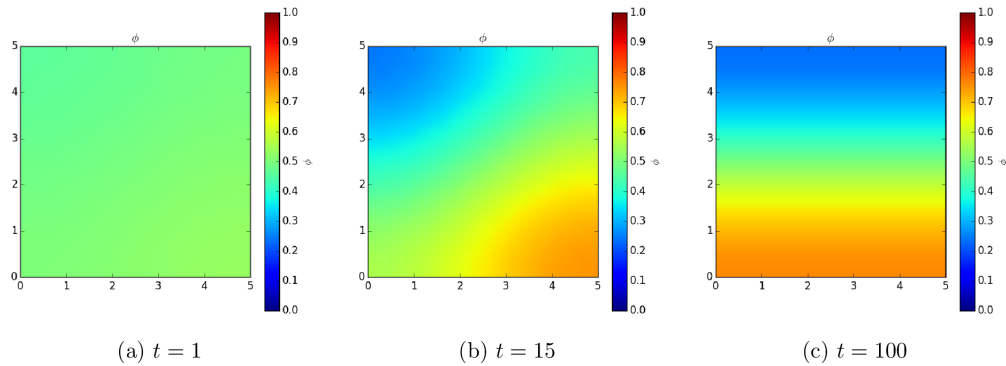
```
>>> eq = eq1 & eq2
```

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)
>>> dfdphi_ = 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),
...                        (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)
```

(continues on next page)

(continued from previous page)

```
...     if __name__ == "__main__":
...         viewer.plot()
```

```
>>> print(numerix.allclose(var, (phi, psi)))
True
```

24.2.3 examples.cahnHilliard.mesh3D

Solves the Cahn-Hilliard problem in a 3D cube

```
>>> from fipy import CellVariable, Grid3D, Viewer, GaussianNoiseVariable, TransientTerm, \
    DiffusionTerm, DefaultSolver
>>> from fipy.tools import numerix
```

The only difference from *examples.cahnHilliard.mesh2D* is the declaration of *mesh*.

```
>>> if __name__ == "__main__":
...     nx = ny = nz = 100
...     else:
...         nx = ny = nz = 10
>>> mesh = Grid3D(nx=nx, ny=ny, nz=nz, dx=0.25, dy=0.25, dz=0.25)
>>> 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 __name__ == "__main__":
...     viewer = Viewer(vars=(phi,), datamin=0., datamax=1.)
```

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:

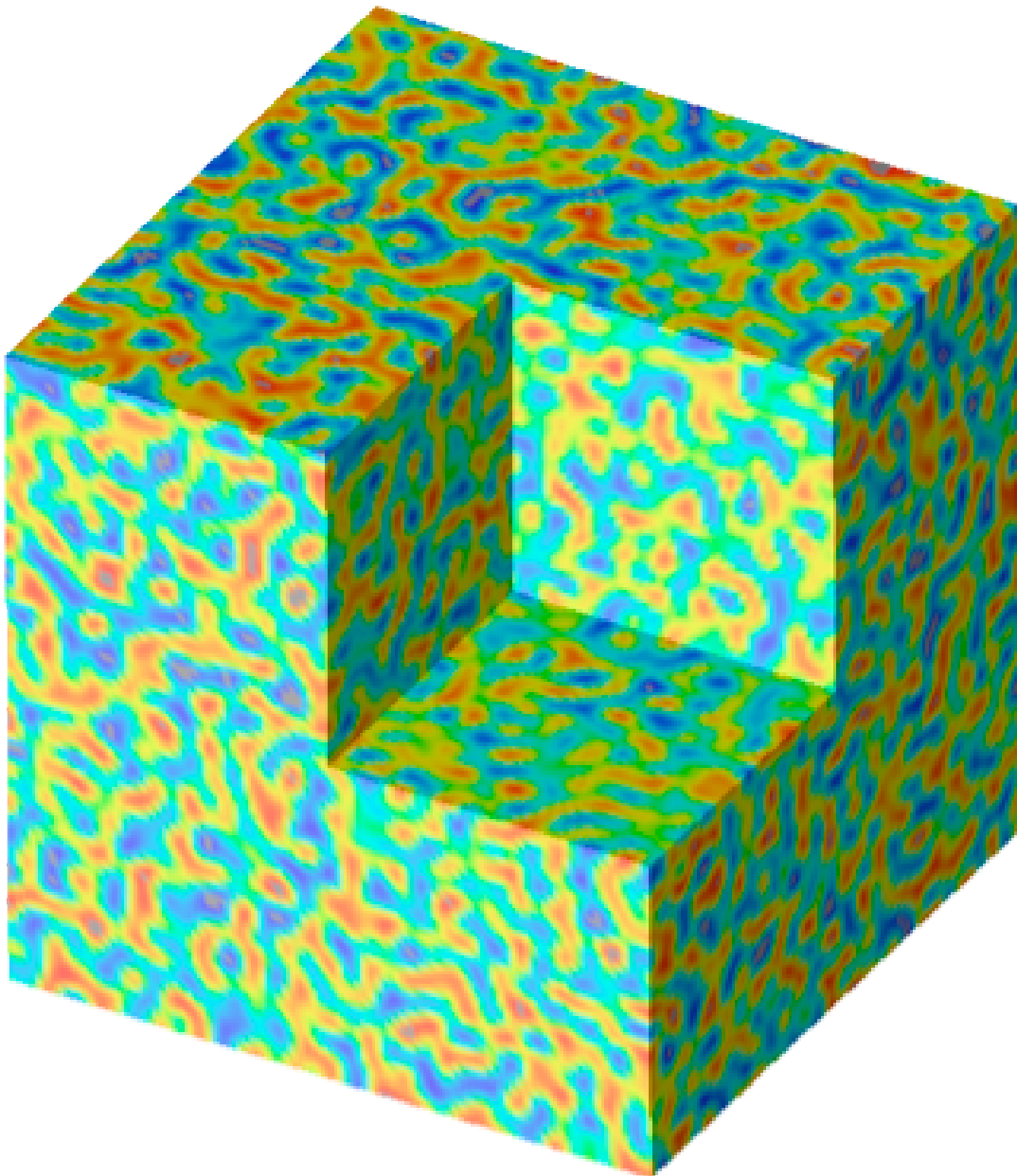
$$\frac{\partial \phi}{\partial t} = \nabla \cdot D a^2 [1 - 6\phi(1 - \phi)] \nabla \phi - \nabla \cdot D \nabla \epsilon^2 \nabla^2 \phi.$$

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 *phi* 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:
...         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()
```



24.2.4 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}}};
...     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.,
```

(continues on next page)

(continued from previous page)

```

...         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 ϕ 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)))

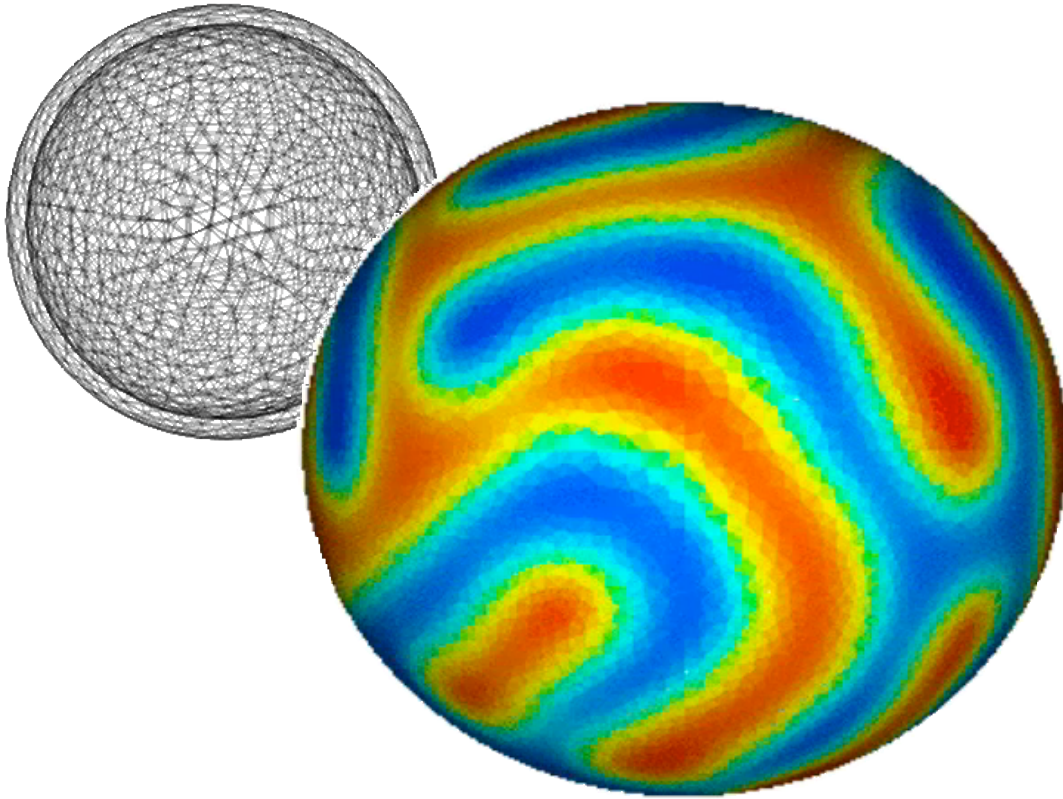
```

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:
...         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()

```



24.2.5 examples.cahnHilliard.sphereDaemon

24.2.6 examples.cahnHilliard.tanh1D

This example solves the Cahn-Hilliard equation given by,

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left(\frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right)$$

where the free energy functional is given by,

$$f = \frac{a^2}{2} \phi^2 (1 - \phi)^2$$

The Cahn-Hilliard equation can be rewritten in the following form,

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \left(\frac{\partial^2 f}{\partial \phi^2} \nabla \phi - \epsilon^2 \nabla^3 \phi \right)$$

The above form of the equation makes the non-linearity part of the diffusion coefficient for the first term on the RHS. This is the correct way to express the equation to *FiPy*.

We solve the problem on a 1D mesh

```
>>> from fipy import CellVariable, Grid1D, NthOrderBoundaryCondition, DiffusionTerm, \
↳ TransientTerm, LinearLUSolver, DefaultSolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 40.
>>> nx = 1000
>>> dx = L / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

and create the solution variable

```
>>> var = CellVariable(
...     name="phase field",
...     mesh=mesh,
...     value=1.)
```

The boundary conditions for this problem are

$$\left. \begin{array}{l} \phi = \frac{1}{2} \\ \frac{\partial^2 \phi}{\partial x^2} = 0 \end{array} \right\} \quad \text{on } x = 0$$

and

$$\left. \begin{array}{l} \phi = 1 \\ \frac{\partial^2 \phi}{\partial x^2} = 0 \end{array} \right\} \quad \text{on } x = L$$

or

```
>>> BCs = (
...     NthOrderBoundaryCondition(faces=mesh.facesLeft, value=0, order=2),
...     NthOrderBoundaryCondition(faces=mesh.facesRight, value=0, order=2))
```

```
>>> var.constrain(1, mesh.facesRight)
>>> var.constrain(.5, mesh.facesLeft)
```

Using

```
>>> asq = 1.0
>>> epsilon = 1
>>> diffusionCoeff = 1
```

we create the Cahn-Hilliard equation:

```
>>> faceVar = var.arithmeticFaceValue
>>> freeEnergyDoubleDerivative = asq * ( 1 - 6 * faceVar * (1 - faceVar))
```

```
>>> diffTerm2 = DiffusionTerm(
...     coeff=diffusionCoeff * freeEnergyDoubleDerivative)
>>> diffTerm4 = DiffusionTerm(coeff=(diffusionCoeff, epsilon**2))
>>> eqch = TransientTerm() == diffTerm2 - diffTerm4
```

```
>>> from fipy.solvers import solver_suite
>>> solver = DefaultSolver(tolerance=1e-15, iterations=100)
```

The solution to this 1D problem over an infinite domain is given by,

$$\phi(x) = \frac{1}{1 + \exp\left(-\frac{a}{\epsilon}x\right)}$$

or

```
>>> a = numerix.sqrt(asq)
>>> answer = 1 / (1 + numerix.exp(-a * (mesh.cellCenters[0]) / epsilon))
```

If we are running interactively, we create a viewer to see the results

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=0., datamax=1.0)
...     viewer.plot()
```

We iterate the solution to equilibrium and, if we are running interactively, we update the display and output data about the progression of the solution

```
>>> dexp=-5
>>> for step in range(100):
...     dt = numerix.exp(dexp)
...     dt = min(10, dt)
...     dexp += 0.5
...     eqch.solve(var=var, boundaryConditions=BCs, solver=solver, dt=dt)
...     if __name__ == '__main__':
...         diff = abs(answer - numerix.array(var))
...         maxarg = numerix.argmax(diff)
...         print('maximum error: {}'.format(diff[maxarg]))
...         print('element id:', maxarg)
...         print('value at element {} is {}'.format(maxarg, var[maxarg]))
...         print('solution value: {}'.format(answer[maxarg]))
...
...     viewer.plot()
```

We compare the analytical solution with the numerical result,

```
>>> print(var.allclose(answer, atol=1e-4))
1
```

24.2.7 examples.cahnHilliard.test

24.3 examples.chemotaxis

Modules

<code>examples.chemotaxis.input</code>	Input file for chemotaxis modeling.
<code>examples.chemotaxis.input2D</code>	Input file for chemotaxis modeling.
<code>examples.chemotaxis.parameters</code>	Input file for parameters
<code>examples.chemotaxis.test</code>	

24.3.1 examples.chemotaxis.input

Input file for chemotaxis modeling.

Here are some test cases for the model.

```
>>> from examples.chemotaxis.parameters import parameters
>>> from fipy import CellVariable, Grid1D, TransientTerm, DiffusionTerm,
↳ ImplicitSourceTerm, Viewer, numerix
```

```
>>> params = parameters['case 2']
```

```
>>> nx = 50
>>> dx = 1.
>>> L = nx * dx
```

```
>>> mesh = Grid1D(nx=nx, dx=dx)
```

```
>>> shift = 1.
```

```
>>> KMVar = CellVariable(mesh=mesh, value=params['KM'] * shift, hasOld=1)
>>> KCVar = CellVariable(mesh=mesh, value=params['KC'] * shift, hasOld=1)
>>> TMVar = CellVariable(mesh=mesh, value=params['TM'] * shift, hasOld=1)
>>> TCVar = CellVariable(mesh=mesh, value=params['TC'] * shift, hasOld=1)
>>> P3Var = CellVariable(mesh=mesh, value=params['P3'] * shift, hasOld=1)
>>> P2Var = CellVariable(mesh=mesh, value=params['P2'] * shift, hasOld=1)
>>> RVar = CellVariable(mesh=mesh, value=params['R'], hasOld=1)
```

```
>>> PN = P3Var + P2Var
```

```
>>> KMscCoeff = params['chiK'] * (RVar + 1) * (1 - KCVar - KMVar.cellVolumeAverage)
>>> KMspCoeff = params['lambdaK'] / (1 + PN / params['kappaK'])
>>> KMEq = TransientTerm() - KMscCoeff + ImplicitSourceTerm(KMspCoeff)
```

```
>>> TMscCoeff = params['chiT'] * (1 - TCVar - TMVar.cellVolumeAverage)
>>> TMspCoeff = params['lambdaT'] * (KMVar + params['zetaT'])
>>> TMEq = TransientTerm() - TMscCoeff + ImplicitSourceTerm(TMspCoeff)
```

```
>>> TCscCoeff = params['lambdaT'] * (TMVar * KMVar).cellVolumeAverage
>>> TCspCoeff = params['lambdaTstar']
>>> TCEq = TransientTerm() - TCscCoeff + ImplicitSourceTerm(TCspCoeff)
```

```
>>> PIP2PITP = PN / (PN / params['kappam'] + PN.cellVolumeAverage / params['kappac'] +
↳ 1) + params['zetaPITP']
```

```
>>> P3spCoeff = params['lambda3'] * (TMVar + params['zeta3T'])
>>> P3scCoeff = params['chi3'] * KMVar * (PIP2PITP / (1 + KMVar / params['kappa3']) +
↳ params['zeta3PITP']) + params['zeta3']
>>> P3Eq = TransientTerm() - DiffusionTerm(params['diffusionCoeff']) - P3scCoeff +
↳ ImplicitSourceTerm(P3spCoeff)
```

```
>>> P2scCoeff = scCoeff = params['chi2'] + params['lambda3'] * params['zeta3T'] * P3Var
>>> P2spCoeff = params['lambda2'] * (TMVar + params['zeta2T'])
>>> P2Eq = TransientTerm() - DiffusionTerm(params['diffusionCoeff']) - P2scCoeff +
↳ ImplicitSourceTerm(P2spCoeff)
```

```
>>> KCscCoeff = params['alphaKstar'] * params['lambdaK'] * (KMVar / (1 + PN / params[
↳ 'kappaK'])).cellVolumeAverage
>>> KCspCoeff = params['lambdaKstar'] / (params['kappaKstar'] + KCVar)
>>> KCEq = TransientTerm() - KCscCoeff + ImplicitSourceTerm(KCspCoeff)
```

```
>>> eqs = ((KMVar, KMEq), (TMVar, TMEq), (TCVar, TCEq), (P3Var, P3Eq), (P2Var, P2Eq),
↳ (KCVar, KCEq))
```

```
>>> if __name__ == '__main__':
...     steps = 100
... else:
...     steps = 28
```

```
>>> for i in range(steps):
...     for var, eqn in eqs:
...         var.updateOld()
...     for var, eqn in eqs:
...         eqn.solve(var, dt=1.)
```

```
>>> accuracy = 1e-2
>>> print(KMVar.allclose(params['KM'], atol=accuracy))
1
>>> print(TMVar.allclose(params['TM'], atol=accuracy))
1
>>> print(TCVar.allclose(params['TC'], atol=accuracy))
1
>>> print(P2Var.allclose(params['P2'], atol=accuracy))
1
>>> print(P3Var.allclose(params['P3'], atol=accuracy))
1
>>> print(KCVar.allclose(params['KC'], atol=accuracy))
1
```

```
>>> PNView = PN / PN.cellVolumeAverage
>>> PNView.name = 'PN'
```

```
>>> KMView = KMVar / KMVar.cellVolumeAverage
>>> KMView.name = 'KM'
```

```
>>> TMView = TMVar / TMVar.cellVolumeAverage
>>> TMView.name = 'TM'
```

```
>>> RVar[:] = params['S'] + (1 + params['S']) * params['G'] * numerix.cos((2 * numerix.
↳ pi * mesh.cellCenters[0]) / L)
```

```

>>> if __name__ == '__main__':
...     KMViewer = Viewer((PNView, KMView, TMView), title = 'Gradient Stimulus: Profile')
...
...     for i in range(100):
...         for var, eqn in eqs:
...             var.updateOld()
...         for var, eqn in eqs:
...             eqn.solve(var, dt=0.1)
...
...     KMViewer.plot()
...
...     input("finished")

```

24.3.2 examples.chemotaxis.input2D

Input file for chemotaxis modeling.

Here are some test cases for the model.

```

>>> from examples.chemotaxis.parameters import parameters
>>> from fipy import CellVariable, Grid2D, TransientTerm, DiffusionTerm,
↳ ImplicitSourceTerm, Viewer, numerix

```

```

>>> params = parameters['case 2']

```

```

>>> nx = 50
>>> ny = 50
>>> dx = 1.
>>> L = nx * dx

```

```

>>> mesh = Grid2D(nx=nx, ny=ny, dx=dx, dy=1.)

```

```

>>> shift = 1.

```

```

>>> KMVar = CellVariable(mesh=mesh, value=params['KM'] * shift, hasOld=1)
>>> KCVar = CellVariable(mesh=mesh, value=params['KC'] * shift, hasOld=1)
>>> TMVar = CellVariable(mesh=mesh, value=params['TM'] * shift, hasOld=1)
>>> TCVar = CellVariable(mesh=mesh, value=params['TC'] * shift, hasOld=1)
>>> P3Var = CellVariable(mesh=mesh, value=params['P3'] * shift, hasOld=1)
>>> P2Var = CellVariable(mesh=mesh, value=params['P2'] * shift, hasOld=1)
>>> RVar = CellVariable(mesh=mesh, value=params['R'], hasOld=1)

```

```

>>> PN = P3Var + P2Var

```

```

>>> KMscCoeff = params['chiK'] * (RVar + 1) * (1 - KCVar - KMVar.cellVolumeAverage)
>>> KMspCoeff = params['lambdaK'] / (1 + PN / params['kappaK'])
>>> KMEq = TransientTerm() - KMscCoeff + ImplicitSourceTerm(KMspCoeff)

```

```
>>> TMscCoeff = params['chiT'] * (1 - TCVar - TMVar.cellVolumeAverage)
>>> TMspCoeff = params['lambdaT'] * (KMVar + params['zetaT'])
>>> TMEq = TransientTerm() - TMscCoeff + ImplicitSourceTerm(TMspCoeff)
```

```
>>> TCscCoeff = params['lambdaT'] * (TMVar * KMVar).cellVolumeAverage
>>> TCspCoeff = params['lambdaTstar']
>>> TCEq = TransientTerm() - TCscCoeff + ImplicitSourceTerm(TCspCoeff)
```

```
>>> PIP2PITP = PN / (PN / params['kappam'] + PN.cellVolumeAverage / params['kappac'] +
↳ 1) + params['zetaPITP']
```

```
>>> P3spCoeff = params['lambda3'] * (TMVar + params['zeta3T'])
>>> P3scCoeff = params['chi3'] * KMVar * (PIP2PITP / (1 + KMVar / params['kappa3']) +
↳ params['zeta3PITP']) + params['zeta3']
>>> P3Eq = TransientTerm() - DiffusionTerm(params['diffusionCoeff']) - P3scCoeff +
↳ ImplicitSourceTerm(P3spCoeff)
```

```
>>> P2scCoeff = scCoeff = params['chi2'] + params['lambda3'] * params['zeta3T'] * P3Var
>>> P2spCoeff = params['lambda2'] * (TMVar + params['zeta2T'])
>>> P2Eq = TransientTerm() - DiffusionTerm(params['diffusionCoeff']) - P2scCoeff +
↳ ImplicitSourceTerm(P2spCoeff)
```

```
>>> KCscCoeff = params['alphaKstar'] * params['lambdaK'] * (KMVar / (1 + PN / params[
↳ 'kappaK'])).cellVolumeAverage
>>> KCspCoeff = params['lambdaKstar'] / (params['kappaKstar'] + KCVar)
>>> KCEq = TransientTerm() - KCscCoeff + ImplicitSourceTerm(KCspCoeff)
```

```
>>> eqs = ((KMVar, KMEq), (TMVar, TMEq), (TCVar, TCEq), (P3Var, P3Eq), (P2Var, P2Eq),
↳ (KCVar, KCEq))
```

```
>>> if __name__ == '__main__':
...     steps = 100
... else:
...     steps = 10
```

```
>>> for i in range(steps):
...     for var, eqn in eqs:
...         var.updateOld()
...     for var, eqn in eqs:
...         eqn.solve(var, dt=1.)
```

```
>>> accuracy = 1e-2
>>> print(KMVar.allclose(params['KM'], atol=accuracy))
1
>>> print(TMVar.allclose(params['TM'], atol=accuracy))
1
>>> print(TCVar.allclose(params['TC'], atol=accuracy))
1
>>> print(P2Var.allclose(params['P2'], atol=accuracy))
1
```

(continues on next page)

(continued from previous page)

```
>>> print(P3Var.allclose(params['P3'], atol=accuracy))
1
>>> print(KCVar.allclose(params['KC'], atol=accuracy))
1
```

```
>>> PNView = PN / PN.cellVolumeAverage
>>> PNView.name = 'PN'
```

```
>>> KMView = KMVar / KMVar.cellVolumeAverage
>>> KMView.name = 'KM'
```

```
>>> TMView = TMVar / TMVar.cellVolumeAverage
>>> TMView.name = 'TM'
```

```
>>> x, y = mesh.cellCenters
>>> RVar[:] = L / numerix.sqrt((x - L / 2)**2 + (y - 2 * L)**2)
```

```
>>> if __name__ == '__main__':
...     PNViewer = Viewer(PNView, datamax=2., datamin=0., title='')
...     KMViewer = Viewer(KMView, datamax=2., datamin=0., title='')
...     TMViewer = Viewer(TMView, datamax=2., datamin=0., title='')
...
...     for i in range(100):
...         for var, eqn in eqs:
...             var.updateOld()
...         for var, eqn in eqs:
...             eqn.solve(var, dt=1.)
...
...     PNViewer.plot()
...     KMViewer.plot()
...     TMViewer.plot()
...
...     input("finished")
```

24.3.3 examples.chemotaxis.parameters

Input file for parameters

24.3.4 examples.chemotaxis.test

24.4 examples.convection

Modules

<code>examples.convection.advection</code>	
<code>examples.convection.exponential1D</code>	
<code>examples.convection.exponential1DBack</code>	
<code>examples.convection.exponential1DSource</code>	
<code>examples.convection.exponential2D</code>	
<code>examples.convection.peclet</code>	This example tests diffusion-convection for increasing Péclet numbers.
<code>examples.convection.powerLaw1D</code>	
<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.
<code>examples.convection.test</code>	

24.4.1 examples.convection.advection

Modules

<code>examples.convection.advection.explicitUpwind</code>	This example shows the failure of advecting a square pulse with a first order explicit upwind scheme.
<code>examples.convection.advection.implicitUpwind</code>	This example shows the failure of advecting a square pulse with a first order implicit upwind scheme.
<code>examples.convection.advection.vanLeerUpwind</code>	This example demonstrates the use of the <code>VanLeerConvectionTerm</code> as defined by http://www.gre.ac.uk/~physica/phy2.12/theory/node173.htm

examples.convection.advection.explicitUpwind

This example shows the failure of advecting a square pulse with a first order explicit upwind scheme.

examples.convection.advection.implicitUpwind

This example shows the failure of advecting a square pulse with a first order implicit upwind scheme.

examples.convection.advection.vanLeerUpwind

This example demonstrates the use of the *VanLeerConvectionTerm* as defined by <http://www.gre.ac.uk/~physical/phy2.12/theory/node173.htm>

In this example a square wave is advected. The Van Leer discretization should in theory do a good job of preserving the shape of the wave. This may or may not be happening in this case. This example needs further testing.

The test case is mainly to check that the periodic mesh is working correctly. We advect the wave on different meshes one periodic and one non-periodic but twice as long. The results are then compared. The periodic wave wraps around the mesh.

```
>>> for step in range(steps):
...     eq1.solve(var=var1, dt=dt, solver=DefaultAsymmetricSolver(tolerance=1e-11,
↪ iterations=10000))
...     eq2.solve(var=var2, dt=dt, solver=DefaultAsymmetricSolver(tolerance=1e-11,
↪ iterations=10000))
```

```
>>> print(numerix.allclose(var1.globalValue[nx // 2:3 * nx // 4],
...                          var2.globalValue[:nx // 4], atol=1e-6))
1
```

Currently after 20 steps the wave has lost 23% of its height. Van Leer should do better than this.

```
>>> print(var1.max() > 0.77)
1
```

24.4.2 examples.convection.exponential1D

Modules

<code>examples.convection.exponential1D.cylindricalMesh1D</code>	This example solves the steady-state cylindrical convection-diffusion equation given by
<code>examples.convection.exponential1D.cylindricalMesh1DNonUniform</code>	This example solves the steady-state cylindrical convection-diffusion equation given by
<code>examples.convection.exponential1D.mesh1D</code>	Solve the steady-state convection-diffusion equation in one dimension.
<code>examples.convection.exponential1D.tri2D</code>	This example solves the steady-state convection-diffusion equation as described in examples.convection.exponential1D.mesh1D but uses a <i>Tri2D</i> mesh.

examples.convection.exponential1D.cylindricalMesh1D

This example solves the steady-state cylindrical convection-diffusion equation given by

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = (10,)$, or

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,)
```

We define a 1D cylindrical mesh representing an annulus

```
>>> from fipy import CellVariable, CylindricalGrid1D, DiffusionTerm,
↳ ExponentialConvectionTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> r0 = 1.
>>> r1 = 2.
>>> nr = 100
>>> mesh = CylindricalGrid1D(dr=(r1 - r0) / nr, nr=nr) + ((r0,),)
```

The solution variable is initialized to valueLeft:

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

```
>>> var = CellVariable(mesh=mesh, name = "variable")
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } r = r_0, \\ 1 & \text{at } r = r_1, \end{cases}$$

with

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

The equation is created with the *DiffusionTerm* and *ExponentialConvectionTerm*.

```
>>> 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

$$\phi = \exp \frac{u}{D} (r_1 - r) \left(\frac{\text{ei} \frac{ur_0}{D} - \text{ei} \frac{ur}{D}}{\text{ei} \frac{ur_0}{D} - \text{ei} \frac{ur_1}{D}} \right)$$

or


```

>>> axis = 0
>>> try:
...     from scipy.special import expi
...     r = mesh.cellCenters[axis]
...     AA = numerix.exp(convCoeff[axis] / diffCoeff * (r1 - r))
...     BB = expi(convCoeff[axis] * r0 / diffCoeff) - expi(convCoeff[axis] * r /
↳diffCoeff)
...     CC = expi(convCoeff[axis] * r0 / diffCoeff) - expi(convCoeff[axis] * r1 /
↳diffCoeff)
...     analyticalArray = AA * BB / CC
... except ImportError:
...     print("The SciPy library is unavailable. It is required for testing purposes.")

```

```

>>> print(var.allclose(analyticalArray, atol=1e-3))
1

```

If the problem is run interactively, we can view the result:

```

>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var)
...     viewer.plot()

```

examples.convection.exponential1D.cylindricalMesh1DNonUniform

This example solves the steady-state cylindrical convection-diffusion equation given by

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = (10, ,)$, or

```

>>> diffCoeff = 1.
>>> convCoeff = ((10.,),)

```

We define a 1D cylindrical mesh representing an annulus. The mesh has a non-constant cell spacing.

```

>>> from fipy import CellVariable, CylindricalGrid1D, DiffusionTerm,
↳ExponentialConvectionTerm, Viewer
>>> from fipy.tools import numerix

```

```

>>> r0 = 1.
>>> r1 = 2.
>>> nr = 100
>>> Rratio = (r1 / r0)**(1 / float(nr))
>>> dr = r0 * (Rratio - 1) * Rratio**numerix.arange(nr)
>>> mesh = CylindricalGrid1D(dr=dr) + ((r0,),)

```

```

>>> valueLeft = 0.
>>> valueRight = 1.

```

The solution variable is initialized to valueLeft:

```
>>> var = CellVariable(mesh=mesh, name = "variable")
```

and impose the boundary conditions

with

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

The equation is created with the *DiffusionTerm* and *ExponentialConvectionTerm*.

```
>>> 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

$$\phi = \exp \frac{u}{D} (r_1 - r) \left(\frac{\text{ei} \frac{ur_0}{D} - \text{ei} \frac{ur}{D}}{\text{ei} \frac{ur_0}{D} - \text{ei} \frac{ur_1}{D}} \right)$$

or

```
>>> axis = 0
```

```
>>> try:
...     U = convCoeff[0][0]
...     from scipy.special import expi
...     r = mesh.cellCenters[axis]
...     AA = numerix.exp(U / diffCoeff * (r1 - r))
...     BB = expi(U * r0 / diffCoeff) - expi(U * r / diffCoeff)
...     CC = expi(U * r0 / diffCoeff) - expi(U * r1 / diffCoeff)
...     analyticalArray = AA * BB / CC
... except ImportError:
...     print("The SciPy library is unavailable. It is required for testing purposes.")
```

```
>>> print(var.allclose(analyticalArray, atol=1e-3))
1
```

If the problem is run interactively, we can view the result:

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

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 $\vec{u} = 10\hat{i}$, 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()
```

examples.convection.exponential1D.tri2D

This example solves the steady-state convection-diffusion equation as described in [examples.convection.exponential1D.mesh1D](#) but uses a *Tri2D* mesh.

Here the axes are reversed ($n_x = 1, n_y = 1000$) and

$$\vec{u} = (0, 10)$$

```
>>> from fipy import CellVariable, Tri2D, DiffusionTerm, ExponentialConvectionTerm, \
...     DefaultAsymmetricSolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 10.
>>> nx = 1
>>> ny = 1000
>>> mesh = Tri2D(dx = L / ny, dy = L / ny, nx = nx, ny = ny)
```

```
>>> valueBottom = 0.
>>> valueTop = 1.
```

```
>>> var = CellVariable(name = "concentration",
...                     mesh = mesh,
...                     value = valueBottom)
```

```
>>> var.constrain(valueBottom, mesh.facesBottom)
>>> var.constrain(valueTop, mesh.facesTop)
```

```
>>> diffCoeff = 1.
>>> convCoeff = numerix.array(((0.), (10.)))
```

```
>>> eq = (DiffusionTerm(coeff=diffCoeff)
...       + ExponentialConvectionTerm(coeff=convCoeff))
```

It should be possible to drive this steady-state solution to high accuracy.

```
>>> solver = DefaultAsymmetricSolver(tolerance=1e-10, iterations=10000)
>>> eq.solve(var=var, solver=solver)
```

The analytical solution test for this problem is given by:

```
>>> axis = 1
>>> y = mesh.cellCenters[axis]
>>> CC = 1. - numerix.exp(-convCoeff[axis] * y / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
```

```
>>> print(var.allclose(analyticalArray, rtol=1e-6, atol=1e-6))
True
```

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

24.4.3 examples.convection.exponential1DBack

Modules

```
examples.convection.exponential1DBack.
mesh1D
```

This example solves the steady-state convection-diffusion equation as described in [examples.convection.exponential1D.mesh1D](#) but with $\vec{u} = (-10,)$.

examples.convection.exponential1DBack.mesh1D

This example solves the steady-state convection-diffusion equation as described in [examples.convection.exponential1D.mesh1D](#) but with $\vec{u} = (-10,)$.

```
>>> from fipy import CellVariable, Grid1D, DiffusionTerm, ExponentialConvectionTerm,
↳ DefaultAsymmetricSolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 10.
>>> nx = 1000
>>> mesh = Grid1D(dx = L / nx, nx = nx)
```

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

```
>>> var = CellVariable(name = "concentration",
...                     mesh = mesh,
...                     value = valueLeft)
```

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

```
>>> diffCoeff = 1.
>>> convCoeff = (-10.,)
```

```
>>> eq = (DiffusionTerm(coeff=diffCoeff)
...      + ExponentialConvectionTerm(coeff=convCoeff))
```

```
>>> eq.solve(var = var,
...          solver = DefaultAsymmetricSolver(tolerance=1.e-15, iterations=10000))
```

We test the solution against the analytical result:

```
>>> 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, rtol = 1e-10, atol = 1e-10))
1
```

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

24.4.4 examples.convection.exponential1DSource

Modules

<code>examples.convection.exponential1DSource.mesh1D</code>	Solve the steady-state convection-diffusion equation with a constant source.
<code>examples.convection.exponential1DSource.tri2D</code>	This example solves the steady-state convection-diffusion equation as described in examples.convection.exponential1D.mesh1D but uses a constant source value such that,

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=100000))
```

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()

```

examples.convection.exponential1DSource.tri2D

This example solves the steady-state convection-diffusion equation as described in [examples.convection.exponential1D.mesh1D](#) but uses a constant source value such that,

$$S_c = 1.$$

Here the axes are reversed ($n_x = 1, n_y = 1000$) and

$$\vec{u} = (0, 10)$$

```

>>> from fipy import CellVariable, Tri2D, DiffusionTerm, ExponentialConvectionTerm, \
↳ DefaultAsymmetricSolver, Viewer
>>> from fipy.tools import numerix

```

```

>>> L = 10.
>>> nx = 1
>>> ny = 1000
>>> mesh = Tri2D(dx = L / ny, dy = L / ny, nx = nx, ny = ny)

```

```

>>> valueBottom = 0.
>>> valueTop = 1.

```

```

>>> var = CellVariable(name = "concentration",
...                     mesh = mesh,
...                     value = valueBottom)

```

```

>>> var.constrain(valueBottom, mesh.facesBottom)
>>> var.constrain(valueTop, mesh.facesTop)

```

```

>>> diffCoeff = 1.
>>> convCoeff = (0., 10.)
>>> sourceCoeff = 1.

```



```
>>> eq = (-sourceCoeff - DiffusionTerm(coeff = diffCoeff)
...       - ExponentialConvectionTerm(coeff = convCoeff))
```

```
>>> eq.solve(var=var,
...          solver=DefaultAsymmetricSolver(tolerance=1.e-8, iterations=10000))
```

The analytical solution test for this problem is given by:

```
>>> axis = 1
>>> y = mesh.cellCenters[axis]
>>> AA = -sourceCoeff * y / convCoeff[axis]
>>> BB = 1. + sourceCoeff * L / convCoeff[axis]
>>> CC = 1. - numerix.exp(-convCoeff[axis] * y / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = AA + BB * CC / DD
>>> print(var.allclose(analyticalArray, atol = 1e-5))
1
```

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

24.4.5 examples.convection.exponential2D

Modules

<code>examples.convection.exponential2D.cylindricalMesh2D</code>	This example solves the steady-state cylindrical convection-diffusion equation given by:
<code>examples.convection.exponential2D.cylindricalMesh2DNonUniform</code>	This example solves the steady-state cylindrical convection-diffusion equation given by:
<code>examples.convection.exponential2D.mesh2D</code>	This example solves the steady-state convection-diffusion equation as described in examples.convection.exponential1D.mesh1D on a 2D mesh with <code>nx = 10</code> and <code>ny = 10</code> :
<code>examples.convection.exponential2D.tri2D</code>	This example solves the steady-state convection-diffusion equation as described in examples.convection.exponential1D.mesh1D with <code>nx = 10</code> and <code>ny = 10</code> .

examples.convection.exponential2D.cylindricalMesh2D

This example solves the steady-state cylindrical convection-diffusion equation given by:

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = (10,)$, or

```
>>> diffCoeff = 1.
>>> convCoeff = ((10.,), (0.,))
```

We define a 2D cylindrical mesh representing an annulus. The mesh is a pseudo 1D mesh, but is a good test case for the `CylindricalGrid2D()` mesh.

```
>>> from fipy import CellVariable, CylindricalGrid2D, DiffusionTerm, \
    ExponentialConvectionTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> r0 = 1.
>>> r1 = 2.
>>> nr = 100
>>> mesh = CylindricalGrid2D(dr=(r1 - r0) / nr, dz=1., nr=nr, nz=1) + ((r0,),)
```

The solution variable is initialized to valueLeft:

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> var = CellVariable(mesh=mesh, name = "variable")
```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } r = r_0, \\ 1 & \text{at } r = r_1, \end{cases}$$

with

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

The equation is created with the `DiffusionTerm` and `ExponentialConvectionTerm`.

```
>>> 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

$$\phi = \exp \frac{u}{D} (r_1 - r) \left(\frac{\exp \frac{ur_0}{D} - \exp \frac{ur}{D}}{\exp \frac{ur_0}{D} - \exp \frac{ur_1}{D}} \right)$$

or

```
>>> axis = 0
>>> try:
...     from scipy.special import expi
...     U = convCoeff[0][0]
...     r = mesh.cellCenters[axis]
...     AA = numerix.exp(U / diffCoeff * (r1 - r))
...     BB = expi(U * r0 / diffCoeff) - expi(U * r / diffCoeff)
```

(continues on next page)

(continued from previous page)

```

...     CC = expi(U * r0 / diffCoeff) - expi(U * r1 / diffCoeff)
...     analyticalArray = AA * BB / CC
... except ImportError:
...     print("The SciPy library is unavailable. It is required for testing purposes.")

```

```

>>> print(var.allclose(analyticalArray, atol=1e-3))
1

```

If the problem is run interactively, we can view the result:

```

>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var)
...     viewer.plot()

```

examples.convection.exponential2D.cylindricalMesh2DNonUniform

This example solves the steady-state cylindrical convection-diffusion equation given by:

$$\nabla \cdot (D \nabla \phi + \vec{u} \phi) = 0$$

with coefficients $D = 1$ and $\vec{u} = (10,)$, or

```

>>> diffCoeff = 1.
>>> convCoeff = ((10.), (0.))

```

We define a 2D cylindrical mesh representing an annulus. The mesh is a pseudo-1D mesh, but is a good test case for the `CylindricalGrid2D()` mesh. The mesh has a non-constant cell spacing.

```

>>> from fipy import CellVariable, CylindricalGrid2D, DiffusionTerm, \
↳ ExponentialConvectionTerm, Viewer
>>> from fipy.tools import numerix

```

```

>>> r0 = 1.
>>> r1 = 2.
>>> nr = 100
>>> Rratio = (r1 / r0)**(1 / float(nr))
>>> dr = r0 * (Rratio - 1) * Rratio**numerix.arange(nr)
>>> mesh = CylindricalGrid2D(dr=dr, dz=1., nz=1) + ((r0.), (0.))

```

The solution variable is initialized to valueLeft:

```

>>> valueLeft = 0.
>>> valueRight = 1.
>>> var = CellVariable(mesh=mesh, name = "variable")

```

and impose the boundary conditions

$$\phi = \begin{cases} 0 & \text{at } r = r_0, \\ 1 & \text{at } r = r_1, \end{cases}$$

with

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

The equation is created with the *DiffusionTerm* and *ExponentialConvectionTerm*.

```
>>> 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

$$\phi = \exp \frac{u}{D} (r_1 - r) \left(\frac{\text{ei} \frac{ur_0}{D} - \text{ei} \frac{ur}{D}}{\text{ei} \frac{ur_0}{D} - \text{ei} \frac{ur_1}{D}} \right)$$

```
>>> axis = 0
>>> try:
...     from scipy.special import expi
...     r = mesh.cellCenters[axis]
...     U = convCoeff[0][0]
...     AA = numerix.exp(U / diffCoeff * (r1 - r))
...     BB = expi(U * r0 / diffCoeff) - expi(U * r / diffCoeff)
...     CC = expi(U * r0 / diffCoeff) - expi(U * r1 / diffCoeff)
...     analyticalArray = AA * BB / CC
... except ImportError:
...     print("The SciPy library is unavailable. It is required for testing purposes.")
```

```
>>> print(var.allclose(analyticalArray, atol=1e-3))
1
```

If the problem is run interactively, we can view the result:

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

examples.convection.exponential2D.mesh2D

This example solves the steady-state convection-diffusion equation as described in *examples.convection.exponential1D.mesh1D* on a 2D mesh with $n_x = 10$ and $n_y = 10$:

```
>>> from fipy import CellVariable, Grid2D, DiffusionTerm, ExponentialConvectionTerm,
...     DefaultAsymmetricSolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 10.
>>> nx = 10
>>> ny = 10
>>> mesh = Grid2D(L / nx, L / ny, nx, ny)
```

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

```
>>> var = CellVariable(name = "concentration",
...                     mesh = mesh,
...                     value = valueLeft)
```

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

```
>>> diffCoeff = 1.
>>> convCoeff = (10., 0.)
```

```
>>> eq = DiffusionTerm(coeff=diffCoeff) + ExponentialConvectionTerm(coeff=convCoeff)
```

```
>>> eq.solve(var = var,
...          solver=DefaultAsymmetricSolver(tolerance=1.e-15, iterations=100000))
```

We test the solution against the analytical result:

```
>>> 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, rtol = 1e-10, atol = 1e-10))
1
```

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

examples.convection.exponential2D.tri2D

This example solves the steady-state convection-diffusion equation as described in [examples.convection.exponential1D.mesh1D](#) with $nx = 10$ and $ny = 10$.

```
>>> from fipy import CellVariable, Tri2D, DiffusionTerm, ExponentialConvectionTerm,
↳ Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 10.
>>> nx = 10
>>> ny = 10
>>> mesh = Tri2D(L / nx, L / ny, nx, ny)
```

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

```
>>> var = CellVariable(name = "concentration",
...                     mesh = mesh,
...                     value = valueLeft)
```

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

```
>>> diffCoeff = 1.
>>> convCoeff = (10., 0.)
```

```
>>> eq = (DiffusionTerm(coeff=diffCoeff)
...      + ExponentialConvectionTerm(coeff=convCoeff))
```

It should be possible to drive this steady-state solution to high accuracy.

```
>>> solver = eq.getDefaultSolver(tolerance=1e-10)
>>> eq.solve(var=var, solver=solver)
```

The analytical solution test for this problem is given by:

```
>>> 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, rtol=1e-10, atol=1e-10))
True
```

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

24.4.6 examples.convection.peclet

This example tests diffusion-convection for increasing Péclet numbers. This test case has been introduced because some solvers were not working with Péclet numbers over 1.

```
>>> from fipy import CellVariable, Grid1D, TransientTerm, DiffusionTerm,
...     PowerLawConvectionTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 1.
>>> nx = 1000
>>> dx = L / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

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

```
>>> var = CellVariable(name = "solution variable", mesh=mesh, value=valueLeft)
```

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

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars = var)
```

```
>>> convCoeff = 1.0
>>> peclet = 1e-3
>>> allcloseList = []
>>> from fipy import input
>>> while peclet < 1e4:
...     var[:] = valueLeft
...     diffCoeff = convCoeff * dx / peclet
...     eq = (TransientTerm(1e-4)
...           == DiffusionTerm(coeff=diffCoeff)
...           + PowerLawConvectionTerm(coeff=(convCoeff,)))
...     eq.solve(var=var, dt=1.)
...     x = mesh.cellCenters[0]
...     arg0 = -convCoeff * x / diffCoeff
...     arg0 = numerix.where(arg0 < -200, -200, arg0)
...     arg1 = -convCoeff * L / diffCoeff
...     arg1 = (arg1 >= -200) * (arg1 + 200) - 200
...     CC = 1. - numerix.exp(arg0)
...     DD = 1. - numerix.exp(arg1)
...     analyticalArray = CC / DD
...     allcloseList.append(var.allclose(CC / DD, rtol = 1e-2, atol = 1e-2).value)
...     if __name__ == '__main__':
...         viewer.plot()
...         input("Peclet number: " + str(peclet) + ", press key")
...     peclet *= 10
```

```
>>> print(allcloseList)
[True, True, True, True, True, True, True]
```

24.4.7 examples.convection.powerLaw1D

Modules

`examples.convection.powerLaw1D.mesh1D`

This example solves the steady-state convection-diffusion equation as described in [examples.convection.exponential1D.mesh1D](#) but uses the `PowerLawConvectionTerm` rather than the `ExponentialConvectionTerm`.

`examples.convection.powerLaw1D.tri2D`

This example solves the steady-state convection-diffusion equation as described in [examples.convection.exponential1D.mesh1D](#) but uses the `PowerLawConvectionTerm` rather than the `ExponentialConvectionTerm` instantiator.

examples.convection.powerLaw1D.mesh1D

This example solves the steady-state convection-diffusion equation as described in [examples.convection.exponential1D.mesh1D](#) but uses the `PowerLawConvectionTerm` rather than the `ExponentialConvectionTerm`.

```
>>> from fipy import CellVariable, Grid1D, DiffusionTerm, PowerLawConvectionTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 10.
>>> nx = 1000
>>> mesh = Grid1D(dx = L / nx, nx = nx)
```

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

```
>>> var = CellVariable(name = "concentration",
...                     mesh = mesh,
...                     value = valueLeft)
```

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

```
>>> diffCoeff = 1.
>>> convCoeff = (10.,)
```

```
>>> eq = (DiffusionTerm(coeff=diffCoeff)
...      + PowerLawConvectionTerm(coeff=convCoeff))
```

```
>>> eq.solve(var = var)
```

We test the solution against the analytical result:

```
>>> axis = 0
>>> x = mesh.cellCenters[axis]
```

(continues on next page)

(continued from previous page)

```
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> print(var.allclose(analyticalArray, rtol = 1e-2, atol = 1e-2))
1
```

If the problem is run interactively, we can view the result:

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

examples.convection.powerLaw1D.tri2D

This example solves the steady-state convection-diffusion equation as described in [examples.convection.exponential1D.mesh1D](#) but uses the *PowerLawConvectionTerm* rather than the *ExponentialConvectionTerm* instantiator.

```
>>> from fipy import CellVariable, Grid2D, DiffusionTerm, PowerLawConvectionTerm,
↳ DefaultAsymmetricSolver, Viewer
>>> from fipy.tools import numerix
```

```
>>> L = 10.
>>> nx = 1000
>>> mesh = Grid2D(dx = L / nx, nx = nx)
```

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

```
>>> var = CellVariable(name = "concentration",
...                     mesh = mesh,
...                     value = valueLeft)
```

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

```
>>> diffCoeff = 1.
>>> convCoeff = (10., 0.)
```

```
>>> eq = (DiffusionTerm(coeff=diffCoeff)
...       + PowerLawConvectionTerm(coeff=convCoeff))
```

```
>>> eq.solve(var=var,
...           solver=DefaultAsymmetricSolver(tolerance=1.e-15, iterations=2000))
```

The analytical solution test for this problem is given by:

```
>>> axis = 0
>>> x = mesh.cellCenters[axis]
```

(continues on next page)

(continued from previous page)

```
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> print(var.allclose(analyticalArray, rtol = 1e-2, atol = 1e-2))
1
```

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

24.4.8 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,

$$0 = \frac{\partial^2 C}{\partial x^2} - P \frac{\partial C}{\partial x} - DC \quad 0 < x < 1$$

$$x = 0 : P = -\frac{\partial C}{\partial x} + PC$$

$$x = 1 : \frac{\partial C}{\partial x} = 0$$

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
```

24.4.9 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")
```

24.4.10 examples.convection.test

24.5 examples.diffusion

Modules

<code>examples.diffusion.anisotropy</code>	Solve the diffusion equation with an anisotropic diffusion coefficient.
<code>examples.diffusion.circle</code>	Solve the diffusion equation in a circular domain meshed with triangles.
<code>examples.diffusion.circleQuad</code>	Solve the diffusion equation in a circular domain meshed with quadrangles.
<code>examples.diffusion.coupled</code>	Solve the biharmonic equation as a coupled pair of diffusion equations.
<code>examples.diffusion.electrostatics</code>	Solve the Poisson equation in one dimension.
<code>examples.diffusion.explicit</code>	
<code>examples.diffusion.mesh1D</code>	Solve a one-dimensional diffusion equation under different conditions.
<code>examples.diffusion.mesh20x20</code>	Solve a two-dimensional diffusion problem in a square domain.
<code>examples.diffusion.mesh20x20Coupled</code>	Solve a coupled set of diffusion equations in two dimensions.
<code>examples.diffusion.nthOrder</code>	
<code>examples.diffusion.steadyState</code>	
<code>examples.diffusion.test</code>	Run all the test cases in examples/diffusion/
<code>examples.diffusion.variable</code>	This example is a 1D steady state diffusion test case as in <code>./examples/diffusion/variable/mesh2x1/input.py</code> with then number of cells set to $nx = 10$.

24.5.1 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
```

```
>>> 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
```

24.5.2 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.)
```

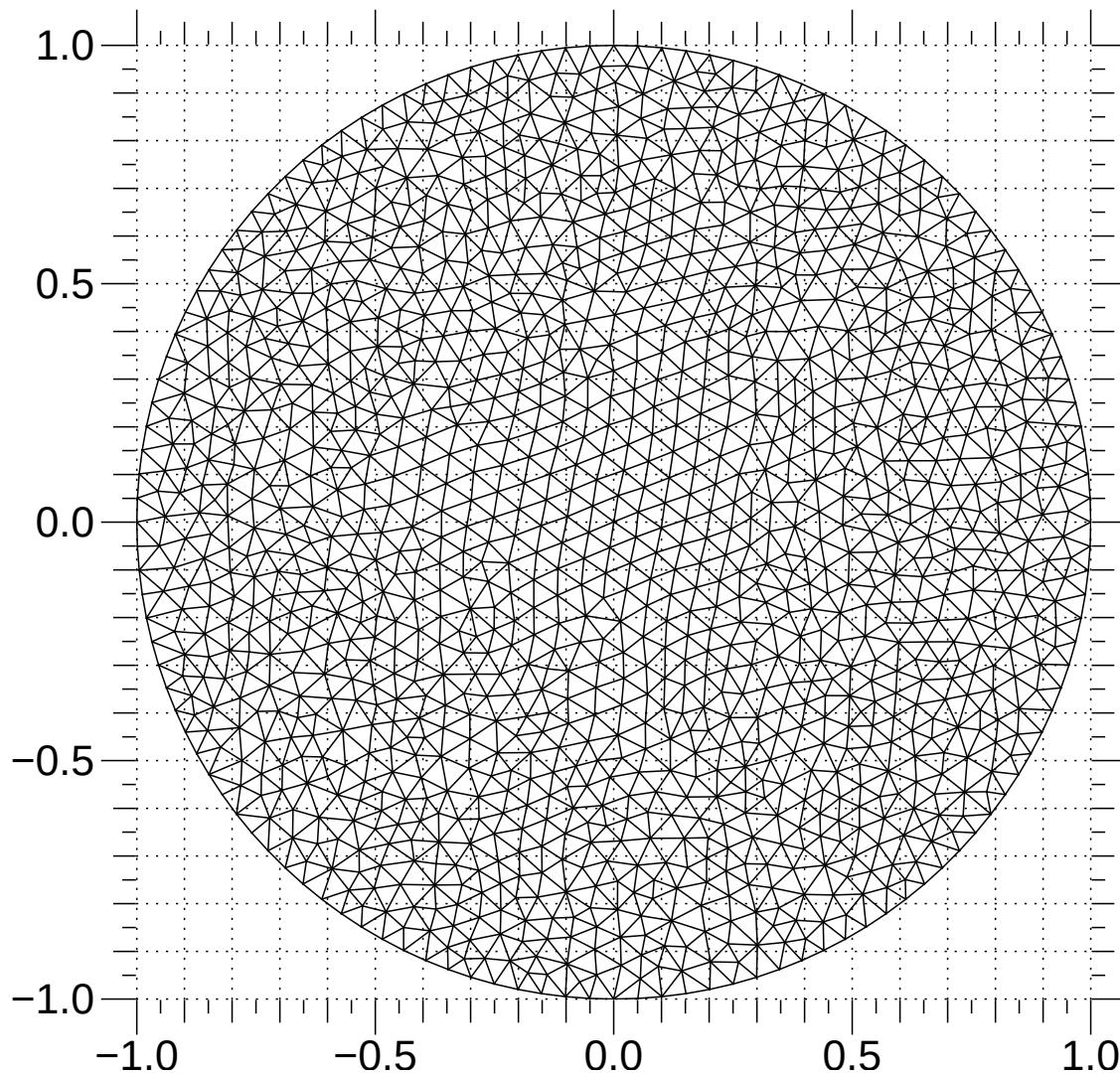
We can now create a Viewer to see the mesh

```
>>> viewer = None
>>> from fipy import input
```

(continues on next page)

(continued from previous page)

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=phi, datamin=-1, datamax=1.)
...     viewer.plotMesh()
```



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
```

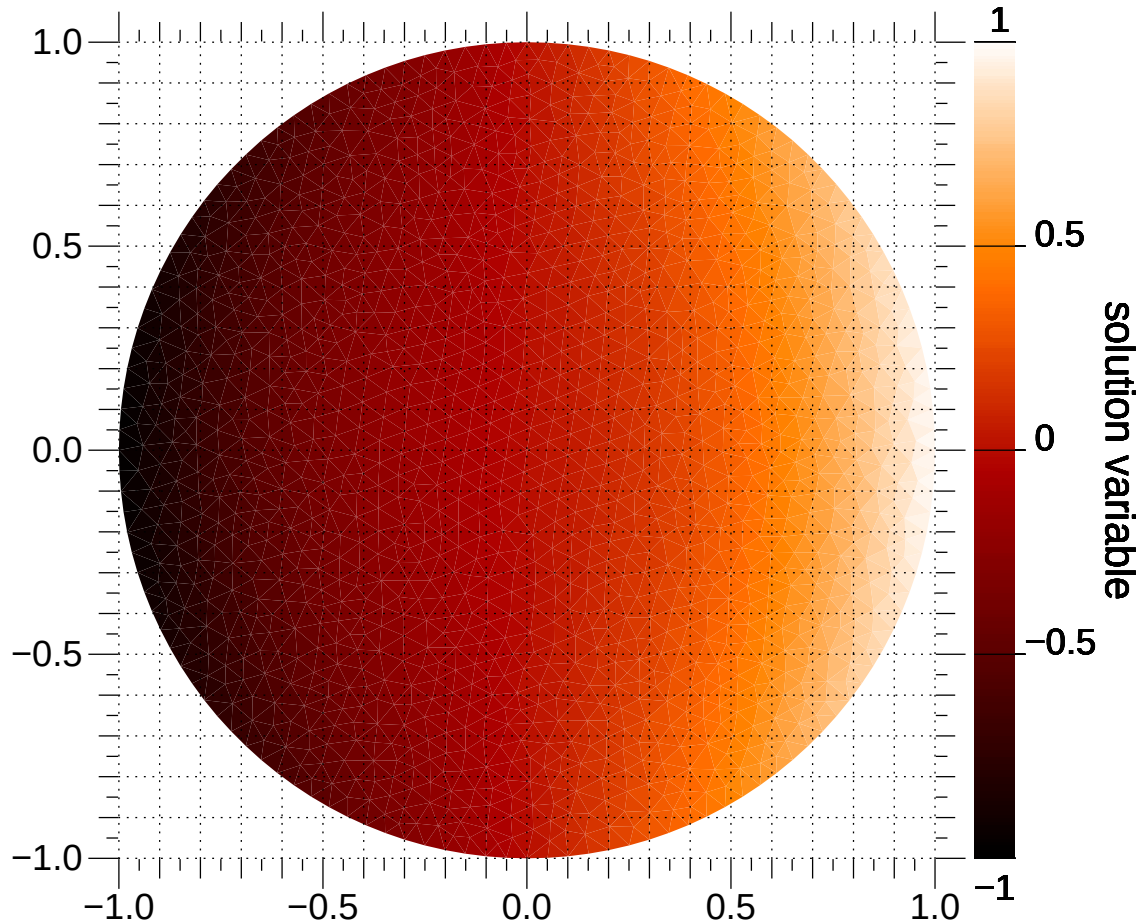
```
>>> phi.constrain(X, mesh.exteriorFaces)
```

We first step through the transient problem


```

>>> timeStepDuration = 10 * 0.9 * cellSize**2 / (2 * D)
>>> steps = 10
>>> for step in range(steps):
...     eq.solve(var=phi,
...               dt=timeStepDuration)
...     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.	
0.229687917881182	0.00757854476106331			
0.0442864953037566	0.79191893162384	0.0375859836421991	-0.	
0.773936613923853	-0.205560697612547			
0.0246775505084069	0.771959648896982	0.020853932412869	-0.	
0.723540342405813	-0.182589694334729			
0.223345558247991	-0.807931073108895	0.203035857140125	-0.	

(continues on next page)

(continued from previous page)

→ 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		
-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.

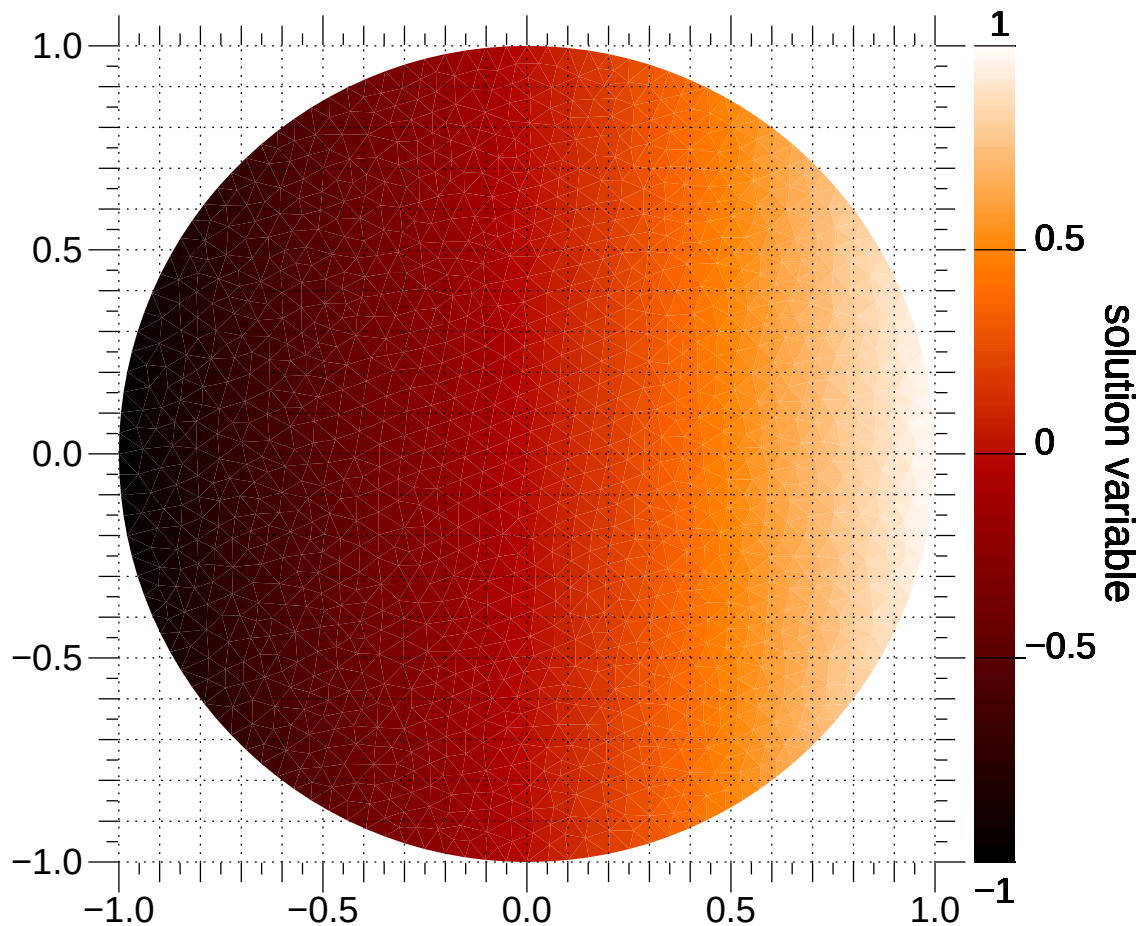
```
>>> DiffusionTerm(coeff=D).solve(var=phi)
```

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...")
```



24.5.3 examples.diffusion.circleQuad

Solve the diffusion equation in a circular domain meshed with quadrangles.

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
```

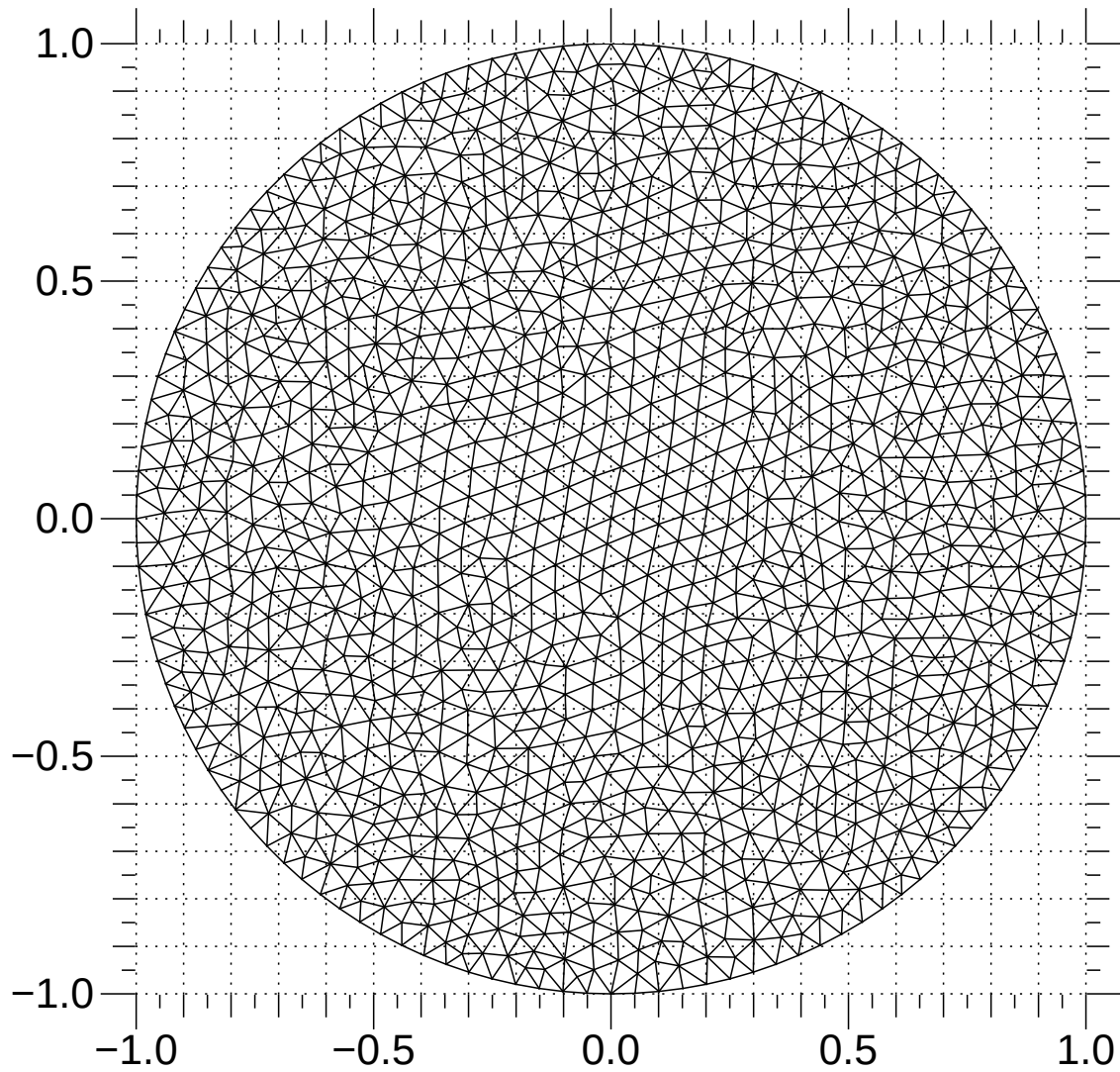
```
>>> mesh = Gmsh2D('''
...     cellSize = %(cellSize)g;
...     radius = %(radius)g;
...     Point(1) = {0, 0, 0, cellSize};
...     Point(2) = {-radius, 0, 0, cellSize};
...     Point(3) = {0, radius, 0, cellSize};
...     Point(4) = {radius, 0, 0, cellSize};
...     Point(5) = {0, -radius, 0, cellSize};
...     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};
...     Recombine Surface{11};
...     ''' % 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__':
...     viewer = Viewer(vars=phi, datamin=-1, datamax=1.)
...     viewer.plotMesh()
...     input("Irregular circular mesh. Press <return> to proceed...")
```



We set up a transient diffusion equation

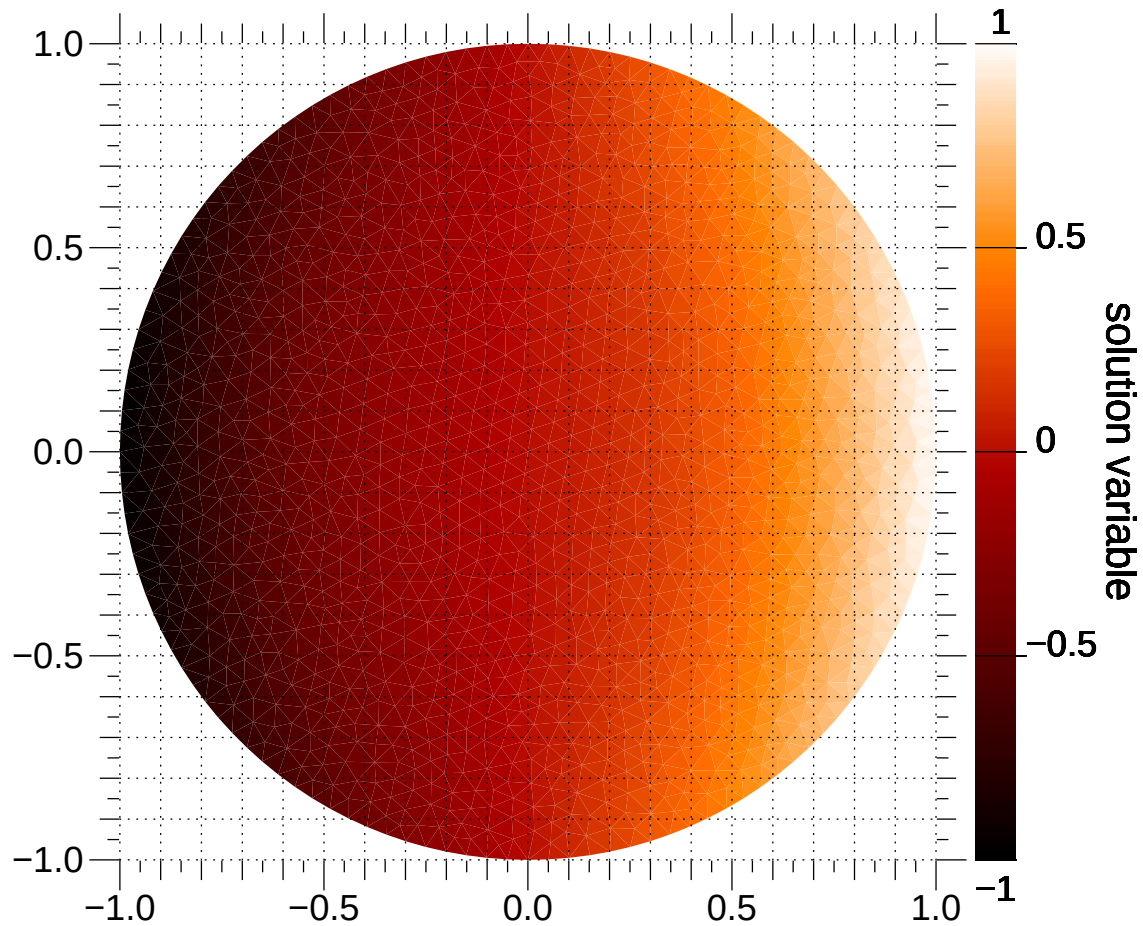
```
>>> D = 1.
>>> eq = TransientTerm() == DiffusionTerm(coeff=D)
```

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
>>> for step in range(steps):
...     eq.solve(var=phi,
...               dt=timeStepDuration)
...     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		
-0.78996770899909		-0.0173672729866294	-0.693887874335319	-1.

(continues on next page)

(continued from previous page)

→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.

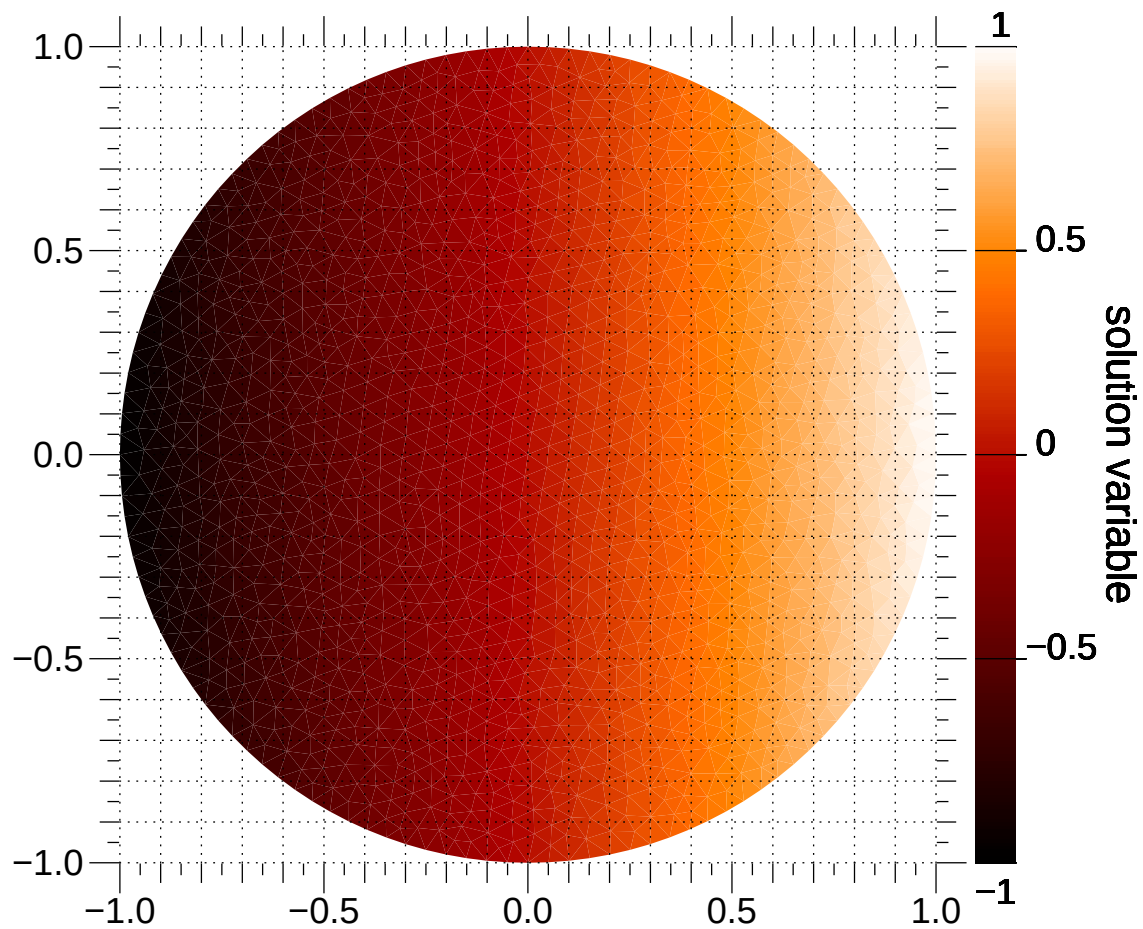
```
>>> DiffusionTerm(coeff=D).solve(var=phi)
```

The values at the elements should be equal to their x coordinate

```
>>> print(phi.allclose(x, atol = 0.035))
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...")
```



24.5.4 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))
```

```
>>> 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
```

```
>>> 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]))
```

```
>>> 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.

24.5.5 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

$$\psi(x) = \frac{x^2}{2} - 2x$$

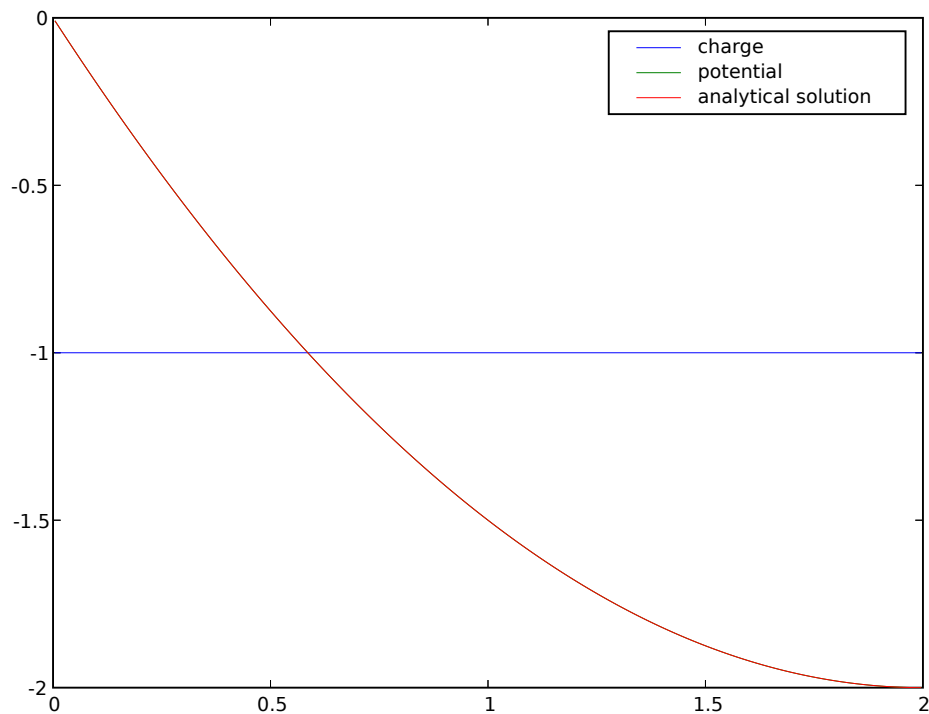
```
>>> x = mesh.cellCenters[0]
>>> analytical = CellVariable(mesh=mesh, name="analytical solution",
...                           value=(x**2)/2 - 2*x)
```

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

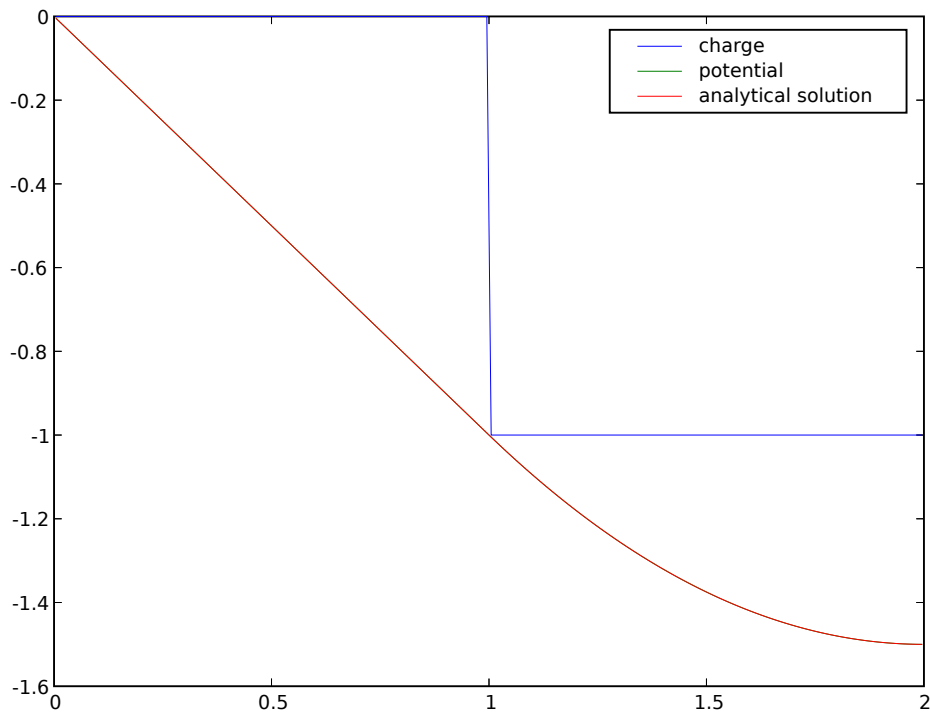
$$\psi(x) = \begin{cases} -x & \text{for } x \leq L/2, \\ \frac{(x-1)^2}{2} - x & \text{for } x > L/2. \end{cases}$$

```
>>> 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

$$\psi(x) = \begin{cases} \frac{x^2}{2} - x & \text{for } x \leq L/2, \\ -\frac{1}{2} & \text{for } x > L/2. \end{cases}$$

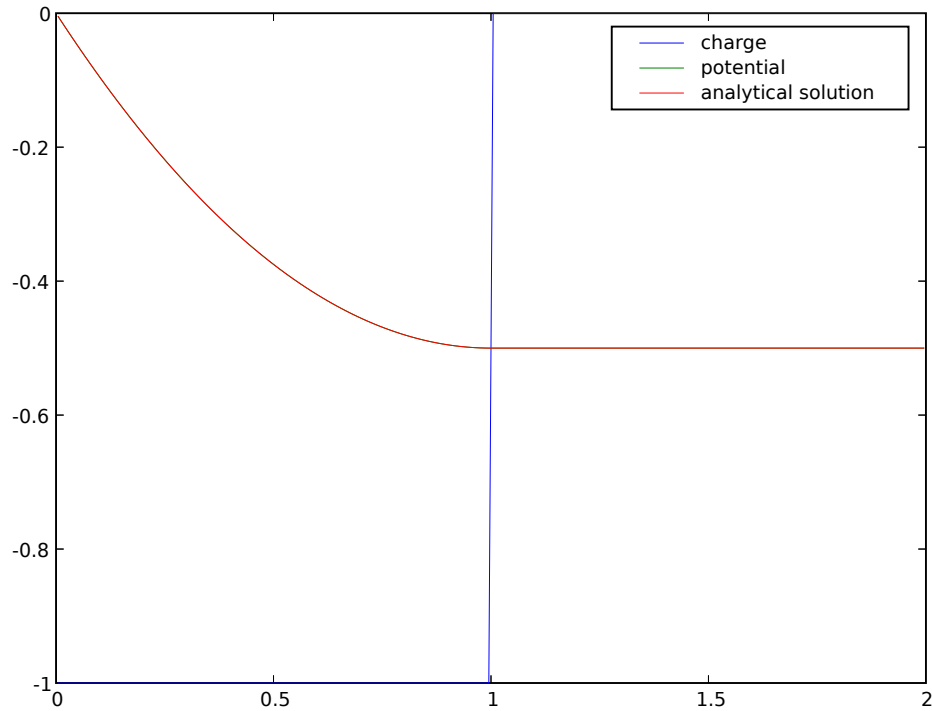
We again verify that the correct equilibrium is attained

```
>>> 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()
```



24.5.6 examples.diffusion.explicit

Modules

<code>examples.diffusion.explicit.mesh1D</code>	This input file again solves a 1D diffusion problem as in <code>examples.diffusion.steadyState.mesh1D</code> , the difference being that this transient example is solved explicitly.
<code>examples.diffusion.explicit.mixedelement</code>	This input file again solves an explicit 1D diffusion problem as in <code>./examples/diffusion/mesh1D.py</code> but on a mesh with both square and triangular elements.
<code>examples.diffusion.explicit.test</code>	
<code>examples.diffusion.explicit.tri2D</code>	This input file again solves a 1D diffusion problem as in <code>./examples/diffusion/steadyState/mesh1D.py</code> .

examples.diffusion.explicit.mesh1D

This input file again solves a 1D diffusion problem as in `examples.diffusion.steadyState.mesh1D`, the difference being that this transient example is solved explicitly.

We create a 1D mesh:

```
>>> from fipy import CellVariable, Grid1D, TransientTerm, ExplicitDiffusionTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> nx = 100
>>> dx = 1.
>>> mesh = Grid1D(dx = dx, nx = nx)
```

and we initialize a `CellVariable` to `initialValue`:

```
>>> valueLeft = 0.
>>> initialValue = 1.
>>> var = CellVariable(
...     name = "concentration",
...     mesh = mesh,
...     value = initialValue)
```

The transient diffusion equation

$$\frac{\partial \phi}{\partial t} = \nabla \cdot (D \nabla \phi)$$

is represented by a `TransientTerm` and an `ExplicitDiffusionTerm`.

We take the diffusion coefficient $D = 1$

```
>>> diffusionCoeff = 1.
```

We build the equation:

```
>>> eq = TransientTerm() == ExplicitDiffusionTerm(coeff = diffusionCoeff)
```

and the boundary conditions:

```
>>> var.constrain(valueLeft, mesh.facesLeft)
```

In this case, many steps have to be taken to reach equilibrium. A loop is required to execute the necessary time steps:

```
>>> timeStepDuration = 0.1
>>> steps = 100
>>> for step in range(steps):
...     eq.solve(var=var, dt=timeStepDuration)
```

The analytical solution for this transient diffusion problem is given by $\phi = \text{erf}(x/2\sqrt{Dt})$.

The result is tested against the expected profile:

```
>>> Lx = nx * dx
>>> x = mesh.cellCenters[0]
>>> t = timeStepDuration * steps
>>> epsi = x / numerix.sqrt(t * diffusionCoeff)
```

(continues on next page)

(continued from previous page)

```
>>> from scipy.special import erf
>>> analyticalArray = erf(eps/2)
>>> print(var.allclose(analyticalArray, atol = 2e-3))
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars = (var,))
...     viewer.plot()
```

examples.diffusion.explicit.mixedelement

This input file again solves an explicit 1D diffusion problem as in `./examples/diffusion/mesh1D.py` but on a mesh with both square and triangular elements. The term used is the *ExplicitDiffusionTerm*. In this case many steps have to be taken to reach equilibrium. The *timeStepDuration* parameter specifies the size of each time step and *steps* is the number of time steps.

```
>>> from fipy import CellVariable, Grid2D, Tri2D, TransientTerm, ExplicitDiffusionTerm,
Viewer
>>> from fipy.tools import numerix
```

```
>>> dx = 1.
>>> dy = 1.
>>> nx = 10
```

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> D = 1.
>>> timeStepDuration = 0.01 * dx**2 / (2 * D)
>>> if __name__ == '__main__':
...     steps = 10000
... else:
...     steps = 10
```

```
>>> gridMesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=2)
>>> triMesh = Tri2D(dx=dx, dy=dy, nx=nx, ny=1) + ((dx*nx,), (0,))
>>> otherGridMesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=1) + ((dx*nx,), (1,))
>>> bigMesh = gridMesh + triMesh + otherGridMesh
```

```
>>> L = dx * nx * 2
```

```
>>> var = CellVariable(name="concentration",
...                     mesh=bigMesh,
...                     value=valueLeft)
```

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

```
>>> var.constrain(valueLeft, where=bigMesh.facesLeft)
>>> var.constrain(valueRight, where=bigMesh.facesRight)
```

In a semi-infinite domain, the analytical solution for this transient diffusion problem is given by $\phi = 1 - \text{erf}((L - x)/2\sqrt{Dt})$, which is a reasonable approximation at early times. At late times, the solution is just a straight line. If the *SciPy* library is available, the result is tested against the expected profile:

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

```
>>> if __name__ == '__main__':
...     varAnalytical = valueLeft + (valueRight - valueLeft) * x / L
...     atol = 0.2
...     else:
...         try:
...             from scipy.special import erf
...             varAnalytical = 1 - erf((L - x) / (2 * numerix.sqrt(D * t)))
...             atol = 0.03
...         except ImportError:
...             print("The SciPy library is not available to test the solution to \
...                   the transient diffusion equation")
```

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var)
```

```
>>> for step in range(steps):
...     eqn.solve(var, dt=timeStepDuration)
...     if (step % 100) == 0 and (__name__ == '__main__'):
...         viewer.plot()
```

We check the answer against the analytical result

```
>>> print(var.allclose(varAnalytical, atol=atol))
1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input('finished')
```

examples.diffusion.explicit.test

examples.diffusion.explicit.tri2D

This input file again solves a 1D diffusion problem as in *./examples/diffusion/steadyState/mesh1D.py*. The difference in this example is that the solution method is explicit. The equation used is the *ExplicitDiffusionEquation*. In this case many steps have to be taken to reach equilibrium. The *timeStepDuration* parameter specifies the size of each time step and *steps* is the number of time steps.

```
>>> dx = 1.
>>> dy = 1.
>>> nx = 10
>>> ny = 1
>>> valueLeft = 0.
```

(continues on next page)

(continued from previous page)

```
>>> valueRight = 1.
>>> timeStepDuration = 0.02
>>> steps = 10
```

A loop is required to execute the necessary time steps:

```
>>> for step in range(steps):
...     eq.solve(var, solver=solver, dt=timeStepDuration)
```

The result is again tested in the same way:

```
>>> Lx = nx * dx
>>> x = mesh.cellCenters[0]
>>> print(var.allclose(answer, rtol = 1e-8))
1
```

24.5.7 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,

$$\frac{\partial \phi}{\partial t} = D \nabla^2 \phi. \quad (24.1)$$

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 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 are 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 no-flux, $\vec{n} \cdot (D \nabla \phi)|_{\text{someFaces}} = 0$.

If you have ever tried to numerically solve Eq. (24.1), 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
```

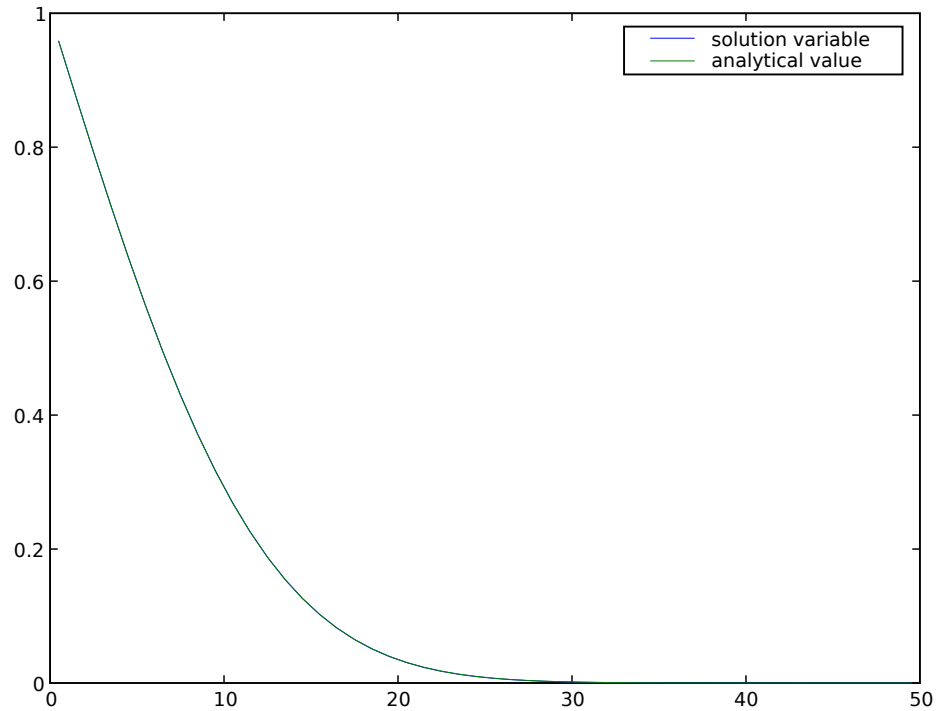
```
>>> try:
...     from scipy.special import erf
...     phiAnalytical.setValue(1 - erf(x / (2 * numerix.sqrt(D * t))))
... except ImportError:
...     print("The SciPy library is not available to test the solution to \
... the transient diffusion equation")
```

We then solve the equation by repeatedly looping in time:

```
>>> 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. (24.1) 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

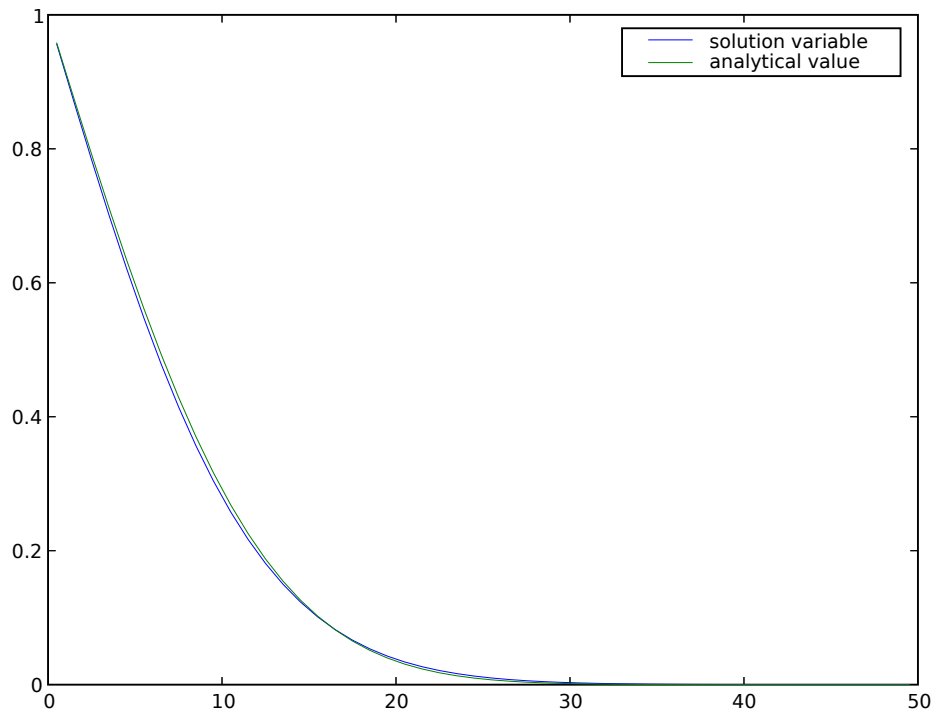
```
>>> phi.setValue(valueRight)
```

and rerun with much larger time steps

```
>>> timeStepDuration *= 10
>>> steps //= 10
>>> 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 [24]).

```
>>> 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

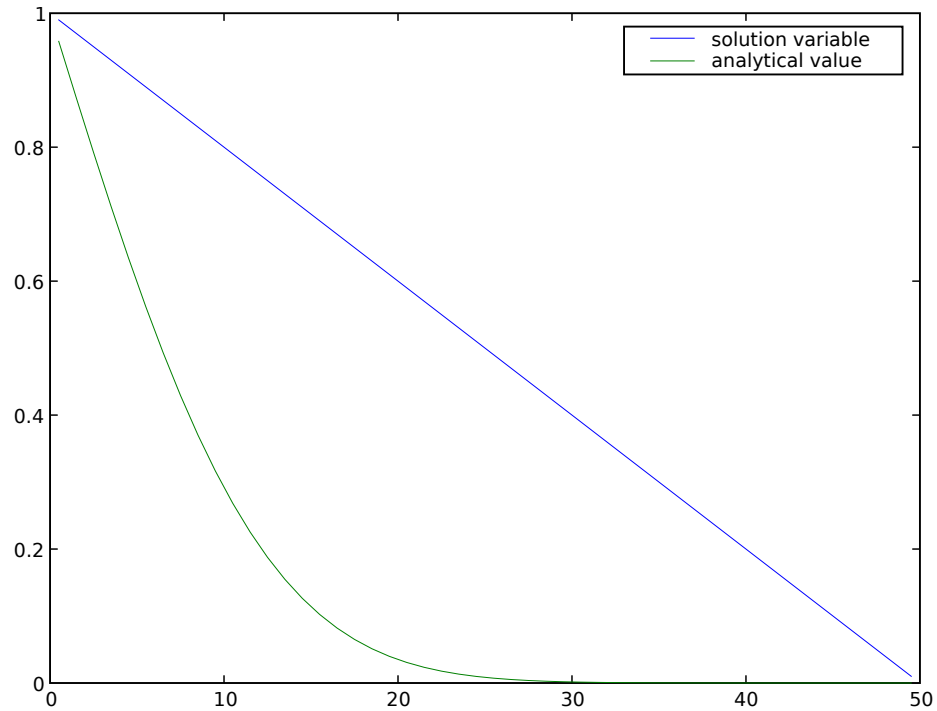
```
>>> DiffusionTerm(coeff=D).solve(var=phi)
```

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```

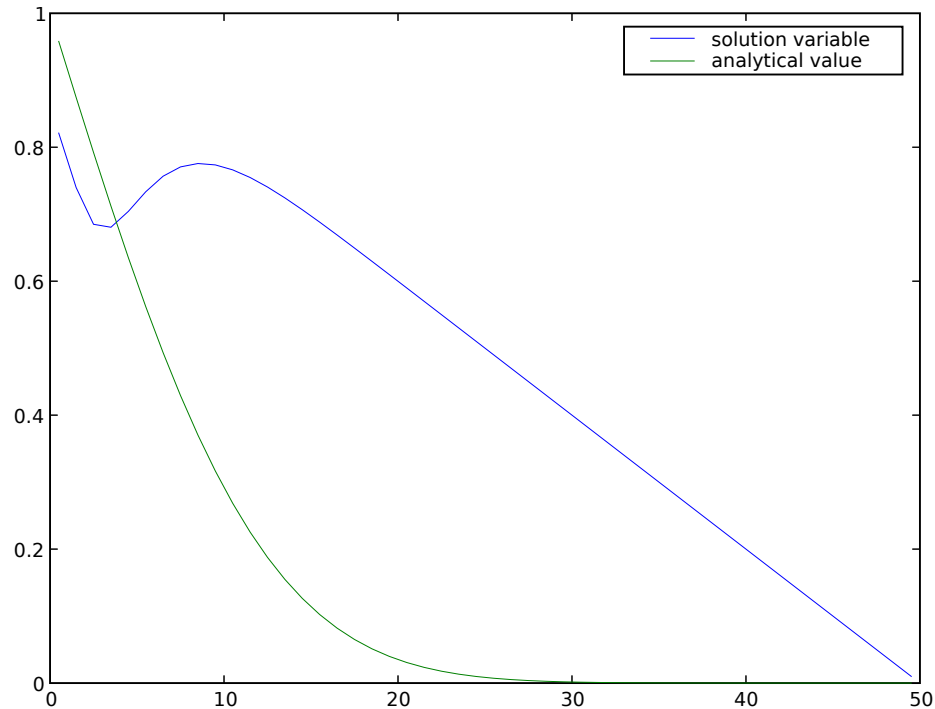
...     eqI.solve(var=phi, dt=dt)
...     if __name__ == '__main__':
...         viewer.plot()

```

```

>>> 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 gradient of

```
>>> gradRight = 1.
```

to the right:

```
>>> phi = CellVariable(mesh=mesh, name="solution variable")
>>> phi.faceGrad.constrain([gradRight], 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

```
>>> eq = DiffusionTerm(coeff=D)
>>> solver = eq.getDefaultSolver(tolerance=1e-7)
>>> eq.solve(var=phi, solver=solver)
```

The analytical solution is simply

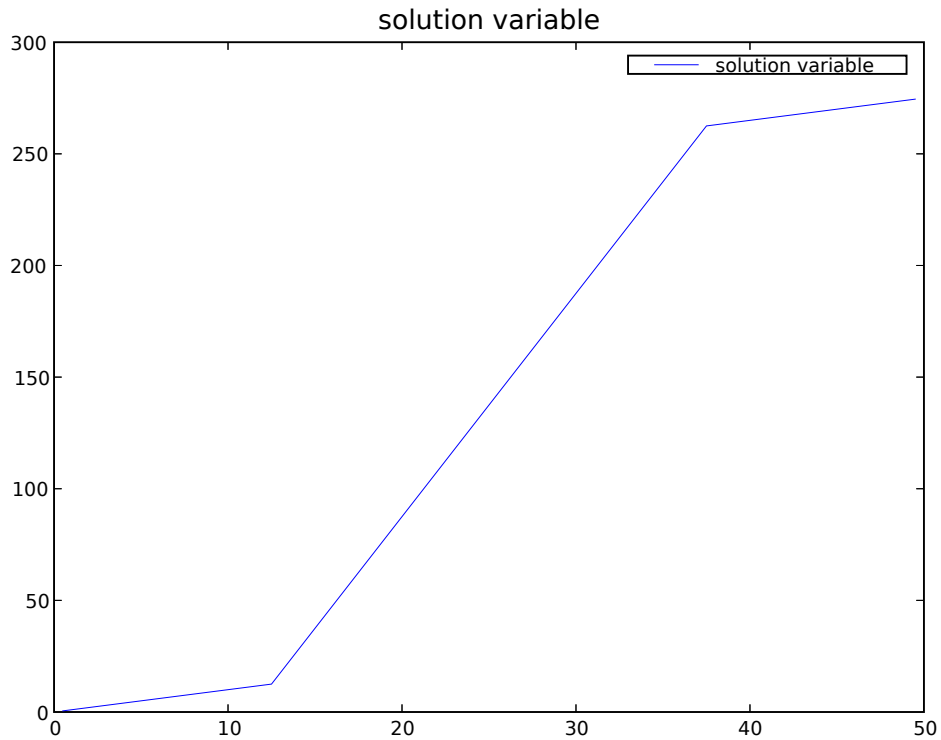
$$\phi = \begin{cases} x & \text{for } 0 < x < L/4, \\ 10x - 9L/4 & \text{for } L/4 \leq x < 3L/4, \\ x + 18L/4 & \text{for } 3L/4 \leq x < L, \end{cases}$$

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. (24.1), 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([gradRight], mesh.facesRight)
>>> phiF.faceGrad.constrain([gradRight], 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)
>>> solver = eqT.getDefaultSolver(tolerance=1e-7)
>>> eqT.solve(var=phiT, solver=solver)

```

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

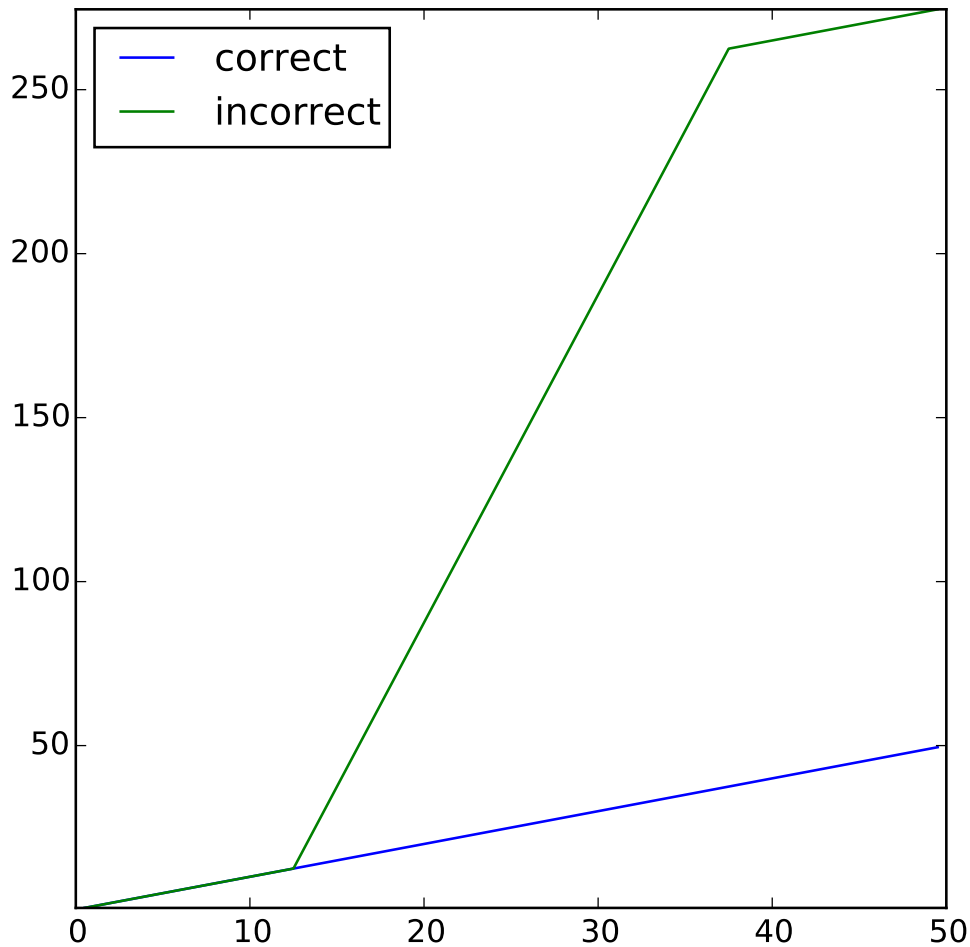
```

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

$$\frac{\partial \phi}{\partial t} = \nabla \cdot [D(\phi) \nabla \phi]. \quad (24.2)$$

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. (24.2) 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
>>> for sweeps in range(1, 5):
...     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]))
>>> solver = eq.getDefaultSolver(tolerance=1e-6)

```

```

>>> phi[0].setValue(valueRight)
>>> res = 1e+10
>>> while res > 1e-6:
...     res = eq.sweep(var=phi[0],
...                     dt=timeStepDuration,
...                     solver=solver)

```

```

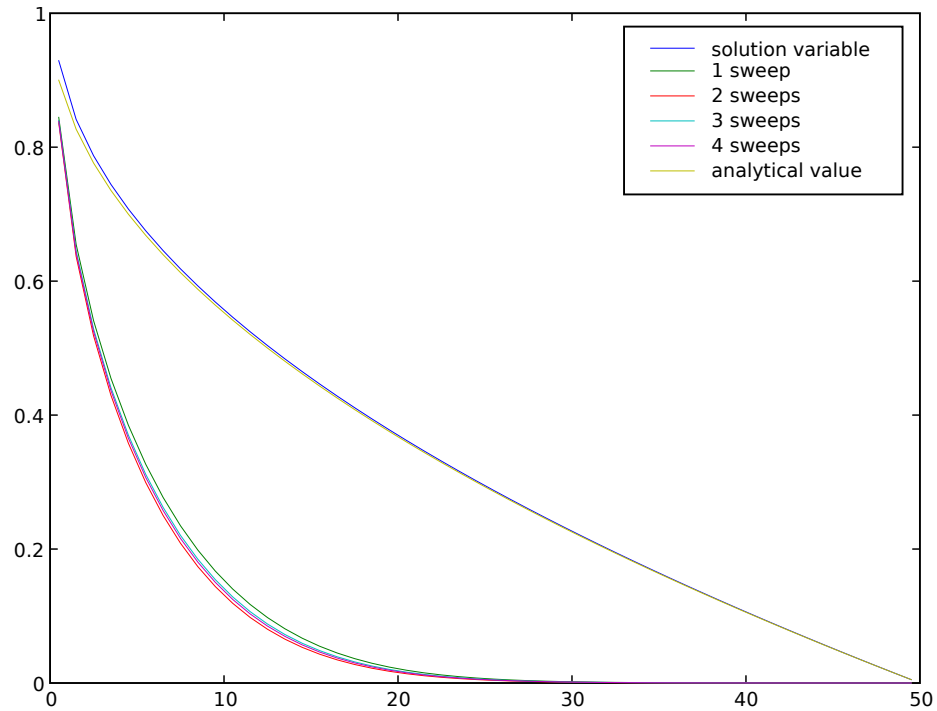
>>> print(phi[0].allclose(phiAnalytical, atol = 1e-1))
1

```

```

>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Implicit variable diffusivity - steady-state. \
... Press <return> to proceed...")

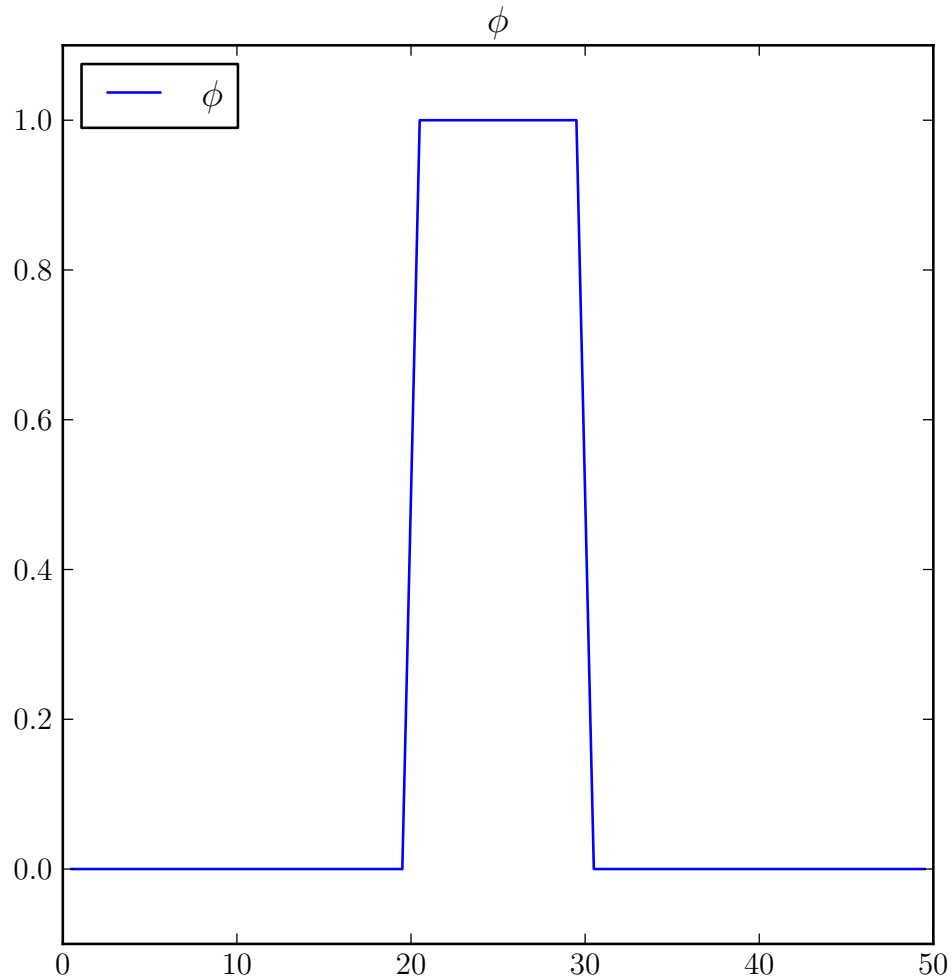
```

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)
```

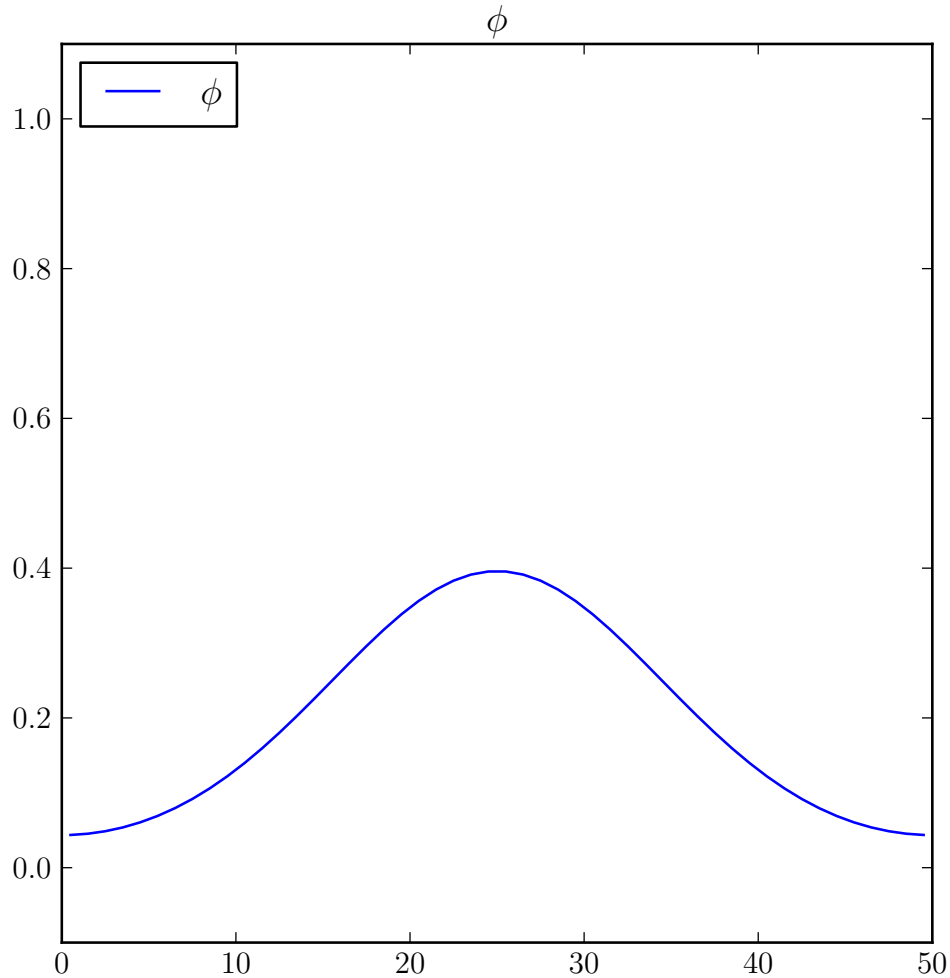
```
>>> dt = 10. * dx**2 / (2 * D)
>>> steps = 200
```

```
>>> 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=4e-5))
True
```

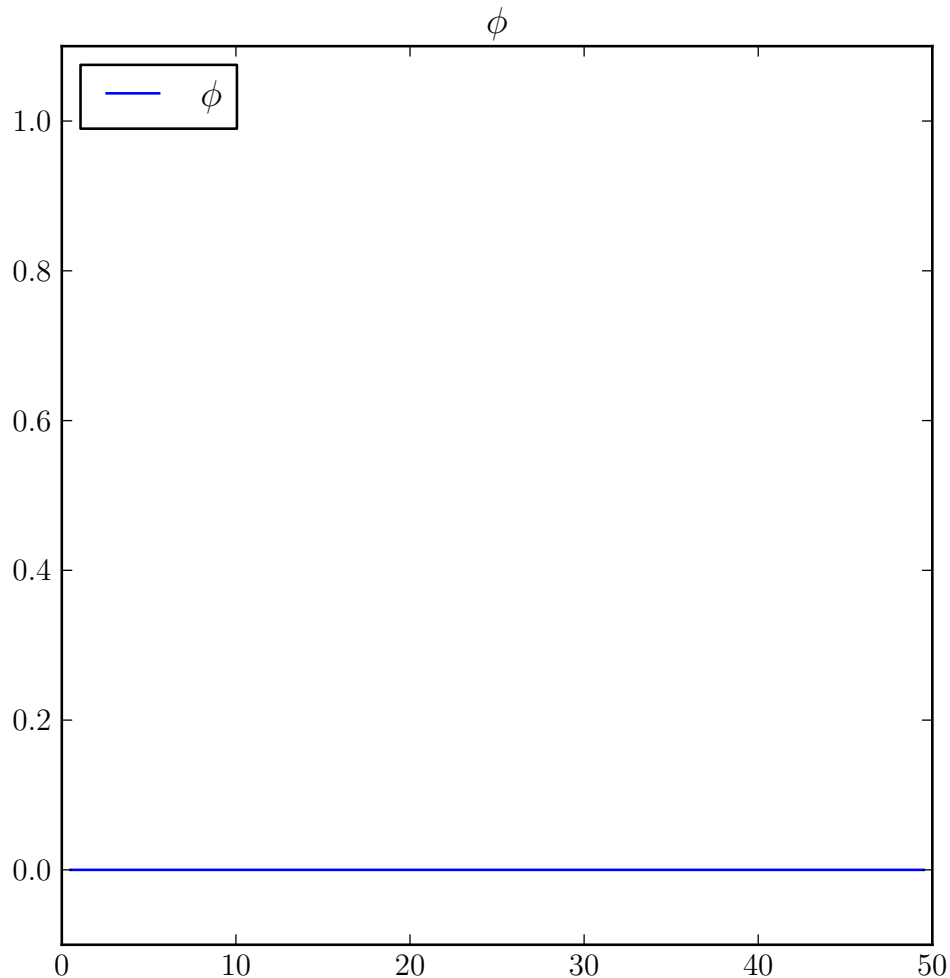
If we reset the initial condition

```
>>> 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

```
>>> try:
...     DiffusionTerm(coeff=D).solve(var=phi)
... except:
...     pass
>>> 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, 0.2, atol=1e-5))
False
```



Depending on the solver, we find that the value may be uniformly zero, infinity, or NaN, or the solver may just fail! What happened to our no-flux boundary conditions? Trilinos actually manages to get the correct solution, but this should not be relied on; this problem has an infinite number of solutions.

The problem is that in the implicit discretization of $\nabla \cdot (D \nabla \phi) = 0$,

$$\begin{pmatrix}
 \frac{D}{\Delta x^2} & -\frac{D}{\Delta x^2} & & & \\
 & \ddots & \ddots & \ddots & \\
 & & -\frac{D}{\Delta x^2} & \frac{2D}{\Delta x^2} & -\frac{D}{\Delta x^2} \\
 & & & -\frac{D}{\Delta x^2} & \frac{2D}{\Delta x^2} & -\frac{D}{\Delta x^2} \\
 & & & & -\frac{D}{\Delta x^2} & \frac{2D}{\Delta x^2} & -\frac{D}{\Delta x^2} \\
 & & & & & \ddots & \ddots & \ddots \\
 & & & & & & -\frac{D}{\Delta x^2} & \frac{D}{\Delta x^2}
 \end{pmatrix}
 \begin{pmatrix}
 \phi_0^{\text{new}} \\
 \vdots \\
 \phi_{j-1}^{\text{new}} \\
 \phi_j^{\text{new}} \\
 \phi_{j+1}^{\text{new}} \\
 \vdots \\
 \phi_{N-1}^{\text{new}}
 \end{pmatrix}
 =
 \begin{pmatrix}
 0 \\
 \vdots \\
 0 \\
 0 \\
 0 \\
 \vdots \\
 0
 \end{pmatrix}$$

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

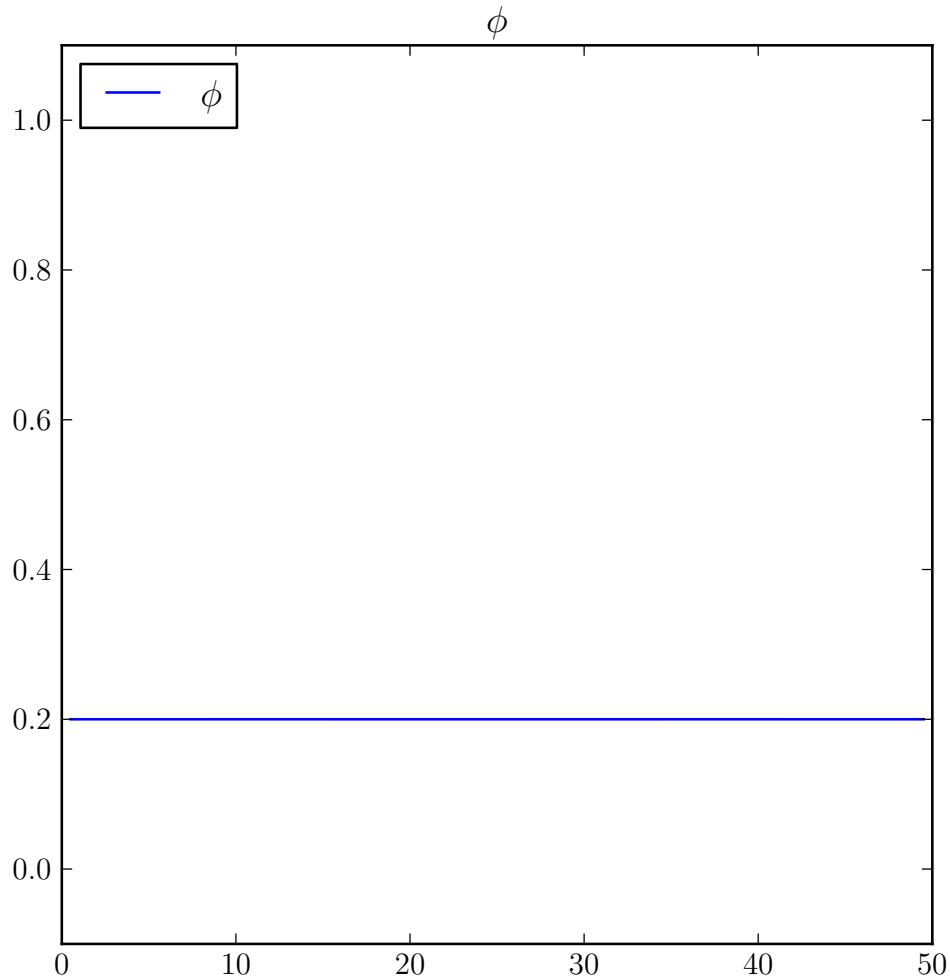
```
>>> phi.value = 0.
>>> phi.setValue(1., where=(x > L/2. - L/10.) & (x < L/2. + L/10.))
>>> if __name__ == '__main__':
...     viewer.plot()
```

```
>>> eq = (TransientTerm() == DiffusionTerm(D))
```

The initial residual is much larger than the norm of the right-hand-side vector, so we use “*initial*” tolerance scaling.

```
>>> solver = eq.getDefaultSolver(criterion="initial", tolerance=1e-9)
>>> eq.solve(var=phi, dt=1e6*dt, solver=solver)
>>> 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
```

(continues on next page)

(continued from previous page)

```

while res > 1e-6:
    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:

```
$ fipy_copy_script .../example.py myExample.py
```

at the command line.

Most of the *FiPy* examples will be a mixture of plain scripts and doctest documentation/tests.

24.5.8 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)))
```

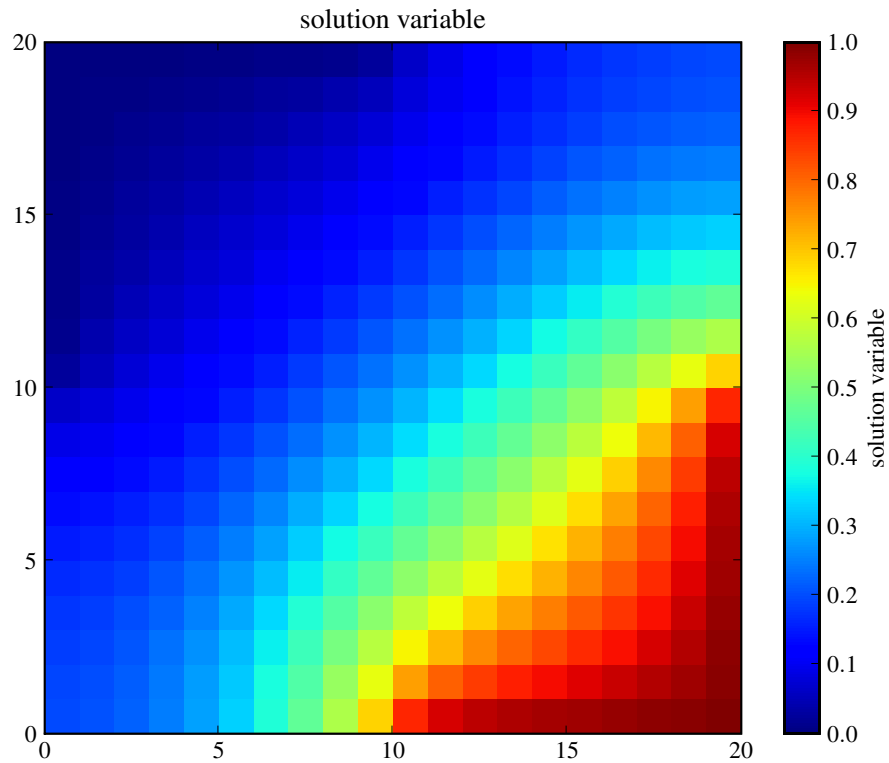
```
>>> phi.constrain(valueTopLeft, facesTopLeft)
>>> phi.constrain(valueBottomRight, facesBottomRight)
```

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

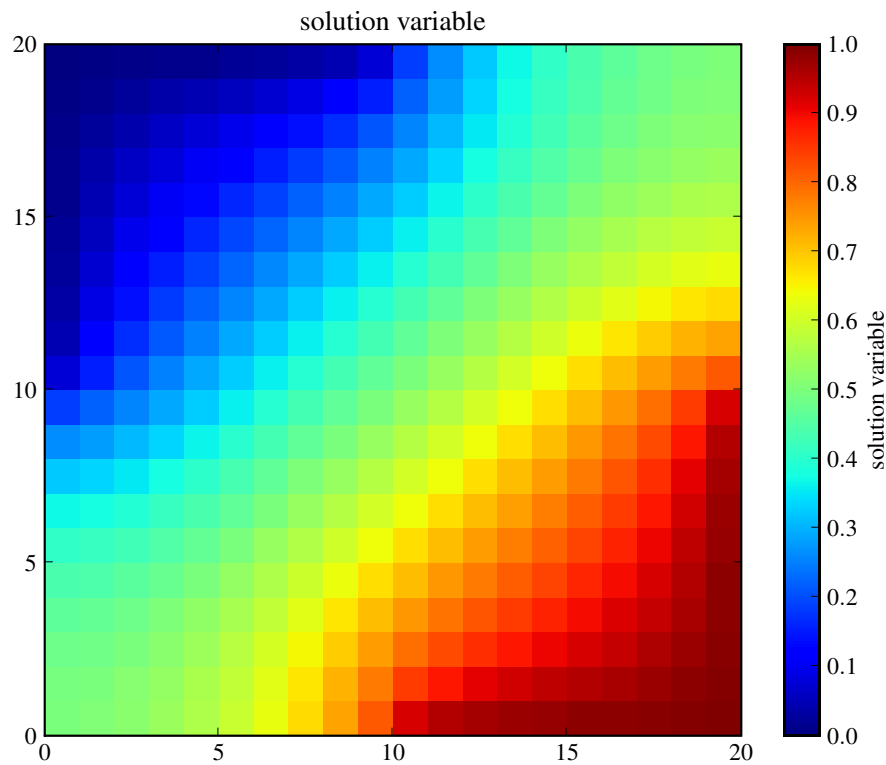
We can test the value of the bottom-right corner cell.

```
>>> 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...")
```

24.5.9 examples.diffusion.mesh20x20Coupled

Solve a coupled set of diffusion equations in two dimensions.

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(var=phi) == DiffusionTerm(coeff=D, var=phi)
```

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)))
```

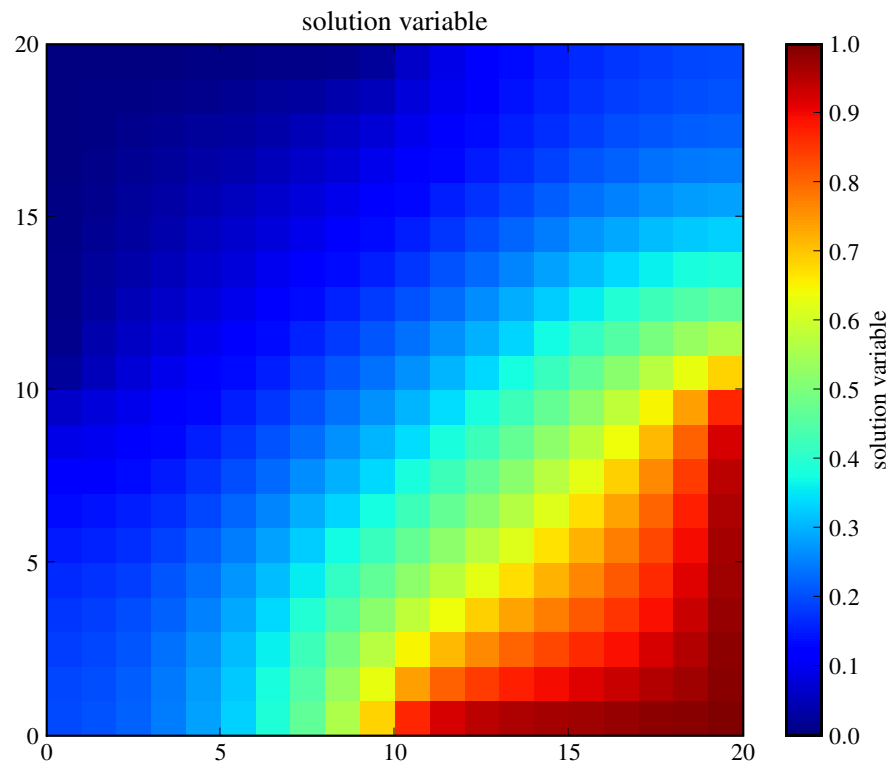
```
>>> phi.constrain(valueTopLeft, facesTopLeft)
>>> phi.constrain(valueBottomRight, facesBottomRight)
```

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
>>> for step in range(steps):
...     eq.solve(dt=timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
```



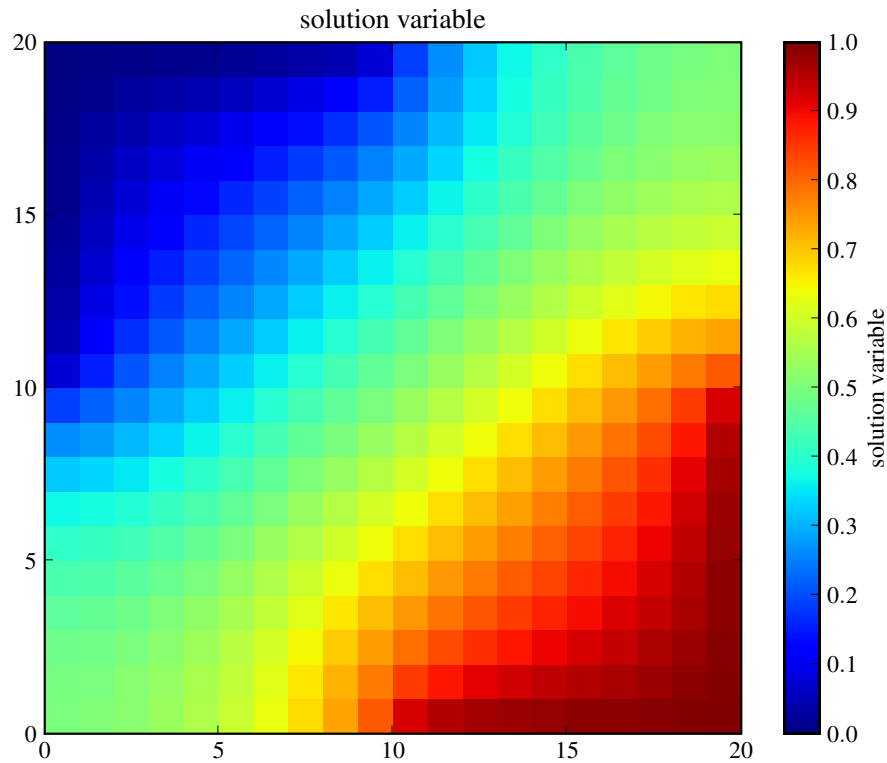
We can test the value of the bottom-right corner cell.

```
>>> 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(var=phi).solve()
>>> 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...")
```

24.5.10 examples.diffusion.nthOrder

Modules

`examples.diffusion.nthOrder.
input4thOrder1D`

Solve a fourth-order diffusion problem.

`examples.diffusion.nthOrder.
input4thOrder_line`

```
>>> eq.solve(var,
```

`examples.diffusion.nthOrder.test`

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, LinearLUSolver
```

```
>>> 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
```

```
>>> solver = LinearLUSolver()
```

We perform one implicit timestep to achieve steady state

```
>>> eq.solve(var=var,
...          boundaryConditions=BCs,
...          solver=solver)
```

The analytical solution is:

$$\phi = \frac{\alpha_4}{6}x^3 + \frac{\alpha_3}{2}x^2 + \left(\alpha_2 - \frac{\alpha_4}{2}L^2 - \alpha_3L\right)x + \alpha_1$$

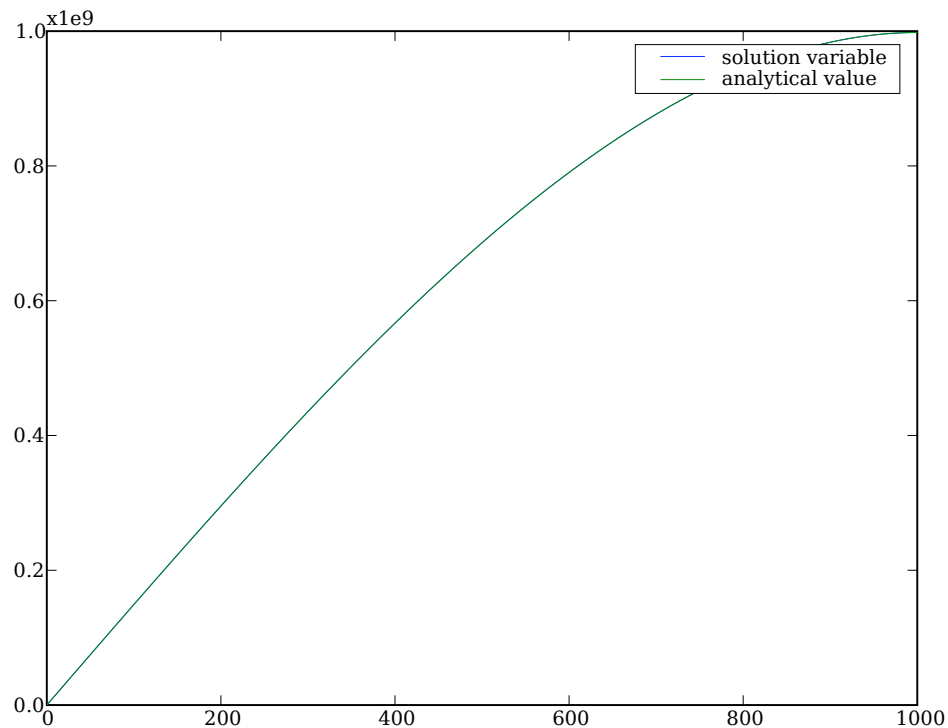
or

```
>>> 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()
```



examples.diffusion.nthOrder.input4thOrder_line

```
>>> eq.solve(var,  
...         boundaryConditions = BCs,  
...         solver = solver)
```

Using the some solvers, the answer is totally inaccurate. This is due to the 4th order term having a high matrix condition number. In this particular example, multigrid preconditioners such as those provided by Trilinos allow a more accurate solution.

```
>>> print(var.allclose(mesh.cellCenters[0], atol = 10))  
1
```

examples.diffusion.nthOrder.test**24.5.11 examples.diffusion.steadyState****Modules**

```
examples.diffusion.steadyState.mesh1D  
examples.diffusion.steadyState.mesh20x20  
examples.diffusion.steadyState.mesh50x50  
examples.diffusion.steadyState.otherMeshes  
examples.diffusion.steadyState.test
```

examples.diffusion.steadyState.mesh1D**Modules**

<code>examples.diffusion.steadyState.mesh1D. inputPeriodic</code>	One can then solve the same problem as in <i>examples/diffusion/steadyState/mesh1D/input.py</i> but with a periodic mesh and no boundary conditions.
<code>examples.diffusion.steadyState.mesh1D. tri2Dinput</code>	To run this example from the base FiPy directory type.

examples.diffusion.steadyState.mesh1D.inputPeriodic

One can then solve the same problem as in *examples/diffusion/steadyState/mesh1D/input.py* but with a periodic mesh and no boundary conditions. The periodic mesh is used to simulate periodic boundary conditions.

```
>>> from fipy import PeriodicGrid1D, CellVariable, TransientTerm, DiffusionTerm, Viewer
```

```
>>> nx = 50
>>> dx = 1.
>>> mesh = PeriodicGrid1D(nx = nx, dx = dx)
```

The variable is initially a line varying from *valueLeft* to *valueRight*.

```
>>> valueLeft = 0
>>> valueRight = 1
>>> x = mesh.cellCenters[0]
```

```
>>> Lx = nx * dx
>>> initialArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> var = CellVariable(name = "solution variable", mesh = mesh,
...                      value = initialArray)
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=0., datamax=1.)
...     viewer.plot()
...     input("press key to continue")
```

A *TransientTerm* is used to provide some fixed point, otherwise the solver has no fixed value and can become unstable.

```
>>> eq = TransientTerm(coeff=1e-8) - DiffusionTerm()
```

The initial residual is much larger than the norm of the right-hand-side vector, so we use “*initial*” tolerance scaling with a tolerance that will drive to an accurate solution.

```
>>> solver = eq.getDefaultSolver(criterion="initial", tolerance=1e-12)
>>> eq.solve(var=var, dt=1., solver=solver)
```

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

The result of the calculation will be the average value over the domain.

```
>>> print(var.allclose((valueLeft + valueRight) / 2., rtol = 1e-5))
True
```

examples.diffusion.steadyState.mesh1D.tri2Dinput

To run this example from the base FiPy directory type:

```
$ python examples/diffusion/steadyState/mesh1D/tri2Dinput.py
```

at the command line. A contour plot should appear and the word *finished* in the terminal.

This example is similar to the example found in [examples.diffusion.mesh1D](#), however, the *mesh* is a [fipy.meshes.tri2D.Tri2D](#) object rather than a [Grid1D\(\)](#) object.

Here, one time step is executed to implicitly find the steady state solution. We refine the solver tolerance from the default 10^{-5} in order to achieve a good solution.

```
>>> eq = DiffusionTerm()
>>> solver = eq.getDefaultSolver(tolerance=1e-8)
>>> eq.solve(var, solver=solver)
```

To test the solution, the analytical result is required. The x coordinates from the mesh are gathered and the length of the domain, Lx , is calculated. An array, *analyticalArray*, is calculated to compare with the numerical result,

```
>>> x = mesh.cellCenters[0]
>>> Lx = nx * dx
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
```

Finally the analytical and numerical results are compared with a tolerance of $1e-10$.

```
>>> print(var.allclose(analyticalArray))
True
```

examples.diffusion.steadyState.mesh20x20**Modules**

examples.diffusion.steadyState.mesh20x20.gmshinput	This input file again solves a 1D diffusion problem as in ./examples/diffusion/steadyState/mesh1D/input.py .
examples.diffusion.steadyState.mesh20x20.isotropy	This input file solves a steady-state 1D diffusion problem as in ./examples/diffusion/mesh1D.py .
examples.diffusion.steadyState.mesh20x20.modifiedMeshInput	This input file again solves a 1D diffusion problem as in ./examples/diffusion/steadyState/mesh1D/input.py .
examples.diffusion.steadyState.mesh20x20.orthoerror	This test file generates lots of different <i>SkewedGrid2D</i> meshes, each with a different non-orthogonality, and runs a 1D diffusion problem on them all.
examples.diffusion.steadyState.mesh20x20.tri2Dinput	This input file again solves a 2D diffusion problem on a triangular mesh.

examples.diffusion.steadyState.mesh20x20.gmshinput

This input file again solves a 1D diffusion problem as in `./examples/diffusion/steadyState/mesh1D/input.py`. In order to test the non-orthogonality error, this uses a *SkewedGrid2D*, which is a *Grid2D* with each interior vertex moved in a random direction.

examples.diffusion.steadyState.mesh20x20.isotropy

This input file solves a steady-state 1D diffusion problem as in `./examples/diffusion/mesh1D.py`. The difference being that it uses a tensor for the diffusion coefficient, even though the coefficient is isotropic.

```
>>> from fipy import Grid2D, CellVariable, DiffusionTerm, Viewer
```

```
>>> Lx = 20
>>> mesh = Grid2D(nx=20, ny=20)
>>> x, y = mesh.cellCenters
```

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

```
>>> var = CellVariable(name = "solution variable",
...                     mesh = mesh,
...                     value = valueLeft)
```

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

```
>>> DiffusionTerm(coeff=((1., 0.),
...                      (0., 1.)),).solve(var)
```

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

```
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> print(var.allclose(analyticalArray, atol = 0.025))
1
```

examples.diffusion.steadyState.mesh20x20.modifiedMeshInput

This input file again solves a 1D diffusion problem as in `./examples/diffusion/steadyState/mesh1D/input.py`. The difference being that it uses a triangular mesh loaded in using the *Gmsh2D* object.

The result is again tested in the same way:

```
>>> from fipy import Gmsh2D, CellVariable, DiffusionTerm, Viewer
>>> from fipy.tools import numerix
```

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

```

>>> Lx = 20
>>> mesh = Gmsh2D('''
...     cellSize = 0.5;
...     Point(2) = {0, 0, 0, cellSize};
...     Point(3) = {% (Lx)g, 0, 0, cellSize};
...     Point(4) = {% (Lx)g, % (Lx)g, 0, cellSize};
...     Point(5) = {0, % (Lx)g, 0, cellSize};
...
...     Line(6) = {2, 3};
...     Line(7) = {3, 4};
...     Line(8) = {4, 5};
...     Line(9) = {5, 2};
...
...     Line Loop(10) = {6, 7, 8, 9};
...
...     Plane Surface(11) = {10};
...     ''' % locals())

```

```

>>> var = CellVariable(name = "solution variable",
...                     mesh = mesh,
...                     value = valueLeft)

```

```

>>> var.constrain(valueLeft, mesh.facesLeft)
>>> var.constrain(valueRight, mesh.facesRight)

```

```

>>> DiffusionTerm().solve(var)

```

```

>>> from fipy import input
>>> x = mesh.cellCenters[0]
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> print(var.allclose(analyticalArray, atol=0.025))
True

```

```

>>> errorVar = abs(var - analyticalArray)
>>> errorVar.name = "absolute error"

```

```

>>> NonOrthoVar = CellVariable(name="non-orthogonality",
...                             mesh=mesh,
...                             value=mesh._nonOrthogonality)
>>> print(max(NonOrthoVar) < 0.51)
True

```

```

>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var)
...     viewer.plot()
...
...     errorViewer = Viewer(vars=errorVar)
...     errorViewer.plot()
...
...     NOViewer = Viewer(vars=NonOrthoVar)
...     NOViewer.plot()

```

(continues on next page)

(continued from previous page)

```
...
... input("finished")
```

examples.diffusion.steadyState.mesh20x20.orthoerror

This test file generates lots of different *SkewedGrid2D* meshes, each with a different non-orthogonality, and runs a 1D diffusion problem on them all. It computes the RMS non-orthogonality and the RMS error for each mesh and displays them in a graph, allowing the relationship of error to non-orthogonality to be investigated.

examples.diffusion.steadyState.mesh20x20.tri2Dinput

This input file again solves a 2D diffusion problem on a triangular mesh. We refine the solver tolerance from the default 10^{-5} in order to achieve a good solution.

```
>>> eq = DiffusionTerm()
>>> solver = eq.getDefaultSolver(tolerance=1e-10)
>>> eq.solve(var, solver=solver)
```

The result is again tested in the same way:

```
>>> Lx = nx * dx
>>> x = mesh.cellCenters[0]
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> print(var.allclose(analyticalArray, rtol = 1e-8))
True
```

examples.diffusion.steadyState.mesh50x50

Modules

```
examples.diffusion.steadyState.mesh50x50.
input
examples.diffusion.steadyState.mesh50x50.
tri2Dinput
```

This input file again solves a 1D diffusion problem as in [examples.diffusion.mesh1D](#).

This input file again solves a 1D diffusion problem as in [./examples/diffusion/steadyState/mesh1D/input.py](#).

examples.diffusion.steadyState.mesh50x50.input

This input file again solves a 1D diffusion problem as in [examples.diffusion.mesh1D](#). The difference being that the mesh is two dimensional.

The result is again tested in the same way:

```
>>> eq = DiffusionTerm()
>>> solver = eq.getDefaultSolver(tolerance=1e-10)
>>> eq.solve(var, solver=solver)
>>> Lx = nx * dx
>>> x = mesh.cellCenters[0]
```

(continues on next page)

(continued from previous page)

```
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> print(var.allclose(analyticalArray, rtol = 1e-9))
True
```

examples.diffusion.steadyState.mesh50x50.tri2Dinput

This input file again solves a 1D diffusion problem as in `./examples/diffusion/steadyState/mesh1D/input.py`. The difference being that the mesh size is given by

```
>>> nx = 50
>>> ny = 50
```

The result is again tested in the same way:

```
>>> eq = DiffusionTerm()
>>> solver = eq.getDefaultSolver(tolerance=1e-10)
>>> eq.solve(var, solver=solver)
>>> Lx = nx * dx
>>> x = mesh.cellCenters[0]
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> print(var.allclose(analyticalArray, atol = 1e-7))
True
```

examples.diffusion.steadyState.otherMeshes

Modules

<code>examples.diffusion.steadyState.otherMeshes.cubicalProblem</code>	Test case for the <i>Grid3D</i> .
<code>examples.diffusion.steadyState.otherMeshes.grid3Dinput</code>	Test case for the <i>Grid3D</i> .
<code>examples.diffusion.steadyState.otherMeshes.prism</code>	This input file again solves a 1D diffusion problem as in <code>./examples/diffusion/steadyState/mesh1D/input.py</code> .

examples.diffusion.steadyState.otherMeshes.cubicalProblem

Test case for the *Grid3D*. Diffusion problem with boundary conditions: 0 on front, 10 on back, and 5 on all other sides.

examples.diffusion.steadyState.otherMeshes.grid3Dinput

Test case for the *Grid3D*.

```
>>> eq3D = DiffusionTerm()
>>> solver = eq3D.getDefaultSolver(tolerance=1e-6)
>>> eq3D.solve(var, solver=solver)
>>> eq2D = DiffusionTerm()
>>> solver = eq2D.getDefaultSolver(tolerance=1e-6)
>>> eq2D.solve(var2, solver=solver)
>>> a = numerix.array(var.globalValue)
>>> b = numerix.array(var2.globalValue)
>>> c = numerix.ravel(numerix.array((b, b, b)))
>>> print(numerix.allclose(a, c))
True
```

examples.diffusion.steadyState.otherMeshes.prism

This input file again solves a 1D diffusion problem as in *./examples/diffusion/steadyState/mesh1D/input.py*. The difference being that it uses a triangular mesh loaded in using the Gmsh.

The result is again tested in the same way:

```
>>> from fipy import CellVariable, GmshGrid3D, DiffusionTerm
```

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

```
>>> mesh = GmshGrid3D(dx=1, dy=1, dz=1, nx=20, ny=1, nz=1)
```

```
>>> var = CellVariable(name = "solution variable",
...                   mesh = mesh,
...                   value = valueLeft)
```

```
>>> exteriorFaces = mesh.exteriorFaces
>>> xFace = mesh.faceCenters[0]
```

```
>>> var.constrain(valueLeft, exteriorFaces & (xFace ** 2 < 0.0000000000000001))
>>> var.constrain(valueRight, exteriorFaces & ((xFace - 20) ** 2 < 0.0000000000000001))
```

```
>>> DiffusionTerm().solve(var)
>>> Lx = 20
>>> x = mesh.cellCenters[0]
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> print(var.allclose(analyticalArray, atol = 0.027))
1
```

examples.diffusion.steadyState.test

24.5.12 examples.diffusion.test

Run all the test cases in examples/diffusion/

24.5.13 examples.diffusion.variable

This example is a 1D steady state diffusion test case as in `./examples/diffusion/variable/mesh2x1/input.py` with then number of cells set to $nx = 10$.

A simple analytical answer can be used to test the result:

```
>>> DiffusionTerm(coeff = diffCoeff).solve(var)
>>> if __name__ == "__main__":
...     viewer = Viewer(vars = var)
...     viewer.plot()
>>> x = mesh.cellCenters[0]
>>> values = numerix.where(x < 3. * L / 4., 10 * x - 9. * L / 4., x + 18. * L / 4.)
>>> values = numerix.where(x < L / 4., x, values)
>>> print(var.allclose(values))
True
```

24.6 examples.elphf

The following examples exhibit various parts of a model to study electrochemical interfaces. In a pair of papers, Guyer, Boettinger, Warren and McFadden [22] [23] have shown that an electrochemical interface can be modeled by an equation for the phase field ξ

$$\underbrace{\frac{1}{M_\xi} \frac{\partial \xi}{\partial t}}_{\text{transient}} = \underbrace{\kappa_\xi \nabla^2 \xi}_{\text{diffusion}} - \underbrace{\sum_{j=1}^n C_j [p'(\xi) \Delta \mu_j^\circ + g'(\xi) W_j]}_{\text{source}} + \underbrace{\frac{\epsilon'(\xi)}{2} (\nabla \phi)^2}_{\text{dielectric}} \quad (24.3)$$

a set of diffusion equations for the concentrations C_j , for the substitutional elements $j = 2, \dots, n-1$

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}} + \underbrace{D_j \nabla \cdot \frac{C_j}{1 - \sum_{k=2}^{n-1} C_k} \left\{ \underbrace{\sum_{\substack{i=2 \\ i \neq j}}^{n-1} \nabla C_i}_{\text{counter diffusion}} + \underbrace{C_n [p'(\xi) \Delta \mu_{jn}^\circ + g'(\xi) W_{jn}]}_{\text{phase transformation}} \nabla \xi + \underbrace{C_n z_{jn} \nabla \phi}_{\text{electromigration}} \right\}}_{\text{convection}} \quad (24.4)$$

a diffusion equation for the concentration C_{e-} of electrons

$$\begin{aligned} \frac{\partial C_{e-}}{\partial t} &= D_{e-} \nabla^2 C_{e-} \\ + D_{e-} \nabla \cdot C_{e-} &\underbrace{\left\{ \underbrace{[p'(\xi)\Delta\mu_{e-}^\circ + g'(\xi)W_{e-}]}_{\text{phase transformation}} \nabla \xi + \underbrace{z_{e-} \nabla \phi}_{\text{electromigration}} \right\}}_{\text{convection}} \end{aligned} \quad (24.5)$$

and Poisson's equation for the electrostatic potential ϕ

$$\underbrace{\nabla \cdot (\epsilon \nabla \phi)}_{\text{diffusion}} + \underbrace{\sum_{j=1}^n z_j C_j}_{\text{source}} = 0 \quad (24.6)$$

M_ξ is the phase field mobility, κ_ξ is the phase field gradient energy coefficient, $p'(\xi) = 30\xi^2(1-\xi)^2$, and $g'(\xi) = 2\xi(1-\xi)(1-2\xi)$. For a given species j , $\Delta\mu_j^\circ$ is the standard chemical potential difference between the electrode and electrolyte for a pure material, W_j is the magnitude of the energy barrier in the double-well free energy function, z_j is the valence, and D_j is the self diffusivity. $\Delta\mu_{jn}^\circ$, W_{jn} , and z_{jn} are the differences of the respective quantities $\Delta\mu_j^\circ$, W_j , and z_j between substitutional species j and the solvent species n . The total charge is denoted by $\sum_{j=1}^n z_j C_j$.

Although unresolved stiffnesses make the full solution of this coupled set of equations intractable in *FiPy*, the following examples demonstrate the setup and solution of various parts.

Modules

<code>examples.elphf.diffusion</code>	
<code>examples.elphf.input</code>	This example adds two more components to <code>examples/elphf/input1DphaseBinary.py</code> one of which is another substitutional species and the other represents electrons and diffuses interstitially.
<code>examples.elphf.phase</code>	A simple 1D example to test the setup of the phase field equation.
<code>examples.elphf.phaseDiffusion</code>	This example combines a phase field problem, as given in <code>examples.elphf.phase</code> , with a binary diffusion problem, such as described in the ternary example <code>examples.elphf.diffusion.mesh1D</code> , on a 1D mesh
<code>examples.elphf.poisson</code>	A simple 1D example to test the setup of the Poisson equation.
<code>examples.elphf.test</code>	

24.6.1 examples.elphf.diffusion

Modules

<code>examples.elphf.diffusion.mesh1D</code>	A simple 1D example to test the setup of the multicomponent diffusion equations.
<code>examples.elphf.diffusion.mesh1Ddimensional</code>	In this example, we present the same three-component diffusion problem introduced in <code>examples/elphf/diffusion/mesh1D.py</code> but we demonstrate FiPy's facility to use dimensional quantities.
<code>examples.elphf.diffusion.mesh2D</code>	The same three-component diffusion problem as introduced in <code>examples.elphf.diffusion.mesh1D</code> but in 2D:

examples.elphf.diffusion.mesh1D

A simple 1D example to test the setup of the multicomponent diffusion equations. The diffusion equation for each species in single-phase multicomponent system can be expressed as

$$\frac{\partial C_j}{\partial t} = D_{jj} \nabla^2 C_j + D_j \nabla \cdot \frac{C_j}{1 - \sum_{\substack{k=2 \\ k \neq j}}^{n-1} C_k} \sum_{\substack{i=2 \\ i \neq j}}^{n-1} \nabla C_i$$

where C_j is the concentration of the j^{th} species, t is time, D_{jj} is the self-diffusion coefficient of the j^{th} species, and $\sum_{\substack{i=2 \\ i \neq j}}^{n-1}$ represents the summation over all substitutional species in the system, excluding the solvent and the component of interest.

We solve the problem on a 1D mesh

```
>>> nx = 400
>>> dx = 0.01
>>> L = nx * dx
```

```
>>> from fipy import CellVariable, FaceVariable, Grid1D, TransientTerm, DiffusionTerm, \
↳ PowerLawConvectionTerm, DefaultAsymmetricSolver, Viewer
>>> mesh = Grid1D(dx = dx, nx = nx)
```

One component in this ternary system will be designated the “solvent”

```
>>> class ComponentVariable(CellVariable):
...     def __init__(self, mesh, value = 0., name = '',
...                   standardPotential = 0., barrier = 0.,
...                   diffusivity = None, valence = 0, equation = None):
...         CellVariable.__init__(self, mesh = mesh, value = value,
...                               name = name)
...         self.standardPotential = standardPotential
...         self.barrier = barrier
...         self.diffusivity = diffusivity
...         self.valence = valence
...         self.equation = equation
... 
```

(continues on next page)

(continued from previous page)

```

...     def copy(self):
...         return self.__class__(mesh = self.mesh,
...                                value = self.value,
...                                name = self.name,
...                                standardPotential =
...                                    self.standardPotential,
...                                barrier = self.barrier,
...                                diffusivity = self.diffusivity,
...                                valence = self.valence,
...                                equation = self.equation)

```

```
>>> solvent = ComponentVariable(mesh = mesh, name = 'Cn', value = 1.)
```

We can create an arbitrary number of components, simply by providing a `tuple` or `list` of components

```

>>> substitutionals = [
...     ComponentVariable(mesh = mesh, name = 'C1', diffusivity = 1.,
...                        standardPotential = 1., barrier = 1.),
...     ComponentVariable(mesh = mesh, name = 'C2', diffusivity = 1.,
...                        standardPotential = 1., barrier = 1.),
... ]

```

```
>>> interstitials = []
```

```

>>> for component in substitutionals:
...     solvent -= component

```

We separate the solution domain into two different concentration regimes

```

>>> x = mesh.cellCenters[0]
>>> substitutionals[0].setValue(0.3)
>>> substitutionals[0].setValue(0.6, where=x > L / 2)
>>> substitutionals[1].setValue(0.6)
>>> substitutionals[1].setValue(0.3, where=x > L / 2)

```

We create one diffusion equation for each substitutional component. The initial residual is much larger than the norm of the right-hand-side vector, so we use “*initial*” tolerance scaling with a tolerance and preconditioner that will drive to an accurate solution.

```
>>> from fipy import solver_suite
```

```

>>> for Cj in substitutionals:
...     CkSum = ComponentVariable(mesh = mesh, value = 0.)
...     CkFaceSum = FaceVariable(mesh = mesh, value = 0.)
...     for Ck in [Ck for Ck in substitutionals if Ck is not Cj]:
...         CkSum += Ck
...         CkFaceSum += Ck.harmonicFaceValue
...
...     convectionCoeff = CkSum.faceGrad \
...         * (Cj.diffusivity / (1. - CkFaceSum))

```

(continues on next page)

(continued from previous page)

```

...     Cj.equation = (TransientTerm()
...                     == DiffusionTerm(coeff=Cj.diffusivity)
...                     + PowerLawConvectionTerm(coeff=convectionCoeff))
...     if solver_suite in ["trilinos"]:
...         from fipy import MultilevelNSSAPreconditioner
...         preconditioner = MultilevelNSSAPreconditioner()
...     else:
...         preconditioner = "default"
...     Cj.solver = DefaultAsymmetricSolver(criterion="initial",
...                                         precon=preconditioner,
...                                         iterations=3200,
...                                         tolerance=1e-13)

```

If we are running interactively, we create a viewer to see the results

```

>>> if __name__ == '__main__':
...     viewer = Viewer(vars=[solvent] + substitutionals,
...                       datamin=0, datamax=1)
...     viewer.plot()

```

Now, we iterate the problem to equilibrium, plotting as we go

```

>>> for i in range(40):
...     for Cj in substitutionals:
...         Cj.equation.solve(var=Cj,
...                             dt=100000.,
...                             solver=Cj.solver)
...     if __name__ == '__main__':
...         viewer.plot()

```

Since there is nothing to maintain the concentration separation in this problem, we verify that the concentrations have become uniform

Note: Between *petsc=3.13.2=h82b89f7_0* and *petsc=3.13.4=h82b89f7_0*, PETSc ceased achieving 1e-7 tolerance when solving on 2 processors on Linux. Solving on macOS is OK. Solving on 1, 3, or 4 processors is OK.

```

>>> print(substitutionals[0].allclose(0.45, rtol = 2e-7, atol = 2e-7))
True
>>> print(substitutionals[1].allclose(0.45, rtol = 2e-7, atol = 2e-7))
True

```

examples.elphf.diffusion.mesh1Ddimensional

In this example, we present the same three-component diffusion problem introduced in `examples/elphf/diffusion/mesh1D.py` but we demonstrate FiPy's facility to use dimensional quantities.

```

>>> import warnings
>>> warnings.warn("\n\n\tSupport for physical dimensions is incomplete.\n\tIt is not
↳possible to solve dimensional equations.\n")

```

```
>>> from fipy import CellVariable, FaceVariable, PhysicalField, Grid1D, TransientTerm, \
↳ DiffusionTerm, PowerLawConvectionTerm, LinearLUSolver, Viewer
>>> from fipy.tools import numerix
```

We solve the problem on a 40 mm long 1D mesh

```
>>> nx = 40
>>> dx = PhysicalField(1., "mm")
>>> L = nx * dx
>>> mesh = Grid1D(dx = dx, nx = nx)
```

Again, one component in this ternary system will be designated the “solvent”

```
>>> class ComponentVariable(CellVariable):
...     def __init__(self, mesh, value = 0., name = '',
...                   standardPotential = 0., barrier = 0.,
...                   diffusivity = None, valence = 0, equation = None):
...         CellVariable.__init__(self, mesh = mesh, value = value,
...                                name = name)
...         self.standardPotential = Variable(standardPotential)
...         self.barrier = Variable(barrier)
...         self.diffusivity = Variable(diffusivity)
...         self.valence = valence
...         self.equation = equation
...
...     def copy(self):
...         return self.__class__(mesh = self.mesh,
...                                value = self.value,
...                                name = self.name,
...                                standardPotential =
...                                    self.standardPotential,
...                                barrier = self.barrier,
...                                diffusivity = self.diffusivity,
...                                valence = self.valence,
...                                equation = self.equation)
```

```
>>> solvent = ComponentVariable(mesh = mesh, name = 'Cn', value = "1 mol/m**3")
```

We can create an arbitrary number of components, simply by providing a *Tuple* or *list* of components

```
>>> substitutionals = [
...     ComponentVariable(mesh = mesh, name = 'C1', diffusivity = "1e-9 m**2/s",
...                       standardPotential = 1., barrier = 1., value = "0.3 mol/m**3"),
...     ComponentVariable(mesh = mesh, name = 'C2', diffusivity = "1e-9 m**2/s",
...                       standardPotential = 1., barrier = 1., value = "0.6 mol/m**3"),
... ]
```

```
>>> interstitials = []
```

```
>>> for component in substitutionals:
...     solvent -= component
```

We separate the solution domain into two different concentration regimes

```

>>> x = mesh.cellCenters[0]
>>> substitutionals[0].setValue("0.3 mol/m**3")
>>> substitutionals[0].setValue("0.6 mol/m**3", where=x > L / 2)
>>> substitutionals[1].setValue("0.6 mol/m**3")
>>> substitutionals[1].setValue("0.3 mol/m**3", where=x > L / 2)

```

We create one diffusion equation for each substitutional component

```

>>> for Cj in substitutionals:
...     CkSum = ComponentVariable(mesh = mesh, value = 0.)
...     CkFaceSum = FaceVariable(mesh = mesh, value = 0.)
...     for Ck in [Ck for Ck in substitutionals if Ck is not Cj]:
...         CkSum += Ck
...         CkFaceSum += Ck.harmonicFaceValue
...
...     convectionCoeff = CkSum.faceGrad \
...         * (Cj.diffusivity / (1. - CkFaceSum))
...
...     Cj.equation = (TransientTerm()
...                     == DiffusionTerm(coeff=Cj.diffusivity)
...                     + PowerLawConvectionTerm(coeff = convectionCoeff))

```

If we are running interactively, we create a viewer to see the results

```

>>> if __name__ == '__main__':
...     viewer = Viewer(vars=[solvent] + substitutionals,
...                        datamin=0, datamax=1)
...     viewer.plot()

```

Now, we iterate the problem to equilibrium, plotting as we go

```

>>> solver = LinearLUSolver()

>>> for i in range(40):
...     for Cj in substitutionals:
...         Cj.updateOld()
...     for Cj in substitutionals:
...         Cj.equation.solve(var = Cj,
...                             dt = "1000 s",
...                             solver = solver)
...     if __name__ == '__main__':
...         viewer.plot()

```

Since there is nothing to maintain the concentration separation in this problem, we verify that the concentrations have become uniform

```

>>> print(substitutionals[0].scaled.allclose("0.45 mol/m**3",
...     atol = "1e-7 mol/m**3", rtol = 1e-7))
1
>>> print(substitutionals[1].scaled.allclose("0.45 mol/m**3",
...     atol = "1e-7 mol/m**3", rtol = 1e-7))
1

```

Note: The absolute tolerance *atol* must be in units compatible with the value to be checked, but the relative tolerance *rtol* is dimensionless.

examples.elphf.diffusion.mesh2D

The same three-component diffusion problem as introduced in [examples.elphf.diffusion.mesh1D](#) but in 2D:

```
>>> from fipy import CellVariable, FaceVariable, Grid2D, TransientTerm, \
    DiffusionTerm, PowerLawConvectionTerm, Viewer
```

```
>>> nx = 40
>>> dx = 1.
>>> L = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dx, nx = nx, ny = nx)
```

One component in this ternary system will be designated the “solvent”

```
>>> class ComponentVariable(CellVariable):
...     def __init__(self, mesh, value = 0., name = '', standardPotential = 0.,
...                   barrier = 0., diffusivity = None, valence = 0, equation = None):
...         CellVariable.__init__(self, mesh = mesh, value = value, name = name)
...         self.standardPotential = standardPotential
...         self.barrier = barrier
...         self.diffusivity = diffusivity
...         self.valence = valence
...         self.equation = equation
...
...     def copy(self):
...         return self.__class__(mesh = self.mesh, value = self.value,
...                                name = self.name,
...                                standardPotential = self.standardPotential,
...                                barrier = self.barrier,
...                                diffusivity = self.diffusivity,
...                                valence = self.valence,
...                                equation = self.equation)
```

```
>>> solvent = ComponentVariable(mesh = mesh, name = 'Cn', value = 1.)
```

We can create an arbitrary number of components, simply by providing a *Tuple* or *list* of components

```
>>> substitutionals = [
...     ComponentVariable(mesh = mesh, name = 'C1', diffusivity = 1.,
...                         standardPotential = 1., barrier = 1.),
...     ComponentVariable(mesh = mesh, name = 'C2', diffusivity = 1.,
...                         standardPotential = 1., barrier = 1.),
... ]
```

```
>>> interstitials = []
```

```
>>> for component in substitutionals:
...     solvent -= component
```

Although we are not interested in them for this problem, we create one field to represent the “phase” (1 everywhere)

```
>>> phase = CellVariable(mesh = mesh, name = 'xi', value = 1.)
```

and one field to represent the electrostatic potential (0 everywhere)

```
>>> potential = CellVariable(mesh = mesh, name = 'phi', value = 0.)
```

Although it is constant in this problem, in later problems we will need the following functions of the phase field

```
>>> def pPrime(xi):
...     return 30. * (xi * (1 - xi))**2
```

```
>>> def gPrime(xi):
...     return 2 * xi * (1 - xi) * (1 - 2 * xi)
```

We separate the solution domain into two different concentration regimes

```
>>> x = mesh.cellCenters[0]
>>> substitutionals[0].setValue(0.3)
>>> substitutionals[0].setValue(0.6, where=x > L / 2)
>>> substitutionals[1].setValue(0.6)
>>> substitutionals[1].setValue(0.3, where=x > L / 2)
```

We create one diffusion equation for each substitutional component

```
>>> for Cj in substitutionals:
...     CkSum = ComponentVariable(mesh = mesh, value = 0.)
...     CkFaceSum = FaceVariable(mesh = mesh, value = 0.)
...     for Ck in [Ck for Ck in substitutionals if Ck is not Cj]:
...         CkSum += Ck
...         CkFaceSum += Ck.harmonicFaceValue
...
...     counterDiffusion = CkSum.faceGrad
...     phaseTransformation = \
...         (pPrime(phase.harmonicFaceValue) * Cj.standardPotential \
...          + gPrime(phase.harmonicFaceValue) * Cj.barrier) \
...         * phase.faceGrad
...     electromigration = Cj.valence * potential.faceGrad
...     convectionCoeff = counterDiffusion \
...         + solvent.harmonicFaceValue \
...         * (phaseTransformation + electromigration)
...     convectionCoeff *= (Cj.diffusivity / (1. - CkFaceSum))
...
...     Cj.equation = (TransientTerm()
...                     == DiffusionTerm(coeff=Cj.diffusivity)
...                     + PowerLawConvectionTerm(coeff=convectionCoeff))
```

If we are running interactively, we create a viewer to see the results


```

>>> if __name__ == '__main__':
...     viewers = [Viewer(vars=field, datamin=0, datamax=1)
...                 for field in [solvent] + substitutionals]
...     for viewer in viewers:
...         viewer.plot()
...     steps = 40
...     tol = 1e-7
... else:
...     steps = 20
...     tol = 1e-4

```

Now, we iterate the problem to equilibrium, plotting as we go

```

>>> for i in range(steps):
...     for Cj in substitutionals:
...         Cj.equation.solve(var = Cj,
...                             dt = 10000)
...     if __name__ == '__main__':
...         for viewer in viewers:
...             viewer.plot()

```

Since there is nothing to maintain the concentration separation in this problem, we verify that the concentrations have become uniform

```

>>> substitutionals[0].allclose(0.45, rtol = tol, atol = tol).value
1
>>> substitutionals[1].allclose(0.45, rtol = tol, atol = tol).value
1

```

We now rerun the problem with an initial condition that only has a concentration step in one corner.

```

>>> x, y = mesh.cellCenters
>>> substitutionals[0].setValue(0.3)
>>> substitutionals[0].setValue(0.6, where=(x > L / 2.) & (y > L / 2.))
>>> substitutionals[1].setValue(0.6)
>>> substitutionals[1].setValue(0.3, where=(x > L / 2.) & (y > L / 2.))

```

We iterate the problem to equilibrium again

```

>>> for i in range(steps):
...     for Cj in substitutionals:
...         Cj.equation.solve(var = Cj,
...                             dt = 10000)
...     if __name__ == '__main__':
...         for viewer in viewers:
...             viewer.plot()

```

and verify that the correct uniform concentrations are achieved

```

>>> substitutionals[0].allclose(0.375, rtol = tol, atol = tol).value
1
>>> substitutionals[1].allclose(0.525, rtol = tol, atol = tol).value
1

```

24.6.2 examples.elphf.input

This example adds two more components to `examples/elphf/input1DphaseBinary.py` one of which is another substitutional species and the other represents electrons and diffuses interstitially.

Parameters from *2004/January/21/elphf0214*

We start by defining a 1D mesh

```
>>> from fipy import PhysicalField as PF
```

```
>>> RT = (PF("1 Nav*kB") * PF("298 K"))
>>> molarVolume = PF("1.80000006366754e-05 m**3/mol")
>>> Faraday = PF("1 Nav*e")
```

```
>>> L = PF("3 nm")
>>> nx = 1200
>>> dx = L / nx
>>> # nx = 200
>>> # dx = PF("0.01 nm")
>>> ## dx = PF("0.001 nm") * (1.001 - 1/cosh(arange(-10, 10, .01)))
>>> # L = nx * dx
>>> mesh = Grid1D(dx = dx, nx = nx)
>>> # mesh = Grid1D(dx = dx)
>>> # L = mesh.facesRight[0].center[0] - mesh.facesLeft[0].center[0]
>>> # L = mesh.cellCenters[0,-1] - mesh.cellCenters[0,0]
```

We create the phase field

```
>>> timeStep = PF("1e-12 s")
```

```
>>> phase = CellVariable(mesh = mesh, name = 'xi', value = 1, hasOld = 1)
>>> phase.mobility = PF("1 m**3/J/s") / (molarVolume / (RT * timeStep))
>>> phase.gradientEnergy = PF("3.6e-11 J/m") / (mesh.scale**2 * RT / molarVolume)
```

```
>>> def p(xi):
...     return xi**3 * (6 * xi**2 - 15 * xi + 10.)
```

```
>>> def g(xi):
...     return (xi * (1 - xi))**2
```

```
>>> def pPrime(xi):
...     return 30. * (xi * (1 - xi))**2
```

```
>>> def gPrime(xi):
...     return 4 * xi * (1 - xi) * (0.5 - xi)
```

We create four components

```
>>> class ComponentVariable(CellVariable):
...     def __init__(self, mesh, value = 0., name = '', standardPotential = 0., barrier_
...     ↪ = 0., diffusivity = None, valence = 0, equation = None, hasOld = 1):
...         self.standardPotential = standardPotential
```

(continues on next page)

(continued from previous page)

```

...     self.barrier = barrier
...     self.diffusivity = diffusivity
...     self.valence = valence
...     self.equation = equation
...     CellVariable.__init__(self, mesh = mesh, value = value, name = name, hasOld_
↳= hasOld)
...
...     def copy(self):
...         return self.__class__(mesh = self.mesh, value = self.value,
...                                name = self.name,
...                                standardPotential = self.standardPotential,
...                                barrier = self.barrier,
...                                diffusivity = self.diffusivity,
...                                valence = self.valence,
...                                equation = self.equation,
...                                hasOld = 0)

```

the solvent

```

>>> solvent = ComponentVariable(mesh = mesh, name = 'H2O', value = 1.)
>>> CnStandardPotential = PF("34139.7265625 J/mol") / RT
>>> CnBarrier = PF("3.6e5 J/mol") / RT
>>> CnValence = 0

```

and two solute species

```

>>> substitutionals = [
...     ComponentVariable(mesh = mesh, name = 'S04',
...                        diffusivity = PF("1e-9 m**2/s") / (mesh.scale**2/timeStep),
...                        standardPotential = PF("24276.6640625 J/mol") / RT,
...                        barrier = CnBarrier,
...                        valence = -2,
...                        value = PF("0.000010414586295976 mol/l") * molarVolume),
...     ComponentVariable(mesh = mesh, name = 'Cu',
...                        diffusivity = PF("1e-9 m**2/s") / (mesh.scale**2/timeStep),
...                        standardPotential = PF("-7231.81396484375 J/mol") / RT,
...                        barrier = CnBarrier,
...                        valence = +2,
...                        value = PF("55.5553718417909 mol/l") * molarVolume)]

```

and one interstitial

```

>>> interstitials = [
...     ComponentVariable(mesh = mesh, name = 'e-',
...                        diffusivity = PF("1e-9 m**2/s") / (mesh.scale**2/timeStep),
...                        standardPotential = PF("-33225.9453125 J/mol") / RT,
...                        barrier = 0.,
...                        valence = -1,
...                        value = PF("111.110723815414 mol/l") * molarVolume)]

```

```

>>> for component in substitutionals:
...     solvent -= component

```

(continues on next page)

(continued from previous page)

```
... component.standardPotential -= CnStandardPotential
... component.barrier -= CnBarrier
... component.valence -= CnValence
```

Finally, we create the electrostatic potential field

```
>>> potential = CellVariable(mesh = mesh, name = 'phi', value = 0.)
```

```
>>> permittivity = PF("78.49 eps0") / (Faraday**2 * mesh.scale**2 / (RT * molarVolume))
```

```
>>> permittivity = 1.
>>> permittivityPrime = 0.
```

The thermodynamic parameters are chosen to give a solid phase rich in electrons and the solvent and a liquid phase rich in the two substitutional species

```
>>> solvent.standardPotential = CnStandardPotential
>>> solvent.barrier = CnBarrier
>>> solvent.valence = CnValence
```

Once again, we start with a sharp phase boundary

```
>>> x = mesh.cellCenters[0]
>>> phase.setValue(x < L / 2)
>>> interstitials[0].setValue("0.000111111503177394 mol/l" * molarVolume, where=x > L / 2)
>>> substitutionals[0].setValue("0.249944439430068 mol/l" * molarVolume, where=x > L / 2)
>>> substitutionals[1].setValue("0.249999982581341 mol/l" * molarVolume, where=x > L / 2)
```

We again create the phase equation as in `examples.elphf.phase.input1D`

```
>>> mesh.setScale(1)
```

```
>>> phase.equation = TransientTerm(coeff = 1/phase.mobility) \
... == DiffusionTerm(coeff = phase.gradientEnergy) \
... - (permittivityPrime / 2.) * potential.grad.dot(potential.grad)
```

We linearize the source term in the same way as in `example.phase.simple.input1D`.

```
>>> enthalpy = solvent.standardPotential
>>> barrier = solvent.barrier
>>> for component in substitutionals + interstitials:
...     enthalpy += component * component.standardPotential
...     barrier += component * component.barrier
```

```
>>> mXi = -(30 * phase * (1 - phase) * enthalpy + 4 * (0.5 - phase) * barrier)
>>> dmXidXi = (-60 * (0.5 - phase) * enthalpy + 4 * barrier)
>>> S1 = dmXidXi * phase * (1 - phase) + mXi * (1 - 2 * phase)
>>> S0 = mXi * phase * (1 - phase) - phase * S1
```

```
>>> phase.equation -= S0 + ImplicitSourceTerm(coeff = S1)
```

and we create the diffusion equation for the solute as in `examples.elphf.diffusion.input1D`

```
>>> for Cj in substitutionals:
...     CkSum = ComponentVariable(mesh = mesh, value = 0.)
...     CkFaceSum = FaceVariable(mesh = mesh, value = 0.)
...     for Ck in [Ck for Ck in substitutionals if Ck is not Cj]:
...         CkSum += Ck
...         CkFaceSum += Ck.harmonicFaceValue
...
...     counterDiffusion = CkSum.faceGrad
...     # phaseTransformation = (pPrime(phase.harmonicFaceValue) * Cj.standardPotential
...     #                       + gPrime(phase.harmonicFaceValue) * Cj.barrier) * phase.faceGrad
...     phaseTransformation = (pPrime(phase).harmonicFaceValue * Cj.standardPotential
...                             + gPrime(phase).harmonicFaceValue * Cj.barrier) * phase.faceGrad
...     # phaseTransformation = (p(phase).faceGrad * Cj.standardPotential
...     #                       + g(phase).faceGrad * Cj.barrier)
...     electromigration = Cj.valence * potential.faceGrad
...     convectionCoeff = counterDiffusion + \
...         solvent.harmonicFaceValue * (phaseTransformation + electromigration)
...     convectionCoeff *= (Cj.diffusivity / (1. - CkFaceSum))
...
...     Cj.equation = (TransientTerm()
...                     == DiffusionTerm(coeff=Cj.diffusivity)
...                     + PowerLawConvectionTerm(coeff=convectionCoeff))
```

```
>>> for Cj in interstitials:
...     # phaseTransformation = (pPrime(phase.harmonicFaceValue) * Cj.standardPotential
...     #                       + gPrime(phase.harmonicFaceValue) * Cj.barrier) * phase.faceGrad
...     phaseTransformation = (pPrime(phase).harmonicFaceValue * Cj.standardPotential
...                             + gPrime(phase).harmonicFaceValue * Cj.barrier) * phase.faceGrad
...     # phaseTransformation = (p(phase).faceGrad * Cj.standardPotential
...     #                       + g(phase).faceGrad * Cj.barrier)
...     electromigration = Cj.valence * potential.faceGrad
...     convectionCoeff = Cj.diffusivity * (1 + Cj.harmonicFaceValue) * \
...         (phaseTransformation + electromigration)
...
...     Cj.equation = (TransientTerm()
...                     == DiffusionTerm(coeff=Cj.diffusivity)
...                     + PowerLawConvectionTerm(coeff=convectionCoeff))
```

And Poisson's equation

```
>>> charge = 0.
>>> for Cj in interstitials + substitutionals:
...     charge += Cj * Cj.valence
```

```
>>> potential.equation = DiffusionTerm(coeff = permittivity) + charge == 0
```

If running interactively, we create viewers to display the results

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase, datamin=0, datamax=1)
```

(continues on next page)

(continued from previous page)

```
...     concViewer = Viewer(vars=[solvent] + substitutionals + interstitials, ylog=True)
...     potentialViewer = Viewer(vars = potential)
...     phaseViewer.plot()
...     concViewer.plot()
...     potentialViewer.plot()
...     input("Press a key to continue")
```

Again, this problem does not have an analytical solution, so after iterating to equilibrium

```
>>> solver = LinearLUSolver(tolerance = 1e-3)
```

```
>>> potential.constrain(0., mesh.facesLeft)
```

```
>>> phase.residual = CellVariable(mesh = mesh)
>>> potential.residual = CellVariable(mesh = mesh)
>>> for Cj in substitutionals + interstitials:
...     Cj.residual = CellVariable(mesh = mesh)
>>> residualViewer = Viewer(vars = [phase.residual, potential.residual] + [Cj.residual_
↳ for Cj in substitutionals + interstitials])
```

```
>>> tsv = TSVViewer(vars = [phase, potential] + substitutionals + interstitials)
```

```
>>> dt = substitutionals[0].diffusivity * 100
>>> # dt = 1.
>>> elapsed = 0.
>>> maxError = 1e-1
>>> SAFETY = 0.9
>>> ERRCON = 1.89e-4
>>> desiredTimestep = 1.
>>> thisTimestep = 0.
>>> print("%3s: %20s | %20s | %20s | %20s" % ("i", "elapsed", "this", "next dt",
↳ "residual"))
>>> residual = 0.
>>> for i in range(500): # iterate
...     if thisTimestep == 0.:
...         tsv.plot(filename = "%s.tsv" % str(elapsed * timeStep))
...
...     for field in [phase, potential] + substitutionals + interstitials:
...         field.updateOld()
...
...     while True:
...         for j in range(10): # sweep
...             print(i, j, dt * timeStep, residual)
...             # raw_input()
...             residual = 0.
...
...             phase.equation.solve(var = phase, dt = dt)
...             # print phase.name, phase.equation.residual.max()
...             residual = max(phase.equation.residual.max(), residual)
...             phase.residual[:] = phase.equation.residual
...
... 
```

(continues on next page)

(continued from previous page)

```

...     potential.equation.solve(var = potential, dt = dt)
...     # print potential.name, potential.equation.residual.max()
...     residual = max(potential.equation.residual.max(), residual)
...     potential.residual[:] = potential.equation.residual
...
...     for Cj in substitutionals + interstitials:
...         Cj.equation.solve(var = Cj,
...                             dt = dt,
...                             solver = solver)
...         # print Cj.name, Cj.equation.residual.max()
...         residual = max(Cj.equation.residual.max(), residual)
...         Cj.residual[:] = Cj.equation.residual
...
...     # print
...     # phaseViewer.plot()
...     # concViewer.plot()
...     # potentialViewer.plot()
...     # residualViewer.plot()
...
...     residual /= maxError
...     if residual <= 1.:
...         break # step succeeded
...
...     dt = max(SAFETY * dt * residual**-0.2, 0.1 * dt)
...     if thisTimeStep + dt == thisTimeStep:
...         raise FloatingPointError("step size underflow")
...
...     thisTimeStep += dt
...
...     if residual > ERRCON:
...         dt *= SAFETY * residual**-0.2
...     else:
...         dt *= 5.
...
...     # dt *= (maxError / residual)**0.5
...
...     if thisTimeStep >= desiredTimestep:
...         elapsed += thisTimeStep
...         thisTimeStep = 0.
...     else:
...         dt = min(dt, desiredTimestep - thisTimeStep)
...
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         concViewer.plot()
...         potentialViewer.plot()
...         print("%3d: %20s | %20s | %20s | %g" % (i, str(elapsed * timeStep),
...         ↪str(thisTimeStep * timeStep), str(dt * timeStep), residual))

```

we confirm that the far-field phases have remained separated

```
>>> ends = take(phase, (0, -1))
```

(continues on next page)

(continued from previous page)

```
>>> allclose(ends, (1.0, 0.0), rtol = 1e-5, atol = 1e-5)
1
```

and that the concentration fields has appropriately segregated into into their respective phases

```
>>> ends = take(interstitials[0], (0, -1))
>>> allclose(ends, (0.4, 0.3), rtol = 3e-3, atol = 3e-3)
1
>>> ends = take(substitutionals[0], (0, -1))
>>> allclose(ends, (0.3, 0.4), rtol = 3e-3, atol = 3e-3)
1
>>> ends = take(substitutionals[1], (0, -1))
>>> allclose(ends, (0.1, 0.2), rtol = 3e-3, atol = 3e-3)
1
```

24.6.3 examples.elphf.phase

A simple 1D example to test the setup of the phase field equation.

We rearrange Eq. (24.3) to

$$\frac{1}{M_\xi} \frac{\partial \xi}{\partial t} = \kappa_\xi \nabla^2 \xi + \frac{\epsilon'(\xi)}{2} (\nabla \phi)^2 - [p'(\xi) \Delta \mu_n^\circ + g'(\xi) W_n] - \sum_{j=2}^{n-1} C_j [p'(\xi) \Delta \mu_{jn}^\circ + g'(\xi) W_{jn}] - C_{e-} [p'(\xi) \Delta \mu_{e-}^\circ + g'(\xi) W_{e-}]$$

The single-component phase field governing equation can be represented as

$$\frac{1}{M_\xi} \frac{\partial \xi}{\partial t} = \kappa_\xi \nabla^2 \xi - 2\xi(1 - \xi)(1 - 2\xi)W$$

where ξ is the phase field, t is time, M_ξ is the phase field mobility, κ_ξ is the phase field gradient energy coefficient, and W is the phase field barrier energy.

We solve the problem on a 1D mesh

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

```
>>> nx = 400
>>> dx = 0.01
>>> L = nx * dx
>>> mesh = Grid1D(dx = dx, nx = nx)
```

We create the phase field

```
>>> phase = CellVariable(mesh = mesh, name = 'xi')
>>> phase.mobility = numerix.inf
>>> phase.gradientEnergy = 0.025
```

Although we are not interested in them for this problem, we create one field to represent the “solvent” component (1 everywhere)


```
>>> class ComponentVariable(CellVariable):
...     def copy(self):
...         new = self.__class__(mesh = self.mesh,
...                               name = self.name,
...                               value = self.value)
...         new.standardPotential = self.standardPotential
...         new.barrier = self.barrier
...         return new
```

```
>>> solvent = ComponentVariable(mesh = mesh, name = 'Cn', value = 1.)
>>> solvent.standardPotential = 0.
>>> solvent.barrier = 1.
```

and one field to represent the electrostatic potential (0 everywhere)

```
>>> potential = CellVariable(mesh = mesh, name = 'phi', value = 0.)
>>> permittivityPrime = 0.
```

We'll have no substitutional species and no interstitial species in this first example

```
>>> substitutionals = []
>>> interstitials = []
```

```
>>> for component in substitutionals:
...     solvent -= component
```

```
>>> phase.equation = TransientTerm(coeff = 1/phase.mobility) \
...     == DiffusionTerm(coeff = phase.gradientEnergy) \
...     - (permittivityPrime / 2.) \
...     * potential.grad.dot(potential.grad)
```

```
>>> enthalpy = solvent.standardPotential
>>> barrier = solvent.barrier
>>> for component in substitutionals + interstitials:
...     enthalpy += component * component.standardPotential
...     barrier += component * component.barrier
```

We linearize the source term in the same way as in [examples.phase.simple](#).

```
>>> mXi = -(30 * phase * (1. - phase) * enthalpy \
...         + 4 * (0.5 - phase) * barrier)
>>> dmXidXi = (-60 * (0.5 - phase) * enthalpy + 4 * barrier)
>>> S1 = dmXidXi * phase * (1 - phase) + mXi * (1 - 2 * phase)
>>> S0 = mXi * phase * (1 - phase) - phase * S1
```

```
>>> phase.equation -= S0 + ImplicitSourceTerm(coeff = S1)
```

Note: Adding a [Term](#) to an equation formed with == will add to the left-hand side of the equation and subtracting a [Term](#) will add to the right-hand side of the equation

We separate the phase field into electrode and electrolyte regimes

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=mesh.cellCenters[0] > L / 2)
```

Even though we are solving the steady-state problem ($M_\phi = \infty$) we still must sweep the solution several times to equilibrate

```
>>> for step in range(10):
...     phase.equation.solve(var = phase, dt=1.)
```

Since we have only a single component n , with $\Delta\mu_n^\circ = 0$, and the electrostatic potential is uniform, Eq. (24.3) reduces to

$$\frac{1}{M_\xi} \frac{\partial \xi}{\partial t} = \kappa_\xi \nabla^2 \xi - g'(\xi) W_n$$

which we know from [examples.phase.simple](#) has the analytical solution

$$\xi(x) = \frac{1}{2} \left(1 - \tanh \frac{x - L/2}{2d} \right)$$

with an interfacial thickness $d = \sqrt{\kappa_\xi / 2W_n}$.

We verify that the correct equilibrium solution is attained

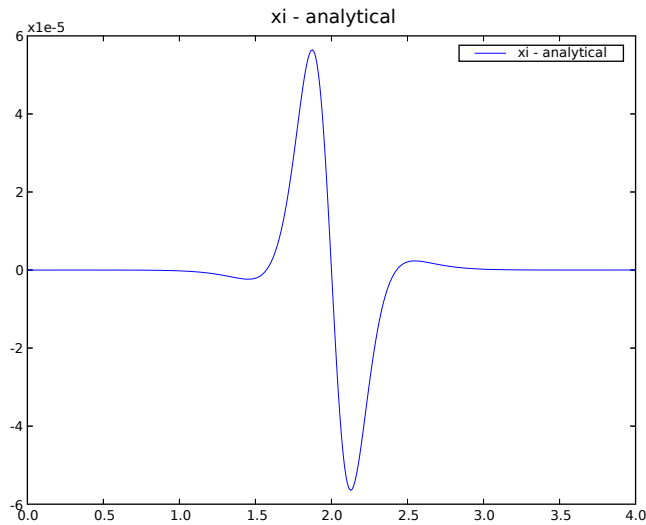
```
>>> x = mesh.cellCenters[0]
```

```
>>> d = numerix.sqrt(phase.gradientEnergy / (2 * solvent.barrier))
>>> analyticalArray = (1. - numerix.tanh((x - L/2.)/(2 * d))) / 2.
```

```
>>> phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4).value
1
```

If we are running interactively, we plot the error

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars = (phase - \
...         CellVariable(name = "analytical", mesh = mesh,
...             value = analyticalArray),))
...     viewer.plot()
```



24.6.4 examples.elphf.phaseDiffusion

This example combines a phase field problem, as given in [examples.elphf.phase](#), with a binary diffusion problem, such as described in the ternary example [examples.elphf.diffusion.mesh1D](#), on a 1D mesh

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

```
>>> nx = 400
>>> dx = 0.01
>>> L = nx * dx
>>> mesh = Grid1D(dx = dx, nx = nx)
```

We create the phase field

```
>>> phase = CellVariable(mesh=mesh, name='xi', value=1., hasOld=1)
>>> phase.mobility = 1.
>>> phase.gradientEnergy = 0.025
```

```
>>> def pPrime(xi):
...     return 30. * (xi * (1 - xi))**2
```

```
>>> def gPrime(xi):
...     return 4 * xi * (1 - xi) * (0.5 - xi)
```

and a dummy electrostatic potential field

```
>>> potential = CellVariable(mesh = mesh, name = 'phi', value = 0.)
>>> permittivityPrime = 0.
```

We start with a binary substitutional system

```
>>> class ComponentVariable(CellVariable):
...     def __init__(self, mesh, value = 0., name = '',
...                 standardPotential = 0., barrier = 0.,
...                 diffusivity = None, valence = 0, equation = None,
...                 hasOld = 1):
...         self.standardPotential = standardPotential
...         self.barrier = barrier
...         self.diffusivity = diffusivity
...         self.valence = valence
...         self.equation = equation
...         CellVariable.__init__(self, mesh = mesh, value = value,
...                               name = name, hasOld = hasOld)
...
...     def copy(self):
...         return self.__class__(mesh = self.mesh,
...                                value = self.value,
...                                name = self.name,
...                                standardPotential =
...                                    self.standardPotential,
...                                barrier = self.barrier,
...                                diffusivity = self.diffusivity,
...                                valence = self.valence,
...                                equation = self.equation,
...                                hasOld = 0)
```

consisting of the solvent

```
>>> solvent = ComponentVariable(mesh = mesh, name = 'Cn', value = 1.)
```

and the solute

```
>>> substitutionals = [
...     ComponentVariable(mesh = mesh, name = 'C1',
...                       diffusivity = 1., barrier = 0.,
...                       standardPotential = numerix.log(.3/.7) - numerix.log(.7/.3))]
>>> interstitials = []
```

```
>>> for component in substitutionals:
...     solvent -= component
```

The thermodynamic parameters are chosen to give a solid phase rich in the solute and a liquid phase rich in the solvent.

```
>>> solvent.standardPotential = numerix.log(.7/.3)
>>> solvent.barrier = 1.
```

We create the phase equation as in [examples.elphf.phase](#) and create the diffusion equations for the different species as in [examples.elphf.diffusion.mesh1D](#). The initial residual of the diffusion equations is much larger than the norm of the right-hand-side vector, so we use “initial” tolerance scaling for those equations

```
>>> def makeEquations(phase, substitutionals, interstitials):
...     phase.equation = TransientTerm(coeff = 1/phase.mobility) \
...         == DiffusionTerm(coeff = phase.gradientEnergy) \
...         - (permittivityPrime / 2.) \
```

(continues on next page)

(continued from previous page)

```

...         * potential.grad.dot(potential.grad)
...     enthalpy = solvent.standardPotential
...     barrier = solvent.barrier
...     for component in substitutionals + interstitials:
...         enthalpy += component * component.standardPotential
...         barrier += component * component.barrier
...
...     mXi = -(30 * phase * (1 - phase) * enthalpy
...           + 4 * (0.5 - phase) * barrier)
...     dmXidXi = (-60 * (0.5 - phase) * enthalpy + 4 * barrier)
...     S1 = dmXidXi * phase * (1 - phase) + mXi * (1 - 2 * phase)
...     S0 = mXi * phase * (1 - phase) - phase * S1
...
...     phase.equation -= S0 + ImplicitSourceTerm(coeff = S1)
...
...     for Cj in substitutionals:
...         CkSum = ComponentVariable(mesh = mesh, value = 0.)
...         CkFaceSum = FaceVariable(mesh = mesh, value = 0.)
...         for Ck in [Ck for Ck in substitutionals if Ck is not Cj]:
...             CkSum += Ck
...             CkFaceSum += Ck.harmonicFaceValue
...
...         counterDiffusion = CkSum.faceGrad
...         phaseTransformation = (pPrime(phase.harmonicFaceValue) \
...                               * Cj.standardPotential
...                               + gPrime(phase.harmonicFaceValue) \
...                               * Cj.barrier) * phase.faceGrad
...         electromigration = Cj.valence * potential.faceGrad
...         convectionCoeff = counterDiffusion + \
...             solvent.harmonicFaceValue \
...             * (phaseTransformation + electromigration)
...         convectionCoeff *= \
...             (Cj.diffusivity / (1. - CkFaceSum))
...
...         Cj.equation = (TransientTerm()
...                       == DiffusionTerm(coeff=Cj.diffusivity)
...                       + PowerLawConvectionTerm(coeff=convectionCoeff))
...
...     for Cj in interstitials:
...         phaseTransformation = (pPrime(phase.harmonicFaceValue) \
...                               * Cj.standardPotential \
...                               + gPrime(phase.harmonicFaceValue) \
...                               * Cj.barrier) * phase.faceGrad
...         electromigration = Cj.valence * potential.faceGrad
...         convectionCoeff = Cj.diffusivity \
...             * (1 + Cj.harmonicFaceValue) \
...             * (phaseTransformation + electromigration)
...
...         Cj.equation = (TransientTerm()
...                       == DiffusionTerm(coeff=Cj.diffusivity)
...                       + PowerLawConvectionTerm(coeff=convectionCoeff))

```

(continues on next page)

(continued from previous page)

```
...     for Cj in substitutionals + interstitials:
...         Cj.solver = Cj.equation.getDefaultSolver(criterion="initial",
...                                                    tolerance=1e-7)
```

```
>>> makeEquations(phase, substitutionals, interstitials)
```

We start with a sharp phase boundary

$$\xi = \begin{cases} 1 & \text{for } x \leq L/2, \\ 0 & \text{for } x > L/2, \end{cases}$$

or

```
>>> x = mesh.cellCenters[0]
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L / 2)
```

and with a uniform concentration field $C_1 = 0.5$ or

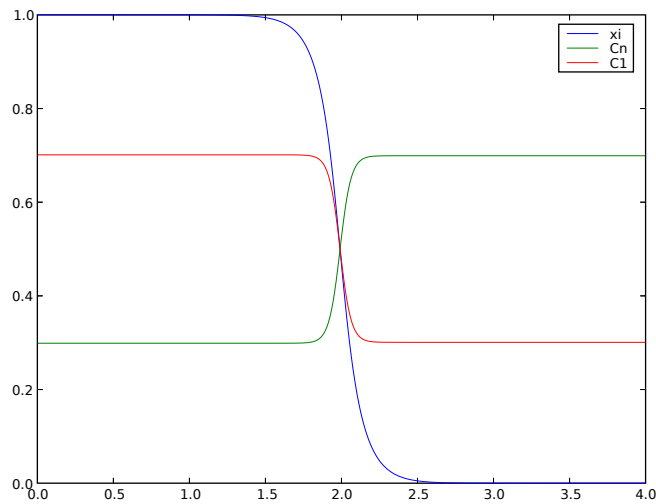
```
>>> substitutionals[0].setValue(0.5)
```

If running interactively, we create viewers to display the results

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=([phase, solvent]
...                           + substitutionals + interstitials),
...                      datamin=0, datamax=1)
...     viewer.plot()
```

This problem does not have an analytical solution, so after iterating to equilibrium

```
>>> dt = 10000
>>> 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=field.solver)
...     if __name__ == '__main__':
...         viewer.plot()
```



we confirm that the far-field phases have remained separated

```
>>> numerix.allclose(phase(((0., L),)), (1.0, 0.0), rtol = 1e-5, atol = 1e-5)
1
```

and that the solute concentration field has appropriately segregated into solute-rich and solute-poor phases.

```
>>> print(numerix.allclose(substitutionals[0](((0., L),)), (0.7, 0.3), rtol = 2e-3, atol =
↪ 2e-3))
1
```

The same system of equations can model a quaternary substitutional system as easily as a binary. Because it depends on the number of substitutional solute species in question, we recreate the solvent

```
>>> solvent = ComponentVariable(mesh = mesh, name = 'Cn', value = 1.)
```

and make three new solute species

```
>>> substitutionals = [
...     ComponentVariable(mesh = mesh, name = 'C1',
...         diffusivity = 1., barrier = 0.,
...         standardPotential = numerix.log(.3/.4) - numerix.log(.1/.2)),
...     ComponentVariable(mesh = mesh, name = 'C2',
...         diffusivity = 1., barrier = 0.,
...         standardPotential = numerix.log(.4/.3) - numerix.log(.1/.2)),
...     ComponentVariable(mesh = mesh, name = 'C3',
...         diffusivity = 1., barrier = 0.,
...         standardPotential = numerix.log(.2/.1) - numerix.log(.1/.2))]
```

```
>>> for component in substitutionals:
...     solvent -= component
>>> solvent.standardPotential = numerix.log(.1/.2)
>>> solvent.barrier = 1.
```

These thermodynamic parameters are chosen to give a solid phase rich in the solvent and the first substitutional com-

ponent and a liquid phase rich in the remaining two substitutional species.

Again, if we're running interactively, we create a viewer

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(phase, solvent]
...                     + substitutionals + interstitials),
...                     datamin=0, datamax=1)
...     viewer.plot()
```

We reinitialize the sharp phase boundary

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L / 2)
```

and the uniform concentration fields, with the substitutional concentrations $C_1 = C_2 = 0.35$ and $C_3 = 0.15$.

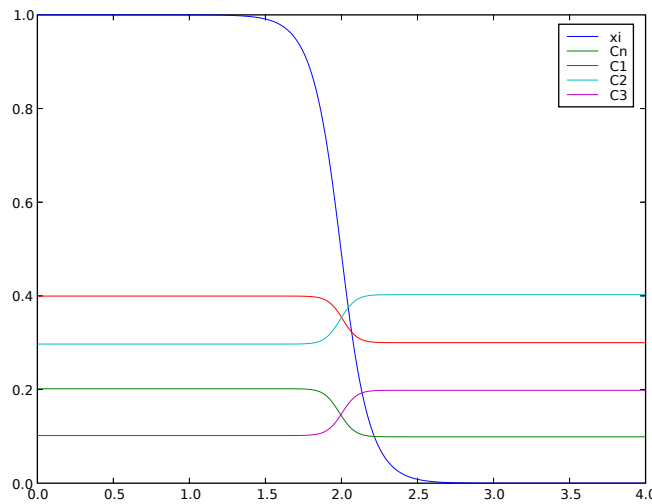
```
>>> substitutionals[0].setValue(0.35)
>>> substitutionals[1].setValue(0.35)
>>> substitutionals[2].setValue(0.15)
```

We make new equations

```
>>> makeEquations(phase, substitutionals, interstitials)
```

and again iterate to equilibrium

```
>>> dt = 10000
>>> 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=field.solver)
...     if __name__ == '__main__':
...         viewer.plot()
```

We confirm that the far-field phases have remained separated

```
>>> numerix.allclose(phase(((0., L)),), (1.0, 0.0), rtol = 1e-5, atol = 1e-5)
1
```

and that the concentration fields have appropriately segregated into their respective phases

```
>>> numerix.allclose(substitutionals[0](((0., L)),), (0.4, 0.3), rtol = 3e-3, atol = 3e-
↪3)
1
>>> numerix.allclose(substitutionals[1](((0., L)),), (0.3, 0.4), rtol = 3e-3, atol = 3e-
↪3)
1
>>> numerix.allclose(substitutionals[2](((0., L)),), (0.1, 0.2), rtol = 3e-3, atol = 3e-
↪3)
1
```

Finally, we can represent a system that contains both substitutional and interstitial species. We recreate the solvent

```
>>> solvent = ComponentVariable(mesh = mesh, name = 'Cn', value = 1.)
```

and two new solute species

```
>>> substitutionals = [
...     ComponentVariable(mesh = mesh, name = 'C2',
...                         diffusivity = 1., barrier = 0.,
...                         standardPotential = numerix.log(.4/.3) - numerix.log(.4/.6)),
...     ComponentVariable(mesh = mesh, name = 'C3',
...                         diffusivity = 1., barrier = 0.,
...                         standardPotential = numerix.log(.2/.1) - numerix.log(.4/.6))]
```

and one interstitial

```
>>> interstitials = [
...     ComponentVariable(mesh = mesh, name = 'C1',
```

(continues on next page)

(continued from previous page)

```
...         diffusivity = 1., barrier = 0.,
...         standardPotential = numerix.log(.3/.4) - numerix.log(1.3/1.4))]
```

```
>>> for component in substitutionals:
...     solvent -= component
>>> solvent.standardPotential = numerix.log(.4/.6) - numerix.log(1.3/1.4)
>>> solvent.barrier = 1.
```

These thermodynamic parameters are chosen to give a solid phase rich in interstitials and the solvent and a liquid phase rich in the two substitutional species.

Once again, if we're running interactively, we create a viewer

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=([phase, solvent]
...                           + substitutionals + interstitials),
...                       datamin=0, datamax=1)
...     viewer.plot()
```

We reinitialize the sharp phase boundary

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L / 2)
```

and the uniform concentration fields, with the interstitial concentration $C_1 = 0.35$

```
>>> interstitials[0].setValue(0.35)
```

and the substitutional concentrations $C_2 = 0.35$ and $C_3 = 0.15$.

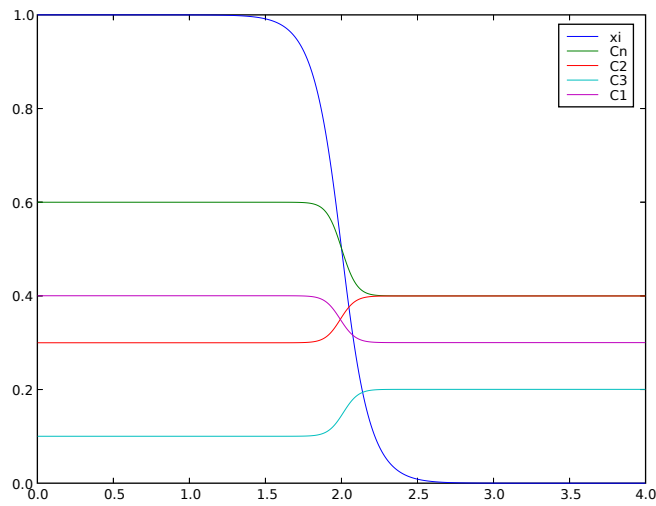
```
>>> substitutionals[0].setValue(0.35)
>>> substitutionals[1].setValue(0.15)
```

We make new equations

```
>>> makeEquations(phase, substitutionals, interstitials)
```

and again iterate to equilibrium

```
>>> dt = 10000
>>> 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=field.solver)
...     if __name__ == '__main__':
...         viewer.plot()
```



We once more confirm that the far-field phases have remained separated

```
>>> numerix.allclose(phase(((0., L))), (1.0, 0.0), rtol = 1e-5, atol = 1e-5)
1
```

and that the concentration fields have appropriately segregated into their respective phases

```
>>> numerix.allclose(interstitials[0](((0., L))), (0.4, 0.3), rtol = 3e-3, atol = 3e-3)
1
>>> numerix.allclose(substitutionals[0](((0., L))), (0.3, 0.4), rtol = 3e-3, atol = 3e-3)
1
>>> numerix.allclose(substitutionals[1](((0., L))), (0.1, 0.2), rtol = 3e-3, atol = 3e-3)
1
```

24.6.5 examples.elphf.poisson

A simple 1D example to test the setup of the Poisson equation.

```
>>> from fipy import CellVariable, Grid1D, DiffusionTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> nx = 200
>>> dx = 0.01
>>> L = nx * dx
>>> mesh = Grid1D(dx = dx, nx = nx)
```

The dimensionless Poisson equation is

$$\nabla \cdot (\epsilon \nabla \phi) = -\rho = -\sum_{j=1}^n z_j C_j$$

where ϕ is the electrostatic potential, ϵ is the permittivity, ρ is the charge density, C_j is the concentration of the j^{th} component, and z_j is the valence of the j^{th} component.

We will be solving for the electrostatic potential

```
>>> potential = CellVariable(mesh = mesh, name = 'phi', value = 0.)
>>> permittivity = 1.
```

We examine a fixed distribution of electrons with $z_{e^-} = -1$.

```
>>> class ComponentVariable(CellVariable):
...     def __init__(self, mesh, value = 0., name = '',
...                   standardPotential = 0., barrier = 0.,
...                   diffusivity = None, valence = 0, equation = None):
...         CellVariable.__init__(self, mesh = mesh,
...                               value = value, name = name)
...         self.standardPotential = standardPotential
...         self.barrier = barrier
...         self.diffusivity = diffusivity
...         self.valence = valence
...         self.equation = equation
...
...     def copy(self):
...         return self.__class__(mesh = self.mesh,
...                                value = self.value,
...                                name = self.name,
...                                standardPotential =
...                                    self.standardPotential,
...                                barrier = self.barrier,
...                                diffusivity = self.diffusivity,
...                                valence = self.valence,
...                                equation = self.equation)
```

Since we're only interested in a single species, electrons, we could simplify the following, but since we will in general be studying multiple components, we explicitly allow for multiple substitutional species and multiple interstitial species:

```
>>> interstitials = [
...     ComponentVariable(mesh = mesh, name = 'e-', valence = -1)]
>>> substitutionals = []
```

Because Poisson's 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)
```

```
>>> charge = 0.
>>> for Cj in interstitials + substitutionals:
...     charge += Cj * Cj.valence
```

```
>>> potential.equation = DiffusionTerm(coeff = permittivity) \
...     + charge == 0
```

First, we obtain a uniform charge distribution by setting a uniform concentration of electrons $C_{e^-} = 1$.

```
>>> interstitials[0].setValue(1.)
```

and we solve for the electrostatic potential

```
>>> potential.equation.solve(var = potential)
```

This problem has the analytical solution

$$\psi(x) = \frac{x^2}{2} - 2x$$

We verify that the correct equilibrium is attained

```
>>> x = mesh.cellCenters[0]
>>> analyticalArray = (x**2)/2 - 2*x
```

```
>>> print(potential.allclose(analyticalArray, 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))
...     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]
>>> interstitials[0].setValue(0.)
>>> interstitials[0].setValue(1., where=x > L / 2.)
```

and again solve for the electrostatic potential

```
>>> potential.equation.solve(var = potential)
```

which now has the analytical solution

$$\psi(x) = \begin{cases} -x & \text{for } x \leq L/2, \\ \frac{(x-1)^2}{2} - x & \text{for } x > L/2. \end{cases}$$

We verify that the correct equilibrium is attained

```
>>> analyticalArray = numerix.where(x < L/2, -x, ((x-1)**2)/2 - x)
```

```
>>> potential.allclose(analyticalArray, rtol = 2e-5, atol = 2e-5).value
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 left side of the domain

$$C_{e^-} = \begin{cases} 1 & \text{for } x \leq L/2, \\ 0 & \text{for } x > L/2. \end{cases}$$

```
>>> interstitials[0].setValue(1.)
>>> interstitials[0].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

```
>>> analyticalArray = numerix.where(x < 1, (x**2)/2 - x, -0.5)
```

```
>>> potential.allclose(analyticalArray, rtol = 2e-5, atol = 2e-5).value
1
```

and again view the result

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

24.6.6 examples.elphf.test

24.7 examples.flow

Modules

examples.flow.stokesCavity
examples.flow.test

Solve the Navier-Stokes equation in the viscous limit.

24.7.1 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 [25] and Rossow [26]. 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,

$$\nabla \cdot \frac{V_P}{a_P} \cdot \nabla p' = \nabla \cdot \vec{u}^*$$

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 $u^\diamond$ 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='vertical', scale=5)
```

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()`.

The pressure corrector needs to work a bit harder than the other equations, so we allow it to iterate longer to drive to a more demanding tolerance.

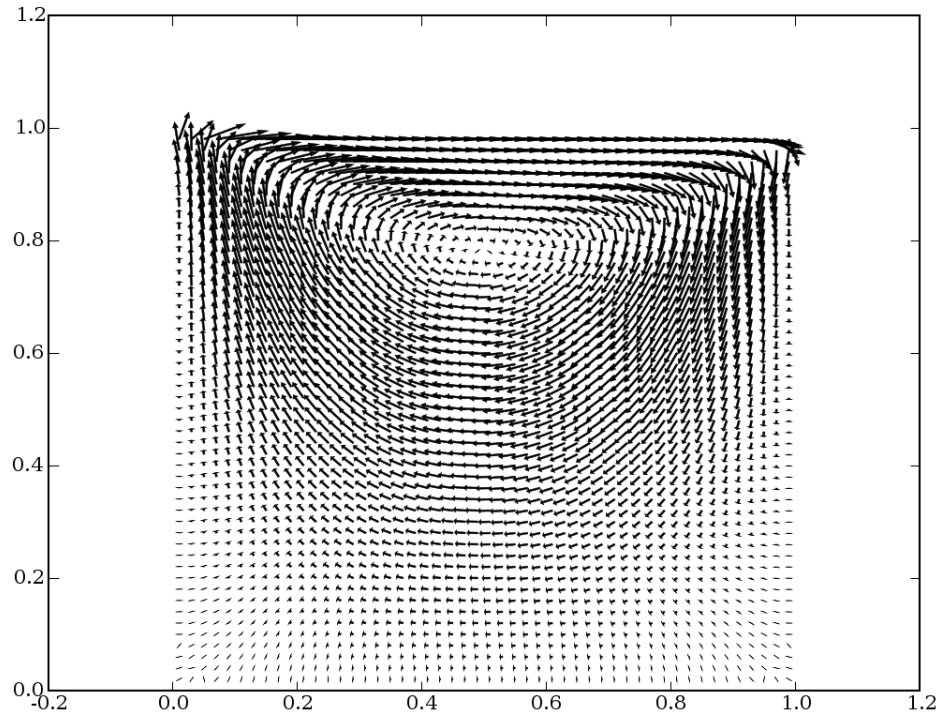
```
>>> solver = pressureCorrectionEq.getDefaultSolver(tolerance=1e-10, iterations=2000)
```

```
>>> 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
...     pres = pressureCorrectionEq.sweep(var=pressureCorrection, solver=solver)
...     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)))
```

(continues on next page)

(continued from previous page)

```
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))
1
```

24.7.2 examples.flow.test

24.8 examples.levelSet

Modules

<code>examples.levelSet.advection</code>	
<code>examples.levelSet.distanceFunction</code>	
<code>examples.levelSet.electroChem</code>	
<code>examples.levelSet.surfactant</code>	
<code>examples.levelSet.test</code>	Run all the test cases in examples/

24.8.1 examples.levelSet.advection

Modules

<code>examples.levelSet.advection.circle</code>	Solve a circular distance function equation and then advect it.
<code>examples.levelSet.advection.mesh1D</code>	Solve the distance function equation in one dimension and then advect it.
<code>examples.levelSet.advection.test</code>	
<code>examples.levelSet.advection.trench</code>	This example creates a trench with the following zero level set:

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.

```

>>> 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.

```

>>> 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))

```

(continues on next page)

(continued from previous page)

```
>>> 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)
>>> 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
```

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.,
```

(continues on next page)

(continued from previous page)

```

...             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.

```

>>> 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:

```

>>> 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
>>> answer = numerix.where(x < distanceTravelled,
...                        x[0] - interfacePosition, answer)
>>> print(var.allclose(answer))
1

```

examples.levelSet.advection.test

examples.levelSet.advection.trench

This example creates a trench with the following zero level set:

$$\begin{aligned} \phi(x, y) &= 0 \text{ when } y = L_y/5 \text{ and } x \geq L_x/2 \\ \phi(x, y) &= 0 \text{ when } L_y/5 \leq y \leq 3L_y/5 \text{ and } x = L_x/2 \\ \phi(x, y) &= 0 \text{ when } y = 3L_y/5 \text{ and } x \leq L_x/2 \end{aligned}$$

```

>>> from fipy import CellVariable, Grid2D, DistanceVariable, TransientTerm,
↳ FirstOrderAdvectionTerm, AdvectionTerm, Viewer
>>> from fipy.tools import numerix, serialComm

```

```

>>> height = 0.5
>>> Lx = 0.4
>>> Ly = 1.
>>> dx = 0.01
>>> velocity = 1.
>>> cfl = 0.1

```

```
>>> nx = int(Lx / dx)
>>> ny = int(Ly / dx)
>>> timeStepDuration = cfl * dx / velocity
>>> steps = 200
```

```
>>> mesh = Grid2D(dx = dx, dy = dx, nx = nx, ny = ny, communicator=serialComm)
```

```
>>> var = DistanceVariable(name = 'level set variable',
...                        mesh = mesh,
...                        value = -1.,
...                        hasOld = 1
...                        )
```

```
>>> x, y = mesh.cellCenters
>>> var.setValue(1, where=(y > 0.6 * Ly) | ((y > 0.2 * Ly) & (x > 0.5 * Lx)))
```

```
>>> var.calcDistanceFunction()
```

```
>>> advEqn = TransientTerm() + FirstOrderAdvectionTerm(velocity)
```

The trench is then advected with a unit velocity. The following test can be made for the initial position of the interface:

```
>>> r1 = -numerix.sqrt((x - Lx / 2)**2 + (y - Ly / 5)**2)
>>> r2 = numerix.sqrt((x - Lx / 2)**2 + (y - 3 * Ly / 5)**2)
>>> d = numerix.zeros((len(x), 3), 'd')
>>> d[:, 0] = numerix.where(x >= Lx / 2, y - Ly / 5, r1)
>>> d[:, 1] = numerix.where(x <= Lx / 2, y - 3 * Ly / 5, r2)
>>> d[:, 2] = numerix.where(numerix.logical_and(Ly / 5 <= y, y <= 3 * Ly / 5), x - Lx / 2, d[:, 0])
>>> argmins = numerix.argmin(numerix.absolute(d), axis = 1)
>>> answer = numerix.take(d.ravel(), numerix.arange(len(argmins))*3 + argmins)
>>> print(var.allclose(answer, atol = 1e-1))
1
```

Advect the interface and check the position.

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-0.1, datamax=0.1)
...
...     viewer.plot()
```

```
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt = timeStepDuration)
...     if __name__ == '__main__':
...         viewer.plot()
```

```
>>> distanceMoved = timeStepDuration * steps * velocity
>>> answer = answer - distanceMoved
>>> answer = numerix.where(answer < 0., 0., answer)
>>> var.setValue(numerix.where(var < 0., 0., var))
```

(continues on next page)

(continued from previous page)

```
>>> print(var.allclose(answer, atol = 1e-1))
1
```

24.8.2 examples.levelSet.distanceFunction

Modules

<code>examples.levelSet.distanceFunction.circle</code>	Solve the level set equation in two dimensions for a circle.
<code>examples.levelSet.distanceFunction.interior</code>	Here we solve the level set equation in two dimension for an interior region.
<code>examples.levelSet.distanceFunction.mesh1D</code>	Create a level set variable in one dimension.
<code>examples.levelSet.distanceFunction.square</code>	Here we solve the level set equation in two dimensions for a square.
<code>examples.levelSet.distanceFunction.test</code>	

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), 'd'))
```

```
>>> var[numerix.array(trialValues == MASK)] = MASK
>>> print(numerix.allclose(var, trialValues))
True
```

examples.levelSet.distanceFunction.interior

Here we solve the level set equation in two dimension for an interior region. The equation is given by:

$$|\nabla\phi| = 1$$

$$\phi = 0 \text{ at } \begin{cases} x = (d, L-d) \text{ for } d \leq y \leq L-d \\ y = (d, L-d) \text{ for } d \leq x \leq L-d \end{cases}$$

Do the tests:

```
>>> var.calcDistanceFunction(order=1)

>>> dX = dx / 2.
>>> dY = dy / 2.
>>> mm = dX * dY / numerix.sqrt(dX**2 + dY**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, dX, dx, dy)[1]
>>> v2 = max(-dY*3, -dX*3)
>>> values = numerix.array(( v1, dY, dY, dY, v1,
...                          dX, -mm, -dY, -mm, dX,
...                          dX, -dX, -v1, -dX, dX,
...                          dX, -mm, -dY, -mm, dX,
...                          v1, dY, dY, dY, v1 ))
>>> print(var.allclose(values, atol = 1e-10))
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
```

examples.levelSet.distanceFunction.square

Here we solve the level set equation in two dimensions for a square. The equation is given by:

$$|\nabla\phi| = 1$$

$$\phi = 0 \quad \text{at} \quad \begin{cases} x = (L/3, 2L/3) & \text{for } L/3 \leq y \leq 2L/3 \\ y = (L/3, 2L/3) & \text{for } L/3 \leq x \leq 2L/3 \end{cases}$$

Do the tests:

```
>>> var.calcDistanceFunction(order=1)
```

```
>>> 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)
>>> val = evalCell(-dy / 2., -dx / 2., dx, dy)[0]
>>> v1 = evalCell(val, -3. * dx / 2., dx, dy)[0]
>>> v2 = evalCell(-3. * dy / 2., val, dx, dy)[0]
>>> v3 = evalCell(v2, v1, dx, dy)[0]
>>> v4 = dx * dy / numerix.sqrt(dx**2 + dy**2) / 2
>>> arr = numerix.array((
...     v3, v2, -3. * dy / 2., v2, v3,
```

(continues on next page)

(continued from previous page)

```

...     v1, val, -dy / 2., val, v1,
...     -3. * dx / 2., -dx / 2., v4, -dx / 2., -3. * dx / 2.,
...     v1, val, -dy / 2., val, v1,
...     v3, v2, -3. * dy / 2., v2, v3          ))
>>> print(var.allclose(arr))
1

```

`examples.levelSet.distanceFunction.test`

24.8.3 `examples.levelSet.electroChem`

Modules

<code>examples.levelSet.electroChem.adsorbingSurfactantEquation</code>	
<code>examples.levelSet.electroChem.adsorption</code>	This example tests 1D adsorption onto an interface and subsequent depletion from the bulk.
<code>examples.levelSet.electroChem.gapFillDistanceVariable</code>	
<code>examples.levelSet.electroChem.gapFillMesh</code>	The <code>gapFillMesh</code> function glues 3 meshes together to form a composite mesh.
<code>examples.levelSet.electroChem.gold</code>	Model electrochemical superfill of gold using the CEAC mechanism.
<code>examples.levelSet.electroChem.howToWriteAScript</code>	Tutorial for writing an electrochemical superfill script.
<code>examples.levelSet.electroChem.leveler</code>	Model electrochemical superfill of copper with leveler and accelerator additives.
<code>examples.levelSet.electroChem.lines</code>	
<code>examples.levelSet.electroChem.matplotlibSurfactantViewer</code>	
<code>examples.levelSet.electroChem.mayaviSurfactantViewer</code>	
<code>examples.levelSet.electroChem.metalIonDiffusionEquation</code>	
<code>examples.levelSet.electroChem.simpleTrenchSystem</code>	Model electrochemical superfill using the CEAC mechanism.
<code>examples.levelSet.electroChem.surfactantBulkDiffusionEquation</code>	
<code>examples.levelSet.electroChem.test</code>	
<code>examples.levelSet.electroChem.trenchMesh</code>	

examples.levelSet.electroChem.adsorbingSurfactantEquation**examples.levelSet.electroChem.adsorption**

This example tests 1D adsorption onto an interface and subsequent depletion from the bulk. The governing equations are given by,

$$c_t = Dc_{xx}$$

$$Dc_x = \Gamma kc(1 - \theta) \quad \text{at } x = 0$$

and

$$c = c^\infty \quad \text{at } x = L$$

and on the interface

$$Dc_x = -kc(1 - \theta) \quad \text{at } x = 0$$

There is a dimensionless number M that governs whether the system is in an interface limited ($M \gg 1$) or diffusion limited ($M \ll 1$) regime. There are analytical solutions for both regimes. The dimensionless number is given by:

$$M = \frac{D}{L^2 k c_{inf}}.$$

The test solution provided here is for the case of interface limited kinetics. The analytical solutions are given by,

$$-D \ln(1 - \theta) + kL\Gamma_0\theta = \frac{kDc^\infty t}{\Gamma_0}$$

and

$$c(x) = \frac{c^\infty [k\Gamma_0(1 - \theta)x/D]}{1 + k\Gamma_0(1 - \theta)L/D}$$

Make sure the dimensionless parameter is large enough

```
>>> (diffusion / cinf / L / L / rateConstant) > 100
True
```

Start time stepping:

```
>>> currentTime = 0.
>>> for i in range(totalTimeSteps):
...     surfEqn.solve(surfactantVar, dt = dt)
...     bulkEqn.solve(bulkVar, dt = dt)
...     currentTime += dt
```

Compare the analytical and numerical results:

```
>>> theta = surfactantVar.interfaceVar[1]
```

```
>>> numerix.allclose(currentTimeFunc(theta), currentTime, rtol = 1e-4)()
1
>>> numerix.allclose(concentrationFunc(theta), bulkVar[1:], rtol = 1e-4)()
1
```

examples.levelSet.electroChem.gapFillDistanceVariable

examples.levelSet.electroChem.gapFillMesh

The *gapFillMesh* function glues 3 meshes together to form a composite mesh. The first mesh is a *Grid2D* object that is fine and deals with the area around the trench or via. The second mesh is a *Gmsh2D* object that forms a transition mesh from a fine to a course region. The third mesh is another *Grid2D* object that forms the boundary layer. This region consists of very large elements and is only used for the diffusion in the boundary layer.

examples.levelSet.electroChem.gold

Model electrochemical superfill of gold using the CEAC mechanism.

This input file is a demonstration of the use of *FiPy* for modeling gold superfill. The material properties and experimental parameters used are roughly those that have been previously published [27].

To run this example from the base FiPy directory type:

```
$ python examples/levelSet/electroChem/gold.py
```

at the command line. The results of the simulation will be displayed and the word **finished** in the terminal at the end of the simulation. The simulation will only run for 10 time steps. To run with a different number of time steps change the `numberOfSteps` argument as follows,

```
>>> runGold(numberOfSteps=10, displayViewers=False)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. This example has a more realistic default boundary layer depth and thus requires *gmsh* to construct a more complex mesh.

There are a few differences between the gold superfill model presented in this example and in [examples.levelSet.electroChem.simpleTrenchSystem](#). Most default values have changed to account for a different metal ion (gold) and catalyst (lead). In this system the catalyst is not present in the electrolyte but instead has a non-zero initial coverage. Thus quantities associated with bulk catalyst and catalyst accumulation are not defined. The current density is given by,

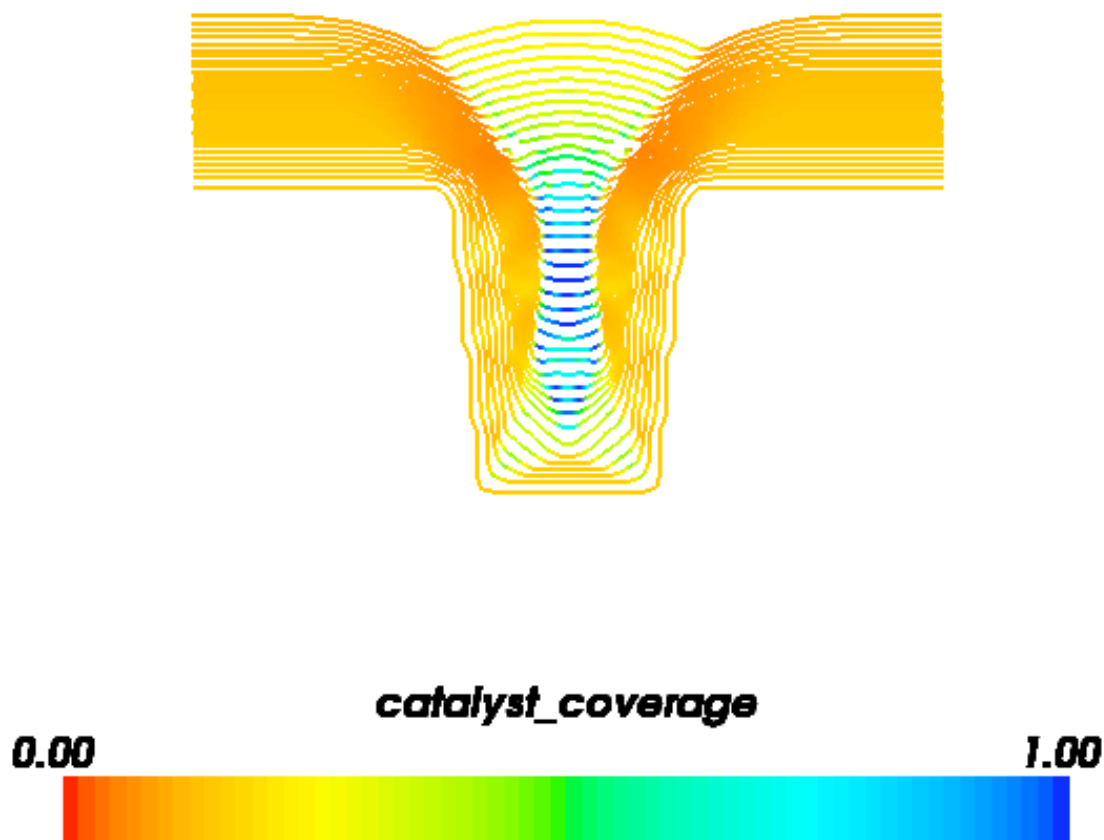
$$i = \frac{c_m}{c_m^\infty} (b_0 + b_1 \theta) .$$

The more common representation of the current density includes an exponential part. Here it is buried in b_0 and b_1 . The governing equation for catalyst evolution includes a term for catalyst consumption on the interface and is given by

$$\dot{\theta} = Jv\theta - k_c v \theta$$

where k_c is the consumption coefficient (`consumptionRateConstant`). The trench geometry is also given a slight taper, given by `taperAngle`.

If the Mayavi plotting software is installed (see [Installation](#)) then a plot should appear that is updated every 10 time steps and will eventually resemble the image below.



`examples.levelSet.electroChem.howToWriteAScript`

Tutorial for writing an electrochemical superfill script.

This input file demonstrates how to create a new superfill script if the existing suite of scripts do not meet the required needs. It provides the functionality of `examples.levelSet.electroChem.simpleTrenchSystem`.

To run this example from the base FiPy directory type:

```
$ python examples/levelSet/electroChem/howToWriteAScript.py --numberOfElements=10000 --
↪ numberOfSteps=800
```

at the command line. The results of the simulation will be displayed and the word `finished` in the terminal at the end of the simulation. To obtain this example in a plain script file in order to edit and run type:

```
$ fipy_copy_script examples/levelSet/electroChem/howToWriteAScript.py myScript.py
```

in the base *FiPy* directory. The file `myScript.py` will contain the script.

The following is an explicit explanation of the input commands required to set up and run the problem. At the top of the file all the parameter values are set. Their use will be explained during the instantiation of various objects and are the same as those explained in `examples.levelSet.electroChem.simpleTrenchSystem`.

The following parameters (all in S.I. units) represent,

- physical constants,


```
>>> faradaysConstant = 9.6e4
>>> gasConstant = 8.314
>>> transferCoefficient = 0.5
```

- properties associated with the catalyst species,

```
>>> rateConstant0 = 1.76
>>> rateConstant3 = -245e-6
>>> catalystDiffusion = 1e-9
>>> siteDensity = 9.8e-6
```

- properties of the cupric ions,

```
>>> molarVolume = 7.1e-6
>>> charge = 2
>>> metalDiffusionCoefficient = 5.6e-10
```

- parameters dependent on experimental constraints,

```
>>> temperature = 298.
>>> overpotential = -0.3
>>> bulkMetalConcentration = 250.
>>> catalystConcentration = 5e-3
>>> catalystCoverage = 0.
```

- parameters obtained from experiments on flat copper electrodes,

```
>>> currentDensity0 = 0.26
>>> currentDensity1 = 45.
```

- general simulation control parameters,

```
>>> cflNumber = 0.2
>>> numberOfCellsInNarrowBand = 10
>>> cellsBelowTrench = 10
>>> cellSize = 0.1e-7
```

- parameters required for a trench geometry,

```
>>> trenchDepth = 0.5e-6
>>> aspectRatio = 2.
>>> trenchSpacing = 0.6e-6
>>> boundaryLayerDepth = 0.3e-6
```

The hydrodynamic boundary layer depth (`boundaryLayerDepth`) is intentionally small in this example to keep the mesh at a reasonable size.

Build the mesh:

```
>>> from fipy.tools.parser import parse
>>> numberOfElements = parse('--numberOfElements', action='store',
...     type='int', default=-1)
>>> numberOfSteps = parse('--numberOfSteps', action='store',
...     type='int', default=2)
```

```
>>> from fipy import *
```

```
>>> if numberOfElements != -1:
...     pos = trenchSpacing * cellsBelowTrench / 4 / numberOfElements
...     sqr = trenchSpacing * (trenchDepth + boundaryLayerDepth) \
...           / (2 * numberOfElements)
...     cellSize = pos + numerix.sqrt(pos**2 + sqr)
... else:
...     cellSize = 0.1e-7
```

```
>>> yCells = cellsBelowTrench \
...       + int((trenchDepth + boundaryLayerDepth) / cellSize)
>>> xCells = int(trenchSpacing / 2 / cellSize)
```

```
>>> from .metalIonDiffusionEquation import buildMetalIonDiffusionEquation
>>> from .adsorbingSurfactantEquation import AdsorbingSurfactantEquation
```

```
>>> from fipy import serialComm
>>> mesh = Grid2D(dx=cellSize,
...              dy=cellSize,
...              nx=xCells,
...              ny=yCells,
...              communicator=serialComm)
```

A distanceVariable object, ϕ , is required to store the position of the interface.

The distanceVariable calculates its value so that it is a distance function (*i.e.* holds the distance at any point in the mesh from the electrolyte/metal interface at $\phi = 0$) and $|\nabla\phi| = 1$.

First, create the ϕ variable, which is initially set to -1 everywhere. Create an initial variable,

```
>>> narrowBandWidth = numberOfCellsInNarrowBand * cellSize
>>> distanceVar = DistanceVariable(
...     name='distance variable',
...     mesh= mesh,
...     value=-1.,
...     hasOld=1)
```

The electrolyte region will be the positive region of the domain while the metal region will be negative.

```
>>> bottomHeight = cellsBelowTrench * cellSize
>>> trenchHeight = bottomHeight + trenchDepth
>>> trenchWidth = trenchDepth / aspectRatio
>>> sideWidth = (trenchSpacing - trenchWidth) / 2
```

```
>>> x, y = mesh.cellCenters
>>> distanceVar.setValue(1., where=(y > trenchHeight)
...                       | ((y > bottomHeight)
...                         & (x < xCells * cellSize - sideWidth)))
```

```
>>> distanceVar.calcDistanceFunction(order=2)
```

The distanceVariable has now been created to mark the interface. Some other variables need to be created that govern the concentrations of various species.

Create the catalyst surfactant coverage, θ , variable. This variable influences the deposition rate.

```
>>> catalystVar = SurfactantVariable(
...     name="catalyst variable",
...     value=catalystCoverage,
...     distanceVar=distanceVar)
```

Create the bulk catalyst concentration, c_θ , in the electrolyte,

```
>>> bulkCatalystVar = CellVariable(
...     name='bulk catalyst variable',
...     mesh=mesh,
...     value=catalystConcentration)
```

Create the bulk metal ion concentration, c_m , in the electrolyte.

```
>>> metalVar = CellVariable(
...     name='metal variable',
...     mesh=mesh,
...     value=bulkMetalConcentration)
```

The following commands build the depositionRateVariable, v . The depositionRateVariable is given by the following equation.

$$v = \frac{i\Omega}{nF}$$

where Ω is the metal molar volume, n is the metal ion charge and F is Faraday's constant. The current density is given by

$$i = i_0 \frac{c_m^i}{c_m^\infty} \exp\left(\frac{-\alpha F}{RT} \eta\right)$$

where c_m^i is the metal ion concentration in the bulk at the interface, c_m^∞ is the far-field bulk concentration of metal ions, α is the transfer coefficient, R is the gas constant, T is the temperature and η is the overpotential. The exchange current density is an empirical function of catalyst coverage,

$$i_0(\theta) = b_0 + b_1\theta$$

The commands needed to build this equation are,

```
>>> expoConstant = -transferCoefficient * faradaysConstant \
...               / (gasConstant * temperature)
>>> tmp = currentDensity1 \
...     * catalystVar.interfaceVar
>>> exchangeCurrentDensity = currentDensity0 + tmp
>>> expo = numerix.exp(expoConstant * overpotential)
>>> currentDensity = expo * exchangeCurrentDensity * metalVar \
...               / bulkMetalConcentration
>>> depositionRateVariable = currentDensity * molarVolume \
...               / (charge * faradaysConstant)
```

Build the extension velocity variable v_{ext} . The extension velocity uses the extensionEquation to spread the velocity at the interface to the rest of the domain.

```
>>> extensionVelocityVariable = CellVariable(
...     name='extension velocity',
...     mesh=mesh,
...     value=depositionRateVariable)
```

Using the variables created above the governing equations will be built. The governing equation for surfactant conservation is given by,

$$\dot{\theta} = Jv\theta + kc_{\theta}^i(1 - \theta)$$

where θ is the coverage of catalyst at the interface, J is the curvature of the interface, v is the normal velocity of the interface, c_{θ}^i is the concentration of catalyst in the bulk at the interface. The value k is given by an empirical function of overpotential,

$$k = k_0 + k_3\eta^3$$

The above equation is represented by the `AdsorbingSurfactantEquation` in *FiPy*:

```
>>> surfactantEquation = AdsorbingSurfactantEquation(
...     surfactantVar=catalystVar,
...     distanceVar=distanceVar,
...     bulkVar=bulkCatalystVar,
...     rateConstant=rateConstant0 \
...         + rateConstant3 * overpotential**3)
```

The variable ϕ is advected by the `advectionEquation` given by,

$$\frac{\partial \phi}{\partial t} + v_{\text{ext}}|\nabla \phi| = 0$$

and is set up with the following commands:

```
>>> advectionEquation = TransientTerm() + AdvectionTerm(extensionVelocityVariable)
```

The diffusion of metal ions from the far field to the interface is governed by,

$$\frac{\partial c_m}{\partial t} = \nabla \cdot D \nabla c_m$$

where,

$$D = \begin{cases} D_m & \text{when } \phi > 0, \\ 0 & \text{when } \phi \leq 0 \end{cases}$$

The following boundary condition applies at $\phi = 0$,

$$D\hat{n} \cdot \nabla c = \frac{v}{\Omega}.$$

The metal ion diffusion equation is set up with the following commands.

```
>>> metalEquation = buildMetalIonDiffusionEquation(
...     ionVar=metalVar,
...     distanceVar=distanceVar,
...     depositionRate=depositionRateVariable,
...     diffusionCoeff=metalDiffusionCoefficient,
...     metalIonMolarVolume=molarVolume,
... )
```

```
>>> metalVar.constrain(bulkMetalConcentration, mesh.facesTop)
```

The surfactant bulk diffusion equation solves the bulk diffusion of a species with a source term for the jump from the bulk to an interface. The governing equation is given by,

$$\frac{\partial c}{\partial t} = \nabla \cdot D \nabla c$$

where,

$$D = \begin{cases} D_\theta & \text{when } \phi > 0 \\ 0 & \text{when } \phi \leq 0 \end{cases}$$

The jump condition at the interface is defined by Langmuir adsorption. Langmuir adsorption essentially states that the ability for a species to jump from an electrolyte to an interface is proportional to the concentration in the electrolyte, available site density and a jump coefficient. The boundary condition at $\phi = 0$ is given by,

$$D \hat{n} \cdot \nabla c = -kc(1 - \theta).$$

The surfactant bulk diffusion equation is set up with the following commands.

```
>>> from .surfactantBulkDiffusionEquation import buildSurfactantBulkDiffusionEquation
>>> bulkCatalystEquation = buildSurfactantBulkDiffusionEquation(
...     bulkVar=bulkCatalystVar,
...     distanceVar=distanceVar,
...     surfactantVar=catalystVar,
...     diffusionCoeff=catalystDiffusion,
...     rateConstant=rateConstant0 * siteDensity
... )
```

```
>>> bulkCatalystVar.constrain(catalystConcentration, mesh.facesTop)
```

If running interactively, create viewers.

```
>>> if __name__ == '__main__':
...     try:
...         from .mayaviSurfactantViewer import MayaviSurfactantViewer
...         viewer = MayaviSurfactantViewer(distanceVar,
...                                           catalystVar.interfaceVar,
...                                           zoomFactor=1e6,
...                                           datamax=1.0,
...                                           datamin=0.0,
...                                           smooth=1)
...     except:
...         viewer = MultiViewer(viewers=(
...             Viewer(distanceVar, datamin=-1e-9, datamax=1e-9),
...             Viewer(catalystVar.interfaceVar)))
...         from fipy.models.levelSet.surfactant.matplotlibSurfactantViewer import ↵
↵ MatplotlibSurfactantViewer
...         viewer = MatplotlibSurfactantViewer(catalystVar.interfaceVar)
...     else:
...         viewer = None
```

The levelSetUpdateFrequency defines how often to call the distanceEquation to reinitialize the distanceVariable to a distance function.

```
>>> levelSetUpdateFrequency = int(0.8 * narrowBandWidth \
...                               / (cellSize * cflNumber * 2))
```

The following loop runs for `numberOfSteps` time steps. The time step is calculated with the CFL number and the maximum extension velocity. v to v_{ext} throughout the whole domain using $\nabla\phi \cdot \nabla v_{\text{ext}} = 0$.

```
>>> for step in range(numberOfSteps):
...     if viewer is not None:
...         viewer.plot()
...
...     if step % levelSetUpdateFrequency == 0:
...         distanceVar.calcDistanceFunction(order=2)
...
...     extensionVelocityVariable.setValue(depositionRateVariable())
...
...     distanceVar.updateOld()
...     distanceVar.extendVariable(extensionVelocityVariable, order=2)
...     dt = cflNumber * cellSize / extensionVelocityVariable.max()
...     advectionEquation.solve(distanceVar, dt=dt)
...     surfactantEquation.solve(catalystVar, dt=dt)
...     metalEquation.solve(var=metalVar, dt=dt)
...     bulkCatalystEquation.solve(var=bulkCatalystVar, dt=dt, solver=GeneralSolver())
```

The following is a short test case. It uses saved data from a simulation with 5 time steps. It is not a test for accuracy but a way to tell if something has changed or been broken.

```
>>> import os
```

```
>>> filepath = os.path.join(os.path.split(__file__)[0],
...                          "simpleTrenchSystem.gz")
...
>>> ##numerix.savetxt(filepath, numerix.array(catalystVar))
>>> print(catalystVar.allclose(numerix.loadtxt(filepath), rtol=1e-4))
1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input('finished')
```

examples.levelSet.electroChem.leveler

Model electrochemical superfill of copper with leveler and accelerator additives.

This input file is a demonstration of the use of *FiPy* for modeling copper superfill with leveler and accelerator additives. The material properties and experimental parameters used are roughly those that have been previously published [28].

To run this example from the base FiPy directory type:

```
$ python examples/levelSet/electroChem/leveler.py
```

at the command line. The results of the simulation will be displayed and the word `finished` in the terminal at the end of the simulation. The simulation will only run for 200 time steps. To run with a different number of time steps change the `numberOfSteps` argument as follows,

```
>>> runLeveler(numberOfSteps=10, displayViewers=False, cellSize=0.25e-7)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. This example requires *gmsh* to construct the mesh.

This example models the case when suppressor, accelerator and leveler additives are present in the electrolyte. The suppressor is assumed to adsorb quickly compared with the other additives. Any unoccupied surface sites are immediately covered with suppressor. The accelerator additive has more surface affinity than suppressor and is thus preferential adsorbed. The accelerator can also remove suppressor when the surface reaches full coverage. Similarly, the leveler additive has more surface affinity than both the suppressor and accelerator. This forms a simple set of assumptions for understanding the behavior of these additives.

The following is a complete description of the equations for the model described here. Any equations that have been omitted are the same as those given in [examples.levelSet.electroChem.simpleTrenchSystem](#). The current density is governed by

$$i = \frac{c_m}{c_m^\infty} \sum_j \left[i_j \theta_j \left(\exp \frac{-\alpha_j F \eta}{RT} - \exp \frac{(1 - \alpha_j) F \eta}{RT} \right) \right]$$

where j represents S for suppressor, A for accelerator, L for leveler and V for vacant. This model assumes a linear interpolation between the three cases of complete coverage for each additive or vacant substrate. The governing equations for the surfactants are given by,

$$\begin{aligned} \dot{\theta}_L &= \kappa v \theta_L + k_L^+ c_L (1 - \theta_L) - k_L^- v \theta_L, \\ \dot{\theta}_A &= \kappa v \theta_A + k_A^+ c_A (1 - \theta_A - \theta_L) - k_L c_L \theta_A - k_A^- \theta_A^{q-1}, \\ \theta_S &= 1 - \theta_A - \theta_L \\ \theta_V &= 0. \end{aligned}$$

It has been found experimentally that $i_L = i_S$.

If the surface reaches full coverage, the equations do not naturally prevent the coverage rising above full coverage due to the curvature terms. Thus, when $\theta_L + \theta_A = 1$ then the equation for accelerator becomes $\dot{\theta}_A = -\dot{\theta}_L$ and when $\theta_L = 1$, the equation for leveler becomes $\dot{\theta}_L = -k_L^- v \theta_L$.

The parameters k_A^+ , k_A^- and q are both functions of η given by,

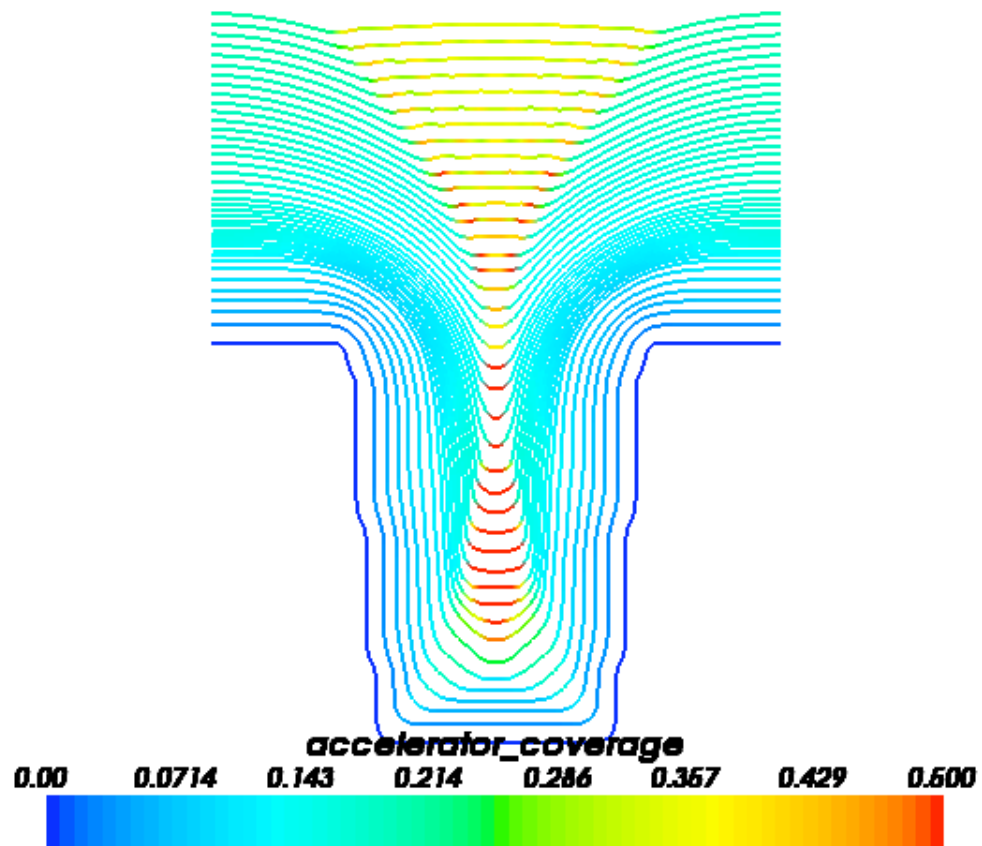
$$\begin{aligned} k_A^+ &= k_{A0}^+ \exp \frac{-\alpha_k F \eta}{RT}, \\ k_A^- &= B_d + \frac{A}{\exp(B_a(\eta + V_d))} + \exp(B_b(\eta + V_d)) \\ q &= m\eta + b. \end{aligned}$$

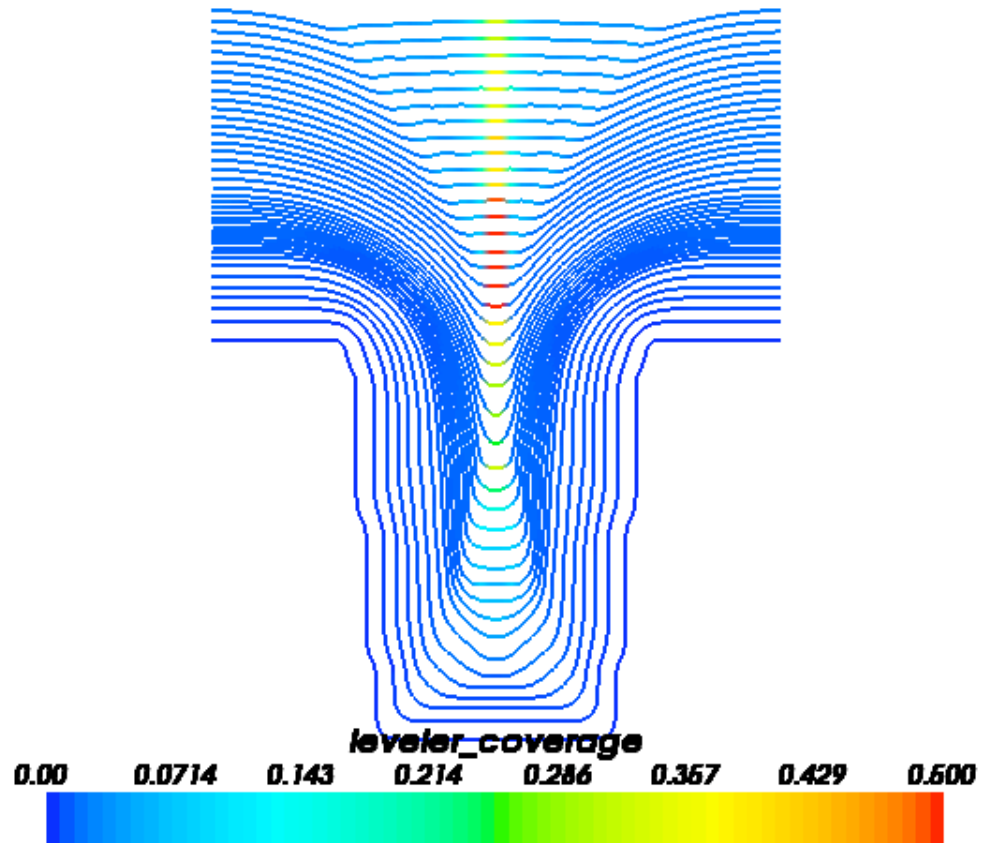
The following table shows the symbols used in the governing equations and their corresponding arguments for the

runLeveler() function.

Symbol	Description	Keyword Argument	Value	Unit
Deposition Rate Parameters				
v	deposition rate			m s^{-1}
i_A	accelerator current density	<code>i0Accelerator</code>		A m^{-2}
i_L	leveler current density	<code>i0Leveler</code>		A m^{-2}
Ω	molar volume	<code>molarVolume</code>	7.1×10^{-6}	$\text{m}^3 \text{mol}^{-1}$
n	ion charge	<code>charge</code>	2	
F	Faraday's constant	<code>faradaysConstant</code>	9.6×10^{-4}	C mol^{-1}
i_0	exchange current density			A m^{-2}
α_A	accelerator transfer coefficient	<code>alphaAccelerator</code>	0.4	
α_S	leveler transfer coefficient	<code>alphaLeveler</code>	0.5	
η	overpotential	<code>overpotential</code>	-0.3	V
R	gas constant	<code>gasConstant</code>	8.314	J K mol^{-1}
T	temperature	<code>temperature</code>	298.0	K
Ion Parameters				
c_I	ion concentration	<code>ionConcentration</code>	250.0	mol m^{-3}
c_I^∞	far field ion concentration	<code>ionConcentration</code>	250.0	mol m^{-3}
D_I	ion diffusion coefficient	<code>ionDiffusion</code>	5.6×10^{-10}	$\text{m}^2 \text{s}^{-1}$
Accelerator Parameters				
θ_A	accelerator coverage	<code>acceleratorCoverage</code>	0.0	
c_A	accelerator concentration	<code>acceleratorConcentration</code>	5.0×10^{-3}	mol m^{-3}
c_A^∞	far field accelerator concentration	<code>acceleratorConcentration</code>	5.0×10^{-3}	mol m^{-3}
D_A	catalyst diffusion coefficient	<code>catalystDiffusion</code>	1.0×10^{-9}	$\text{m}^2 \text{s}^{-1}$
Γ_A	accelerator site density	<code>siteDensity</code>	9.8×10^{-6}	mol m^{-2}
k_A^+	accelerator adsorption			$\text{m}^3 \text{mol}^{-1} \text{s}^{-1}$
k_{A0}^+	accelerator adsorption coeff	<code>kAccelerator0</code>	2.6×10^{-4}	$\text{m}^3 \text{mol}^{-1} \text{s}^{-1}$
α_k	accelerator adsorption coeff	<code>alphaAdsorption</code>	0.62	
k_A^-	accelerator consumption coeff			
B_a	experimental parameter	<code>Bd</code>	-40.0	
B_b	experimental parameter	<code>Bd</code>	60.0	
V_d	experimental parameter	<code>Bd</code>	9.8×10^{-2}	
B_d	experimental parameter	<code>Bd</code>	8.0×10^{-4}	
Geometry Parameters				
D	trench depth	<code>trenchDepth</code>	0.5×10^{-6}	m
D/W	trench aspect ratio	<code>aspectRatio</code>	2.0	
S	trench spacing	<code>trenchSpacing</code>	0.6×10^{-6}	m
δ	boundary layer depth	<code>boundaryLayerDepth</code>	0.3×10^{-6}	m
Simulation Control Parameters				
	computational cell size	<code>cellSize</code>	0.1×10^{-7}	m
	number of time steps	<code>numberOfSteps</code>	5	
	whether to display the viewers	<code>displayViewers</code>	True	

The following images show accelerator and leveler contour plots that can be obtained by running this example.





`examples.levelSet.electroChem.lines`

`examples.levelSet.electroChem.matplotlibSurfactantViewer`

Classes

`MatplotlibSurfactantViewer`(distanceVar[, ...])

The *MatplotlibSurfactantViewer* creates a viewer with the *Matplotlib* python plotting package that displays a *DistanceVariable*.

```

class examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer(distanceVar,
                                                                                          sur-
                                                                                          fac-
                                                                                          tant-
                                                                                          Var=None,
                                                                                          lev-
                                                                                          elSet-
                                                                                          Value=0.0,
                                                                                          ti-
                                                                                          tle=None,
                                                                                          smooth=0,
                                                                                          zoom-
                                                                                          Fac-
                                                                                          tor=1.0,
                                                                                          an-
                                                                                          i-
                                                                                          mate=False,
                                                                                          lim-
                                                                                          its={},
                                                                                          **kwlim-
                                                                                          its)

```

Bases: [AbstractMatplotlibViewer](#)

The *MatplotlibSurfactantViewer* creates a viewer with the [Matplotlib](#) python plotting package that displays a *DistanceVariable*.

Create a *MatplotlibSurfactantViewer*.

```

>>> from fipy import *
>>> m = Grid2D(nx=100, ny=100)
>>> x, y = m.cellCenters
>>> v = CellVariable(mesh=m, value=x**2 + y**2 - 10**2)
>>> s = CellVariable(mesh=m, value=sin(x / 10) * cos(y / 30))
>>> viewer = MatplotlibSurfactantViewer(distanceVar=v, surfactantVar=s)
>>> for r in range(1, 200):
...     v.setValue(x**2 + y**2 - r**2)
...     viewer.plot()

```

```

>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction.distanceVariable import
↳DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)

```

```

>>> x, y = mesh.cellCenters

```

```

>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)

```

(continues on next page)

(continued from previous page)

```
>>> var.calcDistanceFunction()
>>> viewer = MatplotlibSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

```
>>> var = DistanceVariable(mesh = mesh, value = -1)
```

```
>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) |
↳ ((y < Ly / 6.) & (x > Lx / 2)))
>>> var.calcDistanceFunction()
>>> viewer = MatplotlibSurfactantViewer(var)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

```
>>> viewer = MatplotlibSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

Parameters

- **distanceVar** (*DistanceVariable*) –
- **levelSetValue** (*float*) – the value of the contour to be displayed
- **title** (*str*) – displayed at the top of the *Viewer* window
- **animate** (*bool*) – whether to show only the initial condition and the moving top boundary or to show all contours (Default)
- **limits** (*dict, optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.
- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.

- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib Figure* number.

property log

Whether data has logarithmic scaling (*bool*).

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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.
- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.

- **zmin** ([float](#), *optional*) – 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.
- **zmax** ([float](#), *optional*) – 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.
- **datamin** ([float](#), *optional*) – 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.
- **datamax** ([float](#), *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only [Variable](#), otherwise `""`.)

property vars

The [Variable](#) or list of [Variable](#) objects to display.

examples.levelSet.electroChem.mayaviSurfactantViewer**Classes**

MayaviSurfactantViewer(distanceVar[, ...])

The *MayaviSurfactantViewer* creates a viewer with the [Mayavi](#) python plotting package that displays a *Distance-Variable*.

```
class examples.levelSet.electroChem.mayaviSurfactantViewer.MayaviSurfactantViewer(distanceVar,
                                                                                      surfac-
                                                                                      tant-
                                                                                      Var=None,
                                                                                      levelSet-
                                                                                      Value=0.0,
                                                                                      ti-
                                                                                      tle=None,
                                                                                      smooth=0,
                                                                                      zoomFac-
                                                                                      tor=1.0,
                                                                                      ani-
                                                                                      mate=False,
                                                                                      lim-
                                                                                      its={},
                                                                                      **kwlim-
                                                                                      its)
```

Bases: [AbstractViewer](#)

The *MayaviSurfactantViewer* creates a viewer with the [Mayavi](#) python plotting package that displays a *Distance-Variable*.

Create a *MayaviSurfactantViewer*.

```
>>> from fipy import *
>>> dx = 1.
>>> dy = 1.
>>> nx = 11
>>> ny = 11
>>> Lx = ny * dy
>>> Ly = nx * dx
>>> mesh = Grid2D(dx = dx, dy = dy, nx = nx, ny = ny)
>>> # from fipy.models.levelSet.distanceFunction.distanceVariable import
↳DistanceVariable
>>> var = DistanceVariable(mesh = mesh, value = -1)
```

```
>>> x, y = mesh.cellCenters
```

```
>>> var.setValue(1, where=(x - Lx / 2.)**2 + (y - Ly / 2.)**2 < (Lx / 4.)**2)
>>> var.calcDistanceFunction()
>>> viewer = MayaviSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

```
>>> var = DistanceVariable(mesh = mesh, value = -1)
```

```
>>> var.setValue(1, where=(y > 2. * Ly / 3.) | ((x > Lx / 2.) & (y > Ly / 3.)) |
↳((y < Ly / 6.) & (x > Lx / 2)))
>>> var.calcDistanceFunction()
>>> viewer = MayaviSurfactantViewer(var)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

```
>>> viewer = MayaviSurfactantViewer(var, smooth = 2)
>>> viewer.plot()
>>> viewer._promptForOpinion()
>>> del viewer
```

Parameters

- **distanceVar** (*DistanceVariable*) –
- **levelSetValue** (*float*) – the value of the contour to be displayed
- **title** (*str*) – displayed at the top of the *Viewer* window
- **animate** (*bool*) – whether to show only the initial condition and the moving top boundary or to show all contours (Default)
- **limits** (*dict, optional*) – a (deprecated) alternative to limit keyword arguments
- **xmin** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.

- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.
- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

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** (*float, optional*) – 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.
- **xmax** (*float, optional*) – 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.
- **ymin** (*float, optional*) – 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.
- **ymax** (*float, optional*) – 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.
- **zmin** (*float, optional*) – 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.
- **zmax** (*float, optional*) – 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.

- **datamin** (*float, optional*) – 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.
- **datamax** (*float, optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

examples.levelSet.electroChem.metallonDiffusionEquation**examples.levelSet.electroChem.simpleTrenchSystem**

Model electrochemical superfill using the CEAC mechanism.

This input file is a demonstration of the use of *FiPy* for modeling electrodeposition using the CEAC mechanism. The material properties and experimental parameters used are roughly those that have been previously published [29].

To run this example from the base FiPy directory type:

```
$ python examples/levelSet/electroChem/simpleTrenchSystem.py
```

at the command line. The results of the simulation will be displayed and the word *finished* in the terminal at the end of the simulation. To run with a different number of time steps change the `numberOfSteps` argument as follows,

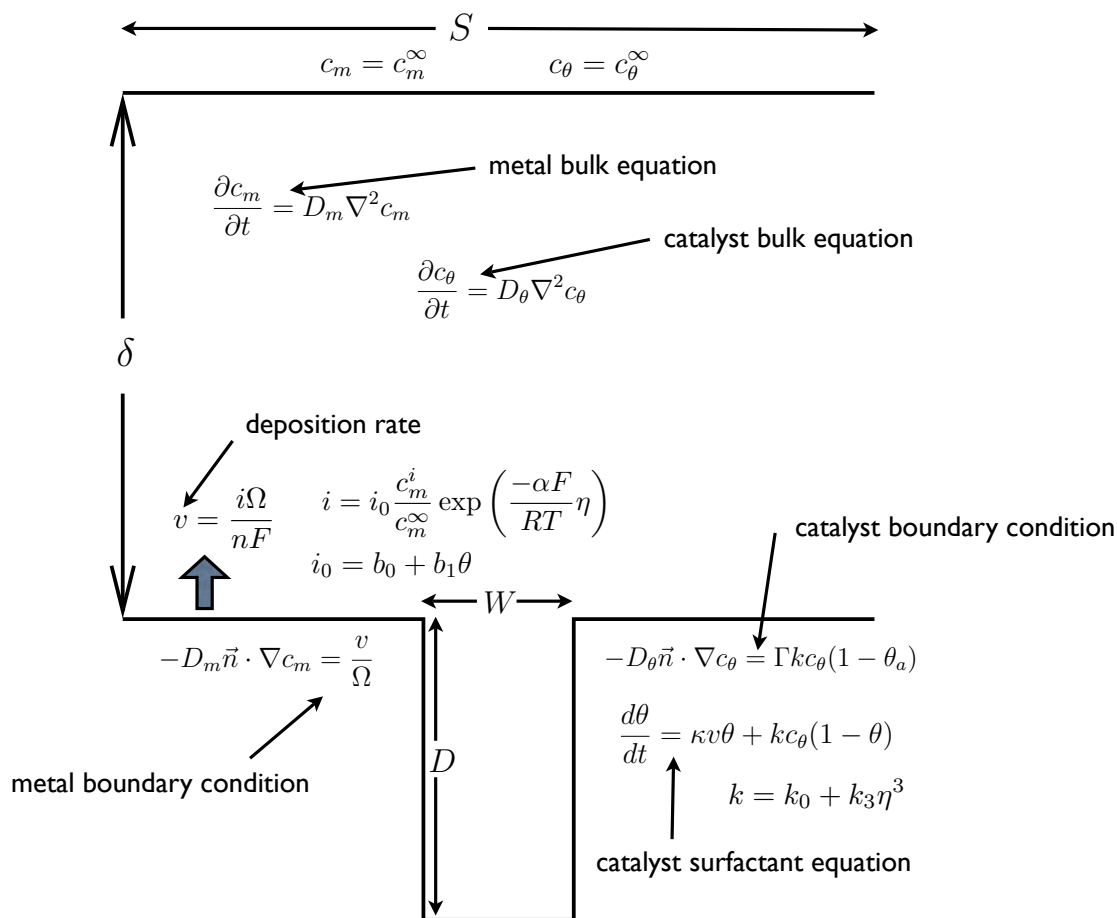
```
>>> runSimpleTrenchSystem(numberOfSteps=2, displayViewers=False)
1
```

Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. Example [examples.levelSet.electroChem.simpleTrenchSystem](#) gives explanation for writing new scripts or modifying existing scripts that are encapsulated by functions.

Any argument parameter can be changed. For example if the initial catalyst coverage is not 0, then it can be reset,

```
>>> runSimpleTrenchSystem(numberOfSteps=2, catalystCoverage=0.1, displayViewers=False)
0
```

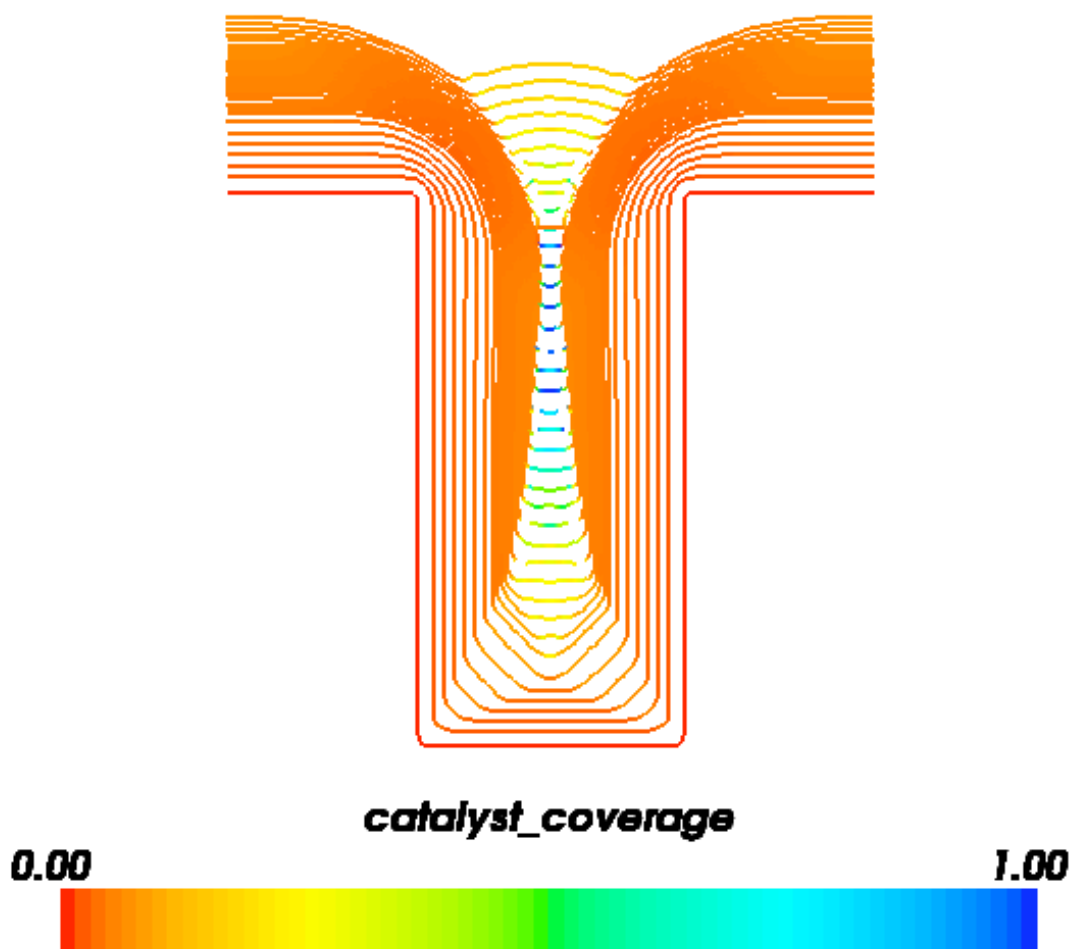
The following image shows a schematic of a trench geometry along with the governing equations for modeling electrodeposition with the CEAC mechanism. All of the given equations are implemented in the `examples.levelSet.electroChem.simpleTrenchSystem.runSimpleTrenchSystem()` function. As stated above, all the parameters in the equations can be changed with function arguments.



The following table shows the symbols used in the governing equations and their corresponding arguments to the `runSimpleTrenchSystem()` function. The boundary layer depth is intentionally small in this example in order not to complicate the mesh. Further examples will simulate more realistic boundary layer depths but will also have more complex meshes requiring the **gmsh** software.

Symbol	Description	Keyword Argument	Value	Unit
Deposition Rate Parameters				
v	deposition rate			m s^{-1}
i	current density			A m^{-2}
Ω	molar volume	molarVolume	7.1×10^{-6}	$\text{m}^3 \text{mol}^{-1}$
n	ion charge	charge	2	
F	Faraday's constant	faradaysConstant	9.6×10^{-4}	C mol^{-1}
i_0	exchange current density			A m^{-2}
α	transfer coefficient	transferCoefficient	0.5	
η	overpotential	overpotential	-0.3	V
R	gas constant	gasConstant	8.314	$\text{J K}^{-1} \text{mol}^{-1}$
T	temperature	temperature	298.0	K
b_0	current density for θ^0	currentDensity0	0.26	A m^{-2}
b_1	current density for θ	currentDensity1	45.0	A m^{-2}
Metal Ion Parameters				
c_m	metal ion concentration	metalConcentration	250.0	mol m^{-3}
c_m^∞	far field metal ion concentration	metalConcentration	250.0	mol m^{-3}
D_m	metal ion diffusion coefficient	metalDiffusion	5.6×10^{-10}	$\text{m}^2 \text{s}^{-1}$
Catalyst Parameters				
θ	catalyst surfactant concentration	catalystCoverage	0.0	
c_θ	bulk catalyst concentration	catalystConcentration	5.0×10^{-3}	mol m^{-3}
c_θ^∞	far field catalyst concentration	catalystConcentration	5.0×10^{-3}	mol m^{-3}
D_θ	catalyst diffusion coefficient	catalystDiffusion	1.0×10^{-9}	$\text{m}^2 \text{s}^{-1}$
Γ	catalyst site density	siteDensity	9.8×10^{-6}	mol m^{-2}
k	rate constant			$\text{m}^3 \text{mol}^{-1} \text{s}^{-1}$
k_0	rate constant for η^0	rateConstant0	1.76	$\text{m}^3 \text{mol}^{-1} \text{s}^{-1}$
k_3	rate constant for η^3	rateConstant3	-245.0×10^{-6}	$\text{m}^3 \text{mol}^{-1} \text{s}^{-1} \text{V}^{-3}$
Geometry Parameters				
D	trench depth	trenchDepth	0.5×10^{-6}	m
D/W	trench aspect ratio	aspectRatio	2.0	
S	trench spacing	trenchSpacing	0.6×10^{-6}	m
δ	boundary layer depth	boundaryLayerDepth	0.3×10^{-6}	m
Simulation Control Parameters				
	computational cell size	cellSize	0.1×10^{-7}	m
	number of time steps	numberOfSteps	5	
	whether to display the viewers	displayViewers	True	

If the Mayavi plotting software is installed (see [Installation](#)) then a plot should appear that is updated every 20 time steps and will eventually resemble the image below.



`examples.levelSet.electroChem.surfactantBulkDiffusionEquation`

`examples.levelSet.electroChem.test`

`examples.levelSet.electroChem.trenchMesh`

24.8.4 `examples.levelSet.surfactant`

Modules

<code>examples.levelSet.surfactant.circle</code>	This example first imposes a circular distance function:
<code>examples.levelSet.surfactant.expandingCircle</code>	This example represents an expanding circular interface with an initial coverage of surfactant.
<code>examples.levelSet.surfactant.square</code>	This example advects a 2 by 2 initially square region outwards.
<code>examples.levelSet.surfactant.test</code>	

examples.levelSet.surfactant.circle

This example first imposes a circular distance function:

$$\phi(x, y) = \left[\left(x - \frac{L}{2} \right)^2 + \left(y - \frac{L}{2} \right)^2 \right]^{1/2} - \frac{L}{4}$$

then the variable is advected with,

$$\frac{\partial \phi}{\partial t} + \vec{u} \cdot \nabla \phi = 0$$

Also a surfactant is present of the interface, governed by the equation:

$$\frac{d\theta}{dt} = Jv\theta$$

The result can be tested with the following code:

```
>>> surfactantBefore = numerix.sum(surfactantVariable * mesh.cellVolumes)
>>> for step in range(steps):
...     distanceVariable.updateOld()
...     surfactantEquation.solve(surfactantVariable, dt=1.)
...     advectionEquation.solve(distanceVariable, dt = timeStepDuration)
>>> surfactantEquation.solve(surfactantVariable, dt=1.)
>>> surfactantAfter = numerix.sum(surfactantVariable * mesh.cellVolumes)
>>> print(surfactantBefore, allclose(surfactantAfter))
1
>>> areas = (distanceVariable.cellInterfaceAreas < 1e-6) * 1e+10 + distanceVariable.
↳ cellInterfaceAreas
>>> answer = initialSurfactantValue * initialRadius / (initialRadius + distanceToTravel)
>>> coverage = surfactantVariable * mesh.cellVolumes / areas
>>> error = (coverage / answer - 1)**2 * (coverage > 1e-3)
>>> print(numerix.sqrt(numerix.sum(error) / numerix.sum(error > 0)))
0.00813776069241
```

examples.levelSet.surfactant.expandingCircle

This example represents an expanding circular interface with an initial coverage of surfactant. The rate of expansion is dependent on the coverage of surfactant, The governing equations are given by:

$$\begin{aligned} \dot{\theta} &= -\frac{\dot{r}}{r}\theta \\ \dot{r} &= k\theta \end{aligned}$$

The solution for these set of equations is given by:

$$\begin{aligned} r &= \sqrt{2kr_0\theta_0t + r_0^2} \\ \theta &= \frac{r_0\theta_0}{\sqrt{2kr_0\theta_0t + r_0^2}} \end{aligned}$$

The following tests can be performed. First test for global conservation of surfactant:

```

>>> surfactantBefore = numerix.sum(surfactantVariable * mesh.cellVolumes)
>>> totalTime = 0
>>> steps = 5
>>> for step in range(steps):
...     velocity.setValue(surfactantVariable.interfaceVar * k)
...     distanceVariable.extendVariable(velocity)
...     timeStepDuration = cfl * dx / velocity.max()
...     distanceVariable.updateOld()
...     advectionEquation.solve(distanceVariable, dt = timeStepDuration)
...     surfactantEquation.solve(surfactantVariable, dt=1)
...     totalTime += timeStepDuration
>>> surfactantEquation.solve(surfactantVariable, dt=1)
>>> surfactantAfter = numerix.sum(surfactantVariable * mesh.cellVolumes)
>>> print(surfactantBefore.allclose(surfactantAfter))
1

```

Next test for the correct local value of surfactant:

```

>>> finalRadius = numerix.sqrt(2 * k * initialRadius * initialSurfactantValue *
↳totalTime + initialRadius**2)
>>> answer = initialSurfactantValue * initialRadius / finalRadius
>>> coverage = surfactantVariable.interfaceVar
>>> error = (coverage / answer - 1)**2 * (coverage > 1e-3)
>>> print(numerix.sqrt(numerix.sum(error) / numerix.sum(error > 0)) < 0.04)
1

```

Test for the correct position of the interface:

```

>>> x, y = mesh.cellCenters
>>> radius = numerix.sqrt((x - L / 2)**2 + (y - L / 2)**2)
>>> solution = radius - distanceVariable
>>> error = (solution / finalRadius - 1)**2 * (coverage > 1e-3)
>>> print(numerix.sqrt(numerix.sum(error) / numerix.sum(error > 0)) < 0.02)
1

```

examples.levelSet.surfactant.square

This example advects a 2 by 2 initially square region outwards. The example checks for global conservation of surfactant.

Advect the interface and check the position.

```

>>> distanceVariable.calcDistanceFunction()
>>> initialSurfactant = numerix.sum(surfactantVariable)
>>> for step in range(steps):
...     distanceVariable.updateOld()
...     surfactantEquation.solve(surfactantVariable, dt=1)
...     advectionEquation.solve(distanceVariable, dt = timeStepDuration)
>>> print(numerix.allclose(initialSurfactant, numerix.sum(surfactantVariable)))
1

```

`examples.levelSet.surfactant.test`

24.8.5 examples.levelSet.test

Run all the test cases in examples/

24.9 examples.meshing

Modules

<code>examples.meshing.gmshRefinement</code>	
<code>examples.meshing.inputGrid2D</code>	To run this example from the base FiPy directory, type.
<code>examples.meshing.sphere</code>	An interesting problem is to solve an equation on a 2D geometry that is embedded in 3D space, such as diffusion on the surface of a sphere (with nothing either inside or outside the sphere).
<code>examples.meshing.test</code>	Run all the test cases in examples/meshing/

24.9.1 examples.meshing.gmshRefinement

24.9.2 examples.meshing.inputGrid2D

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

```
$ python examples/meshing/inputGrid2D.py --numberOfElements=X
```

This example demonstrates how to build a 1D mesh and obtain basic mesh information. The command line argument, X, controls the number of elements on the mesh. Firstly parse the command line argument for *numberOfElements*, with the default set at 100.

```
>>> from fipy.tools.parser import parse
>>> numberOfElements = parse('--numberOfElements', action = 'store', type = 'int',
↪ default = 100)
```

A *Grid2D* object is invoked in the following way,

```
>>> from fipy import Grid2D, CellVariable, Viewer
>>> from fipy.tools import numerix
```

```
>>> nx = int(numerix.sqrt(numberOfElements))
>>> ny = nx
>>> dx = 1.
>>> dy = 1.
>>> mesh = Grid2D(nx = nx, ny = nx, dx = dx, dy = dy)
```

Once the mesh has been built information about the mesh can be obtained. For example the mesh volumes can be obtained with the *getCellVolumes()* method.

```
>>> vols = mesh.cellVolumes
>>> numerix.allclose(dx * dy * numerix.ones(nx * ny), vols)
1
```

Obtain the number of cells in the mesh

```
>>> N = mesh.numberOfCells
>>> numerix.allclose(N, numberOfElements)
1
```

Obtain all the left exterior faces, this is equal to ny .

```
>>> faces = mesh.facesLeft
>>> len(faces) == ny
1
```

One can view the mesh with the following code,

```
>>> if __name__ == '__main__':
...     viewer = Viewer(CellVariable(value = 0, mesh = mesh))
...     viewer.plot()
```

24.9.3 examples.meshing.sphere

An interesting problem is to solve an equation on a 2D geometry that is embedded in 3D space, such as diffusion on the surface of a sphere (with nothing either inside or outside the sphere). This example demonstrates how to create the required mesh.

```
>>> from fipy import Gmsh2DIn3DSpace, CellVariable, MayaviClient
>>> from fipy.tools import numerix

>>> 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}}};
...     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]}}};
```

(continues on next page)

(continued from previous page)

```

...     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]};
...     '').extrude(extrudeFunc=lambda r: 1.1 * r)

```

```
>>> x, y, z = mesh.cellCenters
```

```
>>> var = CellVariable(mesh=mesh, value=x * y * z, name="x*y*z")
```

```

>>> if __name__ == '__main__':
...     viewer = MayaviClient(vars=var)
...     viewer.plot()

```

```

>>> max(numerix.sqrt(x**2 + y**2 + z**2)) < 5.3
True
>>> min(numerix.sqrt(x**2 + y**2 + z**2)) > 5.2
True

```

24.9.4 examples.meshing.test

Run all the test cases in examples/meshing/

24.10 examples.parallel

24.11 examples.phase

Modules

<code>examples.phase.anisotropy</code>	Solve a dendritic solidification problem.
<code>examples.phase.anisotropyOLD</code>	
<code>examples.phase.binary</code>	It is straightforward to extend a phase field model to include binary alloys.
<code>examples.phase.binaryCoupled</code>	Simultaneously solve a phase-field evolution and solute diffusion problem in one-dimension.
<code>examples.phase.impingement</code>	
<code>examples.phase.missOrientation</code>	
<code>examples.phase.phaseViewer</code>	1D Viewer that calculates and plots the position of the phase field interface.
<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.
<code>examples.phase.quaternary</code>	Solve a phase-field evolution and diffusion of four species in one-dimension.
<code>examples.phase.simple</code>	Solve a phase-field (Allen-Cahn) problem in one-dimension.
<code>examples.phase.symmetry</code>	This example creates four symmetric quadrilateral regions in a box.
<code>examples.phase.test</code>	

24.11.1 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 [13] 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):
...             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,))
```

(continues on next page)

(continued from previous page)

```

...     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)))

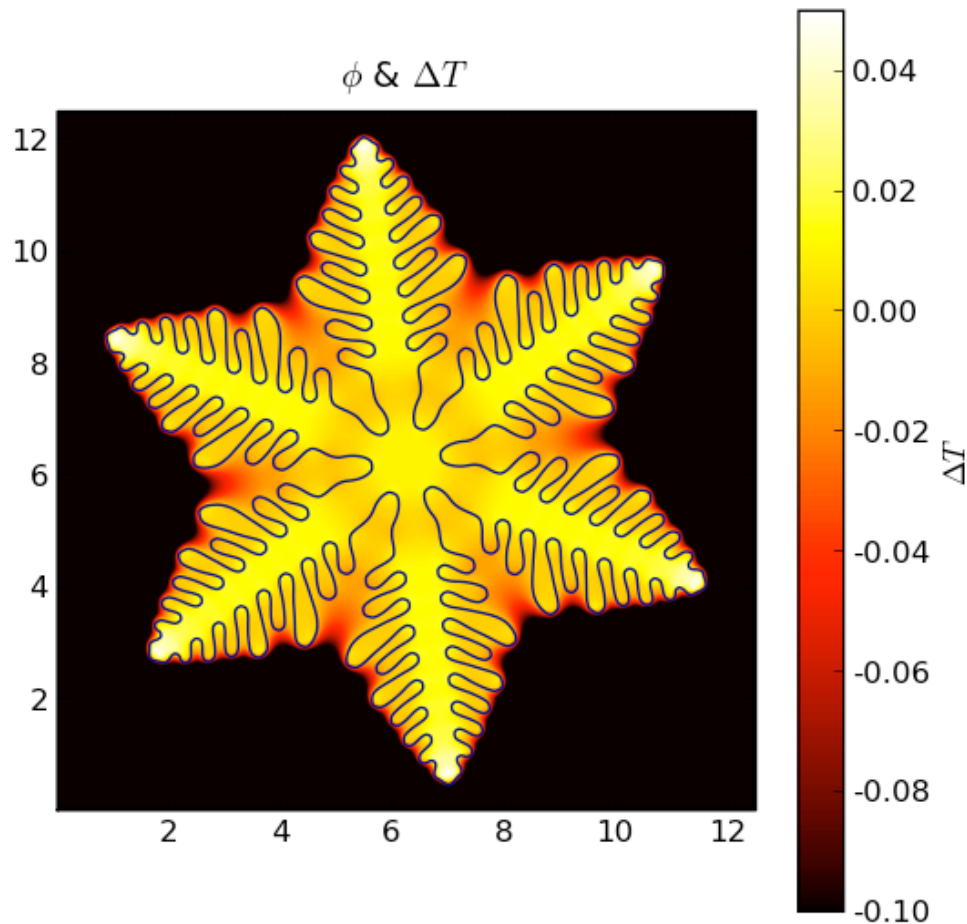
```

and iterate the solution in time, plotting as we go,

```

>>> if __name__ == '__main__':
...     steps = 10000
...     else:
...         steps = 10
>>> 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 [13].

24.11.2 examples.phase.anisotropyOLD

Attention: This example remains only for exact comparison against Ryo Kobayashi's FORTRAN code. See [examples.phase.anisotropy](#) for a better, although not numerically identical implementation.

In this example we solve a coupled phase and temperature equation to model solidification, and eventually dendritic growth, based on the work of Warren, Kobayashi, Lobkovsky and Carter [13].

We start from a circular seed in a 2D mesh:

```
>>> from fipy import CellVariable, Grid2D, TransientTerm, DiffusionTerm, \
↳ ExplicitDiffusionTerm, ImplicitSourceTerm, Viewer
>>> from fipy.tools import numerix
```

```
>>> numberOfCells = 40
>>> Length = numberOfCells * 2.5 / 100.
>>> nx = numberOfCells
>>> ny = numberOfCells
>>> dx = Length / nx
>>> dy = Length / ny
>>> radius = Length / 4.
>>> seedCenter = (Length / 2., Length / 2.)
>>> initialTemperature = -0.4
>>> mesh = Grid2D(dx=dx, dy=dy, nx=nx, ny=ny)
```

Dendritic growth will not be observed with this small test system. If you wish to see dendritic growth reset the following parameters such that `numberOfCells = 500`, `steps = 10000`, `radius = dx * 5`, `seedCenter = (0., 0.)` and `initialTemperature = -0.5`.

The governing equation for the phase field is given by:

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot [D \nabla \phi + A \nabla \xi] + \phi(1 - \phi)m(\phi, T)$$

where

$$m(\phi, T) = \phi - \frac{1}{2} - \frac{\kappa_1}{\pi} \arctan(\kappa_2 T).$$

The coefficients D and A are given by,

$$D = \alpha^2 [1 + c\beta]^2$$

and

$$A = \alpha^2 c [1 + c\beta] \beta_\psi$$

where $\beta = \frac{1-\Phi^2}{1+\Phi^2}$, $\Phi = \tan\left(\frac{N}{2}\psi\right)$, $\psi = \theta + \arctan\left(\frac{\phi_y}{\phi_x}\right)$ and $\xi_x = -\phi_y$ and $\xi_y = \phi_x$.

The governing equation for temperature is given by:

$$\frac{\partial T}{\partial t} = D_T \nabla^2 T + \frac{\partial \phi}{\partial t}$$

Here the phase and temperature equations are solved with an explicit and implicit technique, respectively.

The parameters for these equations are

```
>>> timeStepDuration = 5e-5
>>> tau = 3e-4
>>> alpha = 0.015
>>> c = 0.02
>>> N = 4.
>>> kappa1 = 0.9
>>> kappa2 = 20.
>>> tempDiffusionCoeff = 2.25
>>> theta = 0.
```

The phase variable is 0 for a liquid and 1 for a solid. Here, the phase variable is initialized as a liquid,

```
>>> phase = CellVariable(name='phase field', mesh=mesh, hasOld=1)
```

The hasOld flag keeps the old value of the variable. This is necessary for a transient solution. In this example we wish to set up an interior region that is solid. The domain is seeded with a circular solidified region with parameters seedCenter and radius representing the center and radius of the seed.

```
>>> x, y = mesh.cellCenters
>>> phase.setValue(1., where=((x - seedCenter[0])**2
...                          + (y - seedCenter[1])**2) < radius**2)
```

The temperature field is initialized to a value of -0.4 throughout:

```
>>> temperature = CellVariable(
...     name='temperature',
...     mesh=mesh,
...     value=initialTemperature,
...     hasOld=1)
```

The $m(\phi, T)$ variable

is created from the phase and temperature variables.

```
>>> mVar = phase - 0.5 - kappa1 / numerix.pi * numerix.arctan(kappa2 * temperature)
```

The following section of code builds up the A and D coefficients.

```
>>> phaseY = phase.faceGrad.dot((0, 1))
>>> phaseX = phase.faceGrad.dot((1, 0))
>>> psi = theta + numerix.arctan2(phaseY, phaseX)
>>> Phi = numerix.tan(N * psi / 2)
>>> PhiSq = Phi**2
>>> beta = (1. - PhiSq) / (1. + PhiSq)
>>> betaPsi = -N * 2 * Phi / (1 + PhiSq)
>>> A = alpha**2 * c * (1. + c * beta) * betaPsi
>>> D = alpha**2 * (1. + c * beta)**2
```

The $\nabla \xi$ variable (dxi), given by $(\xi_x, \xi_y) = (-\phi_y, \phi_x)$, is constructed by first obtaining $\nabla \phi$ using `faceGrad`. The axes are rotated ninety degrees.

```
>>> dxi = phase.faceGrad.dot(((0, 1), (-1, 0)))
>>> anisotropySource = (A * dxi).divergence
```

The phase equation can now be constructed.

```
>>> phaseEq = TransientTerm(tau) == ExplicitDiffusionTerm(D) + \
...     ImplicitSourceTerm(mVar * ((mVar < 0) - phase)) + \
...     ((mVar > 0.) * mVar * phase + anisotropySource)
```

The temperature equation is built in the following way,

```
>>> temperatureEq = TransientTerm() == \
...     DiffusionTerm(tempDiffusionCoeff) + \
...     (phase - phase.old) / timeStepDuration
```

If we are running this example interactively, we create viewers for the phase and temperature fields


```
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase)
...     temperatureViewer = Viewer(vars=temperature,
...                               datamin=-0.5, datamax=0.5)
...     phaseViewer.plot()
...     temperatureViewer.plot()
```

we iterate the solution in time, plotting as we go if running interactively,

```
>>> from fipy import solver_suite
>>> if solver_suite in ["trilinos"]:
...     from fipy import MultilevelNSSAPreconditioner
...     preconditioner = MultilevelNSSAPreconditioner()
... else:
...     preconditioner = "default"
>>> from fipy import DefaultAsymmetricSolver
>>> solver = DefaultAsymmetricSolver(precon=preconditioner)
```

```
>>> steps = 10
>>> for i in range(steps):
...     phase.updateOld()
...     temperature.updateOld()
...     phaseEq.solve(phase, dt=timeStepDuration, solver=solver)
...     temperatureEq.solve(temperature, dt=timeStepDuration, solver=solver)
...     if i%10 == 0 and __name__ == '__main__':
...         phaseViewer.plot()
...         temperatureViewer.plot()
```

The solution is compared with test data. The test data was created for `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `anisotropy.gz` extracts the data and compares it with the `phase` variable.

```
>>> import os
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + '.gz')
>>> print(phase.allclose(testData))
1
```

24.11.3 examples.phase.binary

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, PowerLawConvectionTerm, DefaultAsymmetricSolver, 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 [1] [3] [30]

$$\mathcal{F}(\phi, C, T) = \int_{\mathcal{V}} \left\{ f(\phi, C, T) + \frac{\kappa_{\phi}}{2} |\nabla \phi|^2 + \frac{\kappa_C}{2} |\nabla C|^2 \right\} dV$$

over the volume $\mathcal{V}$ as a function of phase ϕ ¹

```
>>> 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

$$\frac{\partial \phi}{\partial t} = M_{\phi} \left(\kappa_{\phi} \nabla^2 \phi - \frac{\partial f}{\partial \phi} \right) \quad (24.7)$$

for phase and

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(M_C \nabla \frac{\partial f}{\partial C} \right) \quad (24.8)$$

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

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.

² we are going to want to examine different temperatures in this example, so we declare T as a *Variable*

```
>>> 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

$$f(\phi, C, T) = p(\phi)f^S(C, T) + (1 - p(\phi))f^L(C, T) + \left[(1 - C)\frac{W_A}{2} + C\frac{W_B}{2}\right]g(\phi).$$

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

$$\begin{aligned} \frac{\partial f}{\partial \phi} &= (1 - C) \frac{\partial f_A}{\partial \phi} + C \frac{\partial f_B}{\partial \phi} \\ &= \left\{ (1 - C) \frac{L_A (T - T_M^A)}{T_M^A} + C \frac{L_B (T - T_M^B)}{T_M^B} \right\} p'(\phi) + \left\{ (1 - C) \frac{W_A}{2} + C \frac{W_B}{2} \right\} g'(\phi) \end{aligned}$$

and

$$\begin{aligned} \frac{\partial f}{\partial C} &= \left[f_B(\phi, T) + \frac{RT}{V_m} \ln C \right] - \left[f_A(\phi, T) + \frac{RT}{V_m} \ln(1 - C) \right] \\ &= [\mu_B(\phi, C, T) - \mu_A(\phi, C, T)] / V_m \end{aligned}$$

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) == DiffusionTerm(coeff=kappa)
...             + S0 + ImplicitSourceTerm(coeff=S1))
```

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
```

(continues on next page)

(continued from previous page)

```
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. (24.8) to the canonical form of Eq. (8.2), hence we must expand Eq. (24.9) 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. (24.8) 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 by $M_C \equiv DC(1 - C)V_m/RT$ and write Eq. (24.8) as

$$\begin{aligned} \frac{\partial C}{\partial t} &= \nabla \cdot (D \nabla C) \\ &+ \nabla \cdot \left(\frac{DC(1 - C)V_m}{RT} \left\{ \left[\frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] \nabla p(\phi) + \frac{W_B - W_A}{2} \nabla g(\phi) \right\} \right). \end{aligned}$$

The first term is clearly a *DiffusionTerm*. The second is less obvious, but by factoring out C , we can see that this is a *ConvectionTerm* with a velocity

$$\vec{u}_\phi = \frac{D(1 - C)V_m}{RT} \left\{ \left[\frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] \nabla p(\phi) + \frac{W_B - W_A}{2} \nabla g(\phi) \right\}$$

due to phase transformation, such that

$$\frac{\partial C}{\partial t} = \nabla \cdot (D \nabla C) + \nabla \cdot (C \vec{u}_\phi)$$

or

```
>>> D1 = Variable(value=1e-5) # cm**2 / s
>>> Ds = Variable(value=1e-9) # cm**2 / s
>>> D = (Ds - D1) * phase.arithmeticFaceValue + D1
```

```
>>> phaseTransformationVelocity = (((enthalpyB - enthalpyA) * p(phase).faceGrad
...                               + 0.5 * (WB - WA) * g(phase).faceGrad)
...                               * D * (1. - C).harmonicFaceValue * Vm / (R * T))
```

```
>>> diffusionEq = (TransientTerm()
...               == DiffusionTerm(coeff=D)
...               + PowerLawConvectionTerm(coeff=phaseTransformationVelocity))
```

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)
```

```
>>> elapsed = Variable(value=0.) # s
```

```
>>> if __name__ == '__main__':
...     try:
...         from phaseViewer import PhaseViewer
...
...         viewer = PhaseViewer(phase=phase, C=C, sharp=sharp,
...                               elapsed=elapsed,
...                               L=L, deltaA=deltaA,
...                               tmin=1e-5, tmax=300 * 3600,
...                               datamin=0., datamax=1.)
...     except ImportError:
...         viewer = Viewer(vars=(phase, C, sharp),
```

(continues on next page)

(continued from previous page)

```
...          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

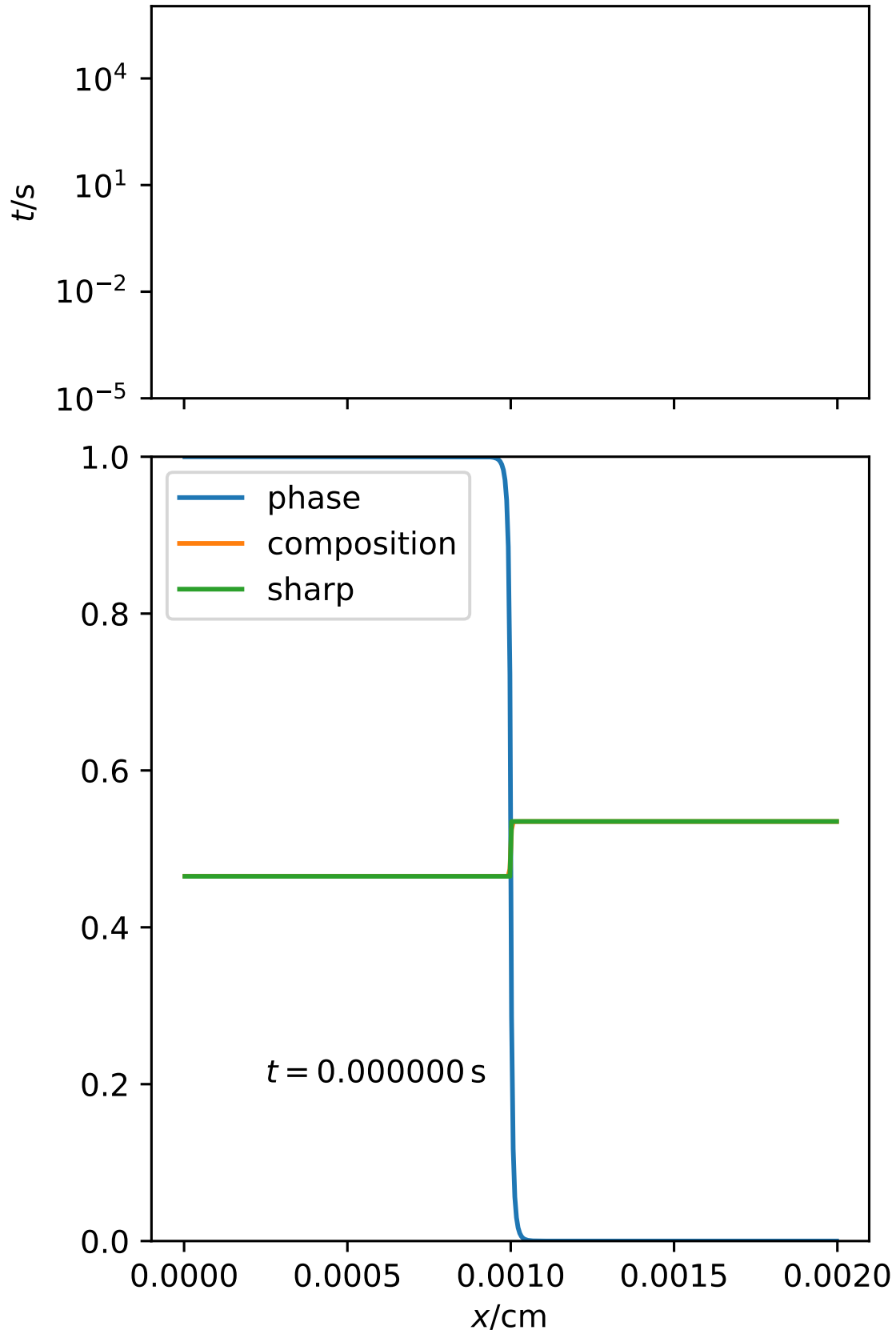
```
>>> dt = 1.e5
```

Because the phase field equation is coupled to the composition through enthalpy and $\bar{W}$ and the diffusion equation is coupled to the phase field through `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. Because of the `ConvectionTerm`, the solution matrix for `diffusionEq` is asymmetric, so we use a `DefaultAsymmetricSolver` for this equation.

We now use the “`sweep()`” method instead of “`solve()`” because we require the residual. The initial residual of the diffusion equation is much larger than the norm of the right-hand-side vector, so we use “`initial`” tolerance scaling.

```
>>> solver = DefaultAsymmetricSolver(criterion="initial", tolerance=1e-10)
```

```
>>> phase.updateOld()
>>> C.updateOld()
>>> phaseRes = 1e+10
>>> diffRes = 1e+10
>>> while phaseRes > 1e-8 or diffRes > 1e-8 or abs(Cavg.value - 0.5) > 5e-7:
...     phaseRes = phaseEq.sweep(var=phase, dt=dt)
...     diffRes = diffusionEq.sweep(var=C, dt=dt, solver=solver)
>>> 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=2e-4))
True
>>> print(Cl.allclose(C.faceValue[X.value==L], atol=2e-4))
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. (24.9). If we assume that the phase changes from 1 to 0 in a single grid spacing, that the diffusivity is D_l at the interface, and that the term due to the difference in barrier heights is negligible:

$$\begin{aligned}\vec{u}_\phi &= \frac{D_\phi}{C} \nabla \phi \\ &\approx \frac{D_l \frac{1}{2} V_m}{RT} \left[\frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] \frac{1}{\Delta x} \\ &\approx \frac{D_l \frac{1}{2} V_m}{RT} (L_B + L_A) \frac{T_M^A - T_M^B}{T_M^A + T_M^B} \frac{1}{\Delta x} \\ &\approx 0.28 \text{ cm/s}\end{aligned}$$

To get a $\text{CFL} = \vec{u}_\phi \Delta t / \Delta x < 1$, we need a time step of about 10^{-5} s.

```
>>> dt0 = 1.e-5
```

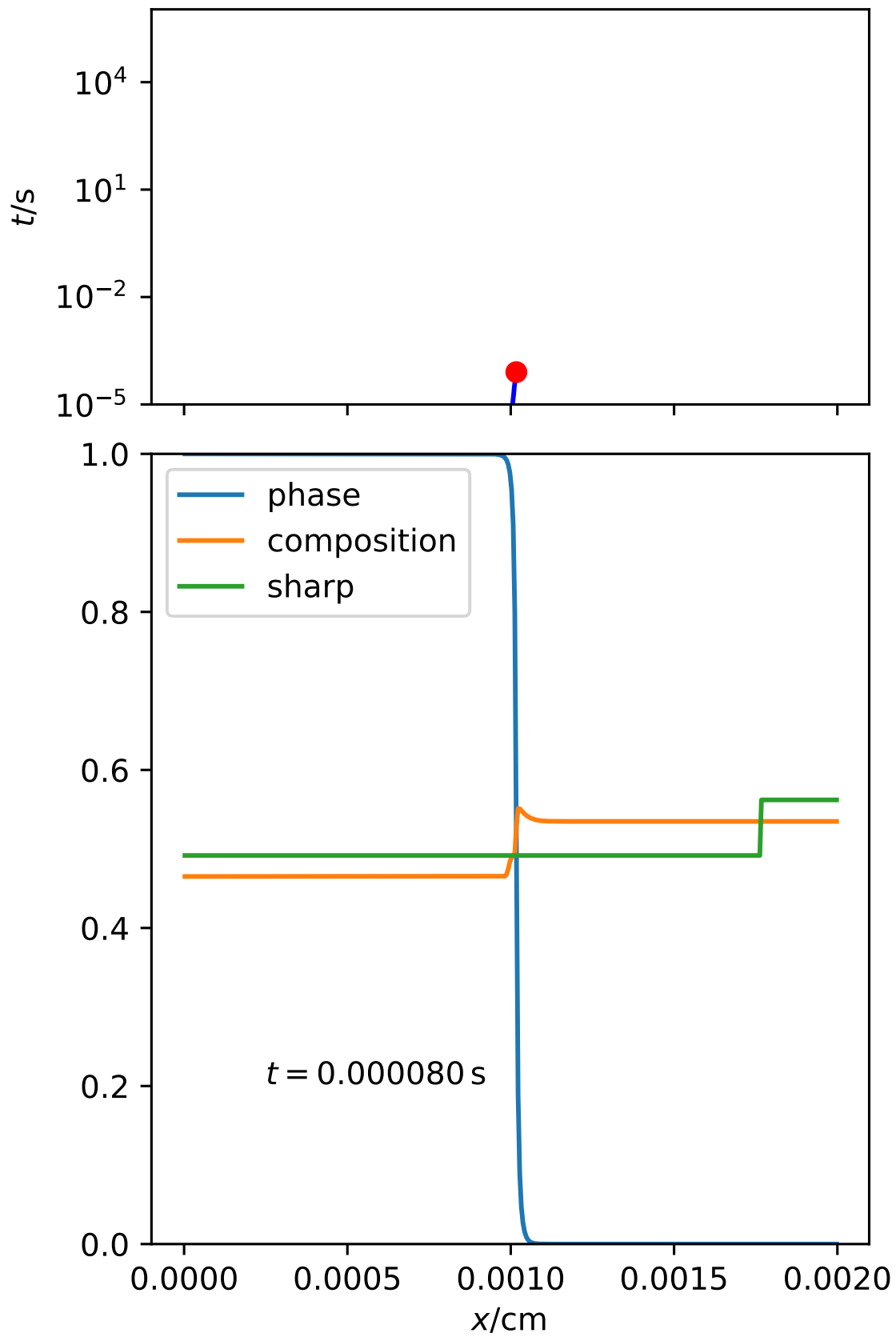
```
>>> for i in range(8):
...     phase.updateOld()
...     C.updateOld()
...     phaseRes = 1e+10
...     diffRes = 1e+10
...     while phaseRes > 5e-8 or diffRes > 1e-14 or abs(Cavg.value - 0.5) > 5e-7:
...         phaseRes = phaseEq.sweep(var=phase, dt=dt0)
```

(continues on next page)

(continued from previous page)

```
...     diffRes = diffusionEq.sweep(var=C, dt=dt0, solver=solver)
...     elapsed.value = (i + 1) * dt0
...     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 approaches 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.

We can estimate the time to equilibration by examining the time for the diffusion field to become uniform. In the liquid, this will take $\mathcal{O}((10\text{ }\mu\text{m})^2/D_l) = 0.1\text{ s}$ and in the solid $\mathcal{O}((10\text{ }\mu\text{m})^2/D_s) = 1000\text{ s}$.

Not wanting to take a hundred-million steps, we employ adaptive time stepping, using the [steppyngstounes](#) package. This package takes care of many of the messy details of stepping, like overshoot, underflow, and step size adaptation, while keeping the structure of our solve loop largely intact.

```
>>> from steppyngstounes import SequenceStepper, PIDStepper
>>> from itertools import count
```

Assuming the process is dominated by diffusion, we can take steps that increase geometrically. Since we're unsure if diffusion is the only process controlling dynamics, we take each increasing step with an adaptive stepper that uses a [PID controller](#) to keep the equation residuals and mass conservation within acceptable limits. The total number of solves is not strongly sensitive to the number of sweeps, but two sweeps seems to be both sufficient and efficient.

We'll only advance the step if it's successful, so we need to update the old values before we get started.

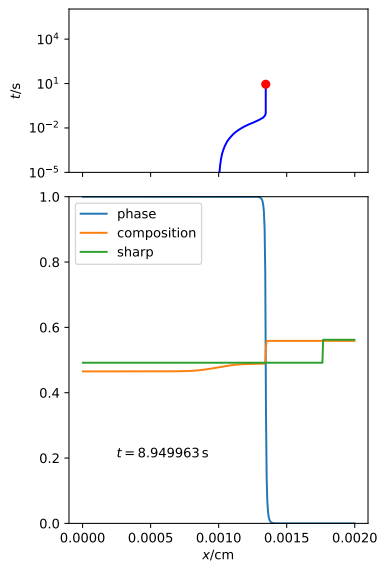
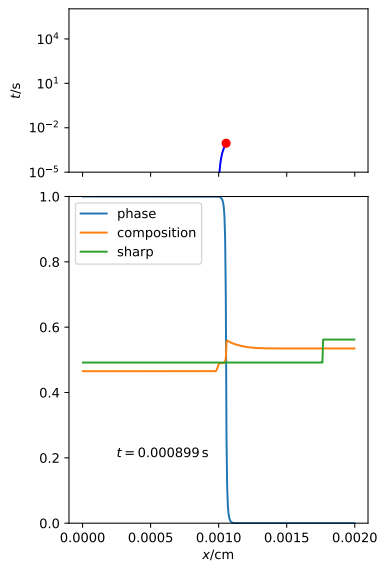
```
>>> phase.updateOld()
>>> C.updateOld()
```

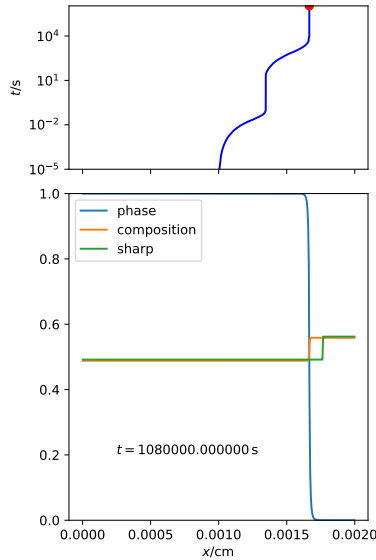
```
>>> if __name__ == '__main__':
...     totaltime = 300 * 3600 # 300 h
... else:
...     totaltime = 32e-5 # 320 us
```

```
>>> dt = dt0
```

```
>>> for checkpoint in SequenceStepper(start=float(elapsed), stop=totaltime,
...                                   sizes=(dt0 * 2**(n/2) for n in count(7))):
...     for step in PIDStepper(start=checkpoint.begin,
...                             stop=checkpoint.end,
...                             size=dt):
...         for sweep in range(2):
...             phaseRes = phaseEq.sweep(var=phase, dt=step.size)
...             diffRes = diffusionEq.sweep(var=C, dt=step.size, solver=solver)
...             err = max(phaseRes / 1e-3,
...                       diffRes / 1e-3,
...                       abs(Cavg.value - 0.5) / 5e-7)
...             if step.succeeded(error=err):
...                 phase.updateOld()
...                 C.updateOld()
...                 elapsed.value = step.end
...             else:
...                 phase.value = phase.old
...                 C.value = C.old
...         # the last step might have been smaller than possible,
...         # if it was near the end of the checkpoint range
...         dt = step.want
...         _ = checkpoint.succeeded()
...         if __name__ == '__main__':
...             viewer.plot()
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Re-equilibrated phase field. Press <return> to proceed...")
```





The interface moves $\approx 3.4 \mu\text{m}$ in 80 ms, driven by diffusion in the liquid phase (compare the estimate above of 0.1 s). For the next 20 s, the interface stalls while the solute step trapped in the solid phase diffuses outward ($(3.4 \mu\text{m})^2/D_s = \mathcal{O}(100 \text{ s})$). Once the solute gradient in the solid reaches the new position of the interface, the solidification front begins to move, driven by diffusion in the solid. When the solute in the solid becomes uniform, the interface stalls again after $\approx 4000 \text{ s}$, having moved another $3.2 \mu\text{m}$ (recall the estimate of 1000 s for equilibration in the solid). After this point, there is essentially no further motion of the interface and barely perceptible changes in the concentration field. The fact that the interface does not reach the predicted phase fraction is due to the fact that this phase field model accounts for adsorption at the interface, resulting in the bulk phases not having exactly the concentrations assumed in the sharp interface treatment.

24.11.4 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, DefaultAsymmetricSolver
>>> 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 [1] [3] [30]

$$\mathcal{F}(\phi, C, T) = \int_{\mathcal{V}} \left\{ f(\phi, C, T) + \frac{\kappa_{\phi}}{2} |\nabla \phi|^2 + \frac{\kappa_C}{2} |\nabla C|^2 \right\} dV$$

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

$$\frac{\partial \phi}{\partial t} = M_\phi \left(\kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \right) \quad (24.9)$$

for phase and

$$\frac{\partial C}{\partial t} = \nabla \cdot \left(M_C \nabla \frac{\partial f}{\partial C} \right) \quad (24.10)$$

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

$$\begin{aligned} \frac{\partial f}{\partial \phi} &= (1 - C) \frac{\partial f_A}{\partial \phi} + C \frac{\partial f_B}{\partial \phi} \\ &= \left\{ (1 - C) \frac{L_A (T - T_M^A)}{T_M^A} + C \frac{L_B (T - T_M^B)}{T_M^B} \right\} p'(\phi) + \left\{ (1 - C) \frac{W_A}{2} + C \frac{W_B}{2} \right\} g'(\phi) \end{aligned}$$

and

$$\begin{aligned} \frac{\partial f}{\partial C} &= \left[f_B(\phi, T) + \frac{RT}{V_m} \ln C \right] - \left[f_A(\phi, T) + \frac{RT}{V_m} \ln(1 - C) \right] \\ &= [\mu_B(\phi, C, T) - \mu_A(\phi, C, T)] / V_m \end{aligned}$$

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. (24.10) to the canonical form of Eq. (8.2), hence we must expand Eq. (24.11) 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. (24.10) 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. (24.10) 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)
```

```
>>> elapsed = Variable(value=0.) # s
```

```
>>> if __name__ == '__main__':
...     try:
...         from phaseViewer import PhaseViewer
...
...         viewer = PhaseViewer(phase=phase, C=C, sharp=sharp,
...                               elapsed=elapsed,
...                               L=L, deltaA=deltaA,
...                               tmin=1e-5, tmax=300 * 3600,
...                               datamin=0., datamax=1.)
...     except Exception:
...         viewer = Viewer(vars=(phase, C, sharp),
```

(continues on next page)

(continued from previous page)

```
...          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 `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.

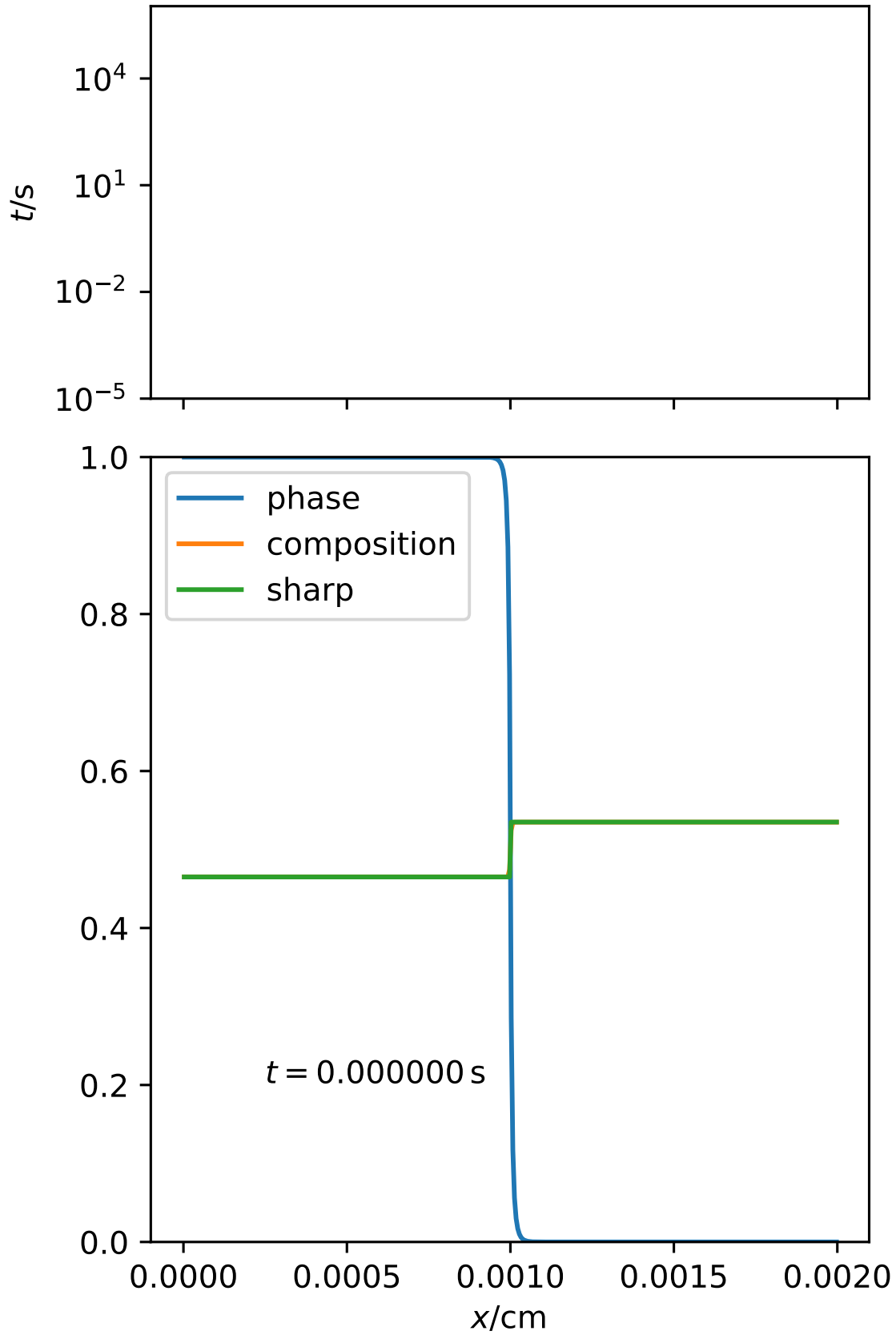
We now use the “`sweep()`” method instead of “`solve()`” because we require the residual.

```
>>> import fipy.solvers.solver
>>> if fipy.solvers.solver_suite == 'petsc':
...     from fipy import HYPREPreconditioner
...     solver = DefaultAsymmetricSolver(criterion="initial",
...                                     precon=HYPREPreconditioner(),
...                                     tolerance=1e-10)
... elif fipy.solvers.solver_suite in ['trilinos']:
...     from fipy import LinearBicgstabSolver
...     solver = LinearBicgstabSolver(criterion="initial",
...                                   tolerance=1e-10)
... else:
...     solver = DefaultAsymmetricSolver(criterion="initial",
...                                     precon=None,
...                                     tolerance=1e-10)
```

```
>>> phase.updateOld()
>>> C.updateOld()
>>> res = 1.
>>> initialRes = None
>>> sweep = 0
```

```
>>> while res > 1e-5 and sweep < 100:
...     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. (24.11) (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 D_l at the interface, and that the term due to the difference in barrier heights is negligible:

$$\begin{aligned}\vec{u}_\phi &= \frac{D_\phi}{C} \nabla \phi \\ &\approx \frac{D_l \frac{1}{2} V_m}{RT} \left[\frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] \frac{1}{\Delta x} \\ &\approx \frac{D_l \frac{1}{2} V_m}{RT} (L_B + L_A) \frac{T_M^A - T_M^B}{T_M^A + T_M^B} \frac{1}{\Delta x} \\ &\approx 0.28 \text{ cm/s}\end{aligned}$$

To get a $\text{CFL} = \vec{u}_\phi \Delta t / \Delta x < 1$, we need a time step of about 10^{-5} s.

```
>>> dt0 = 1.e-5
```

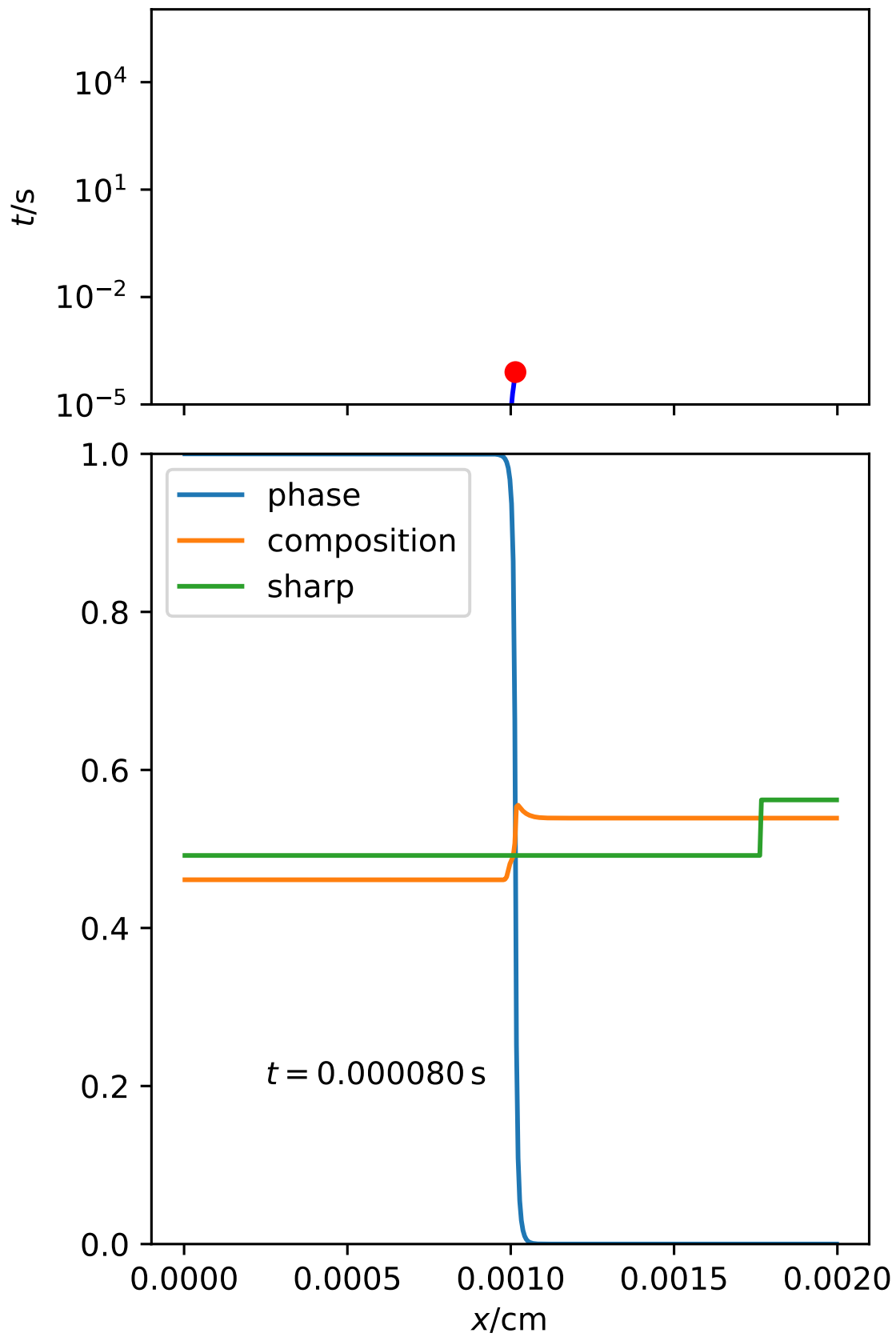
```
>>> for i in range(8):
...     phase.updateOld()
...     C.updateOld()
...     res = 1e+10
...     sweep = 0
...     while (res > 1e-3 or abs(Cavg.value - 0.5) > 2e-6) and sweep < 20:
```

(continues on next page)

(continued from previous page)

```
...     res = eq.sweep(dt=dt0, solver=solver)
...     sweep += 1
...     elapsed.value = (i + 1) * dt0
...     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 approaches 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.

We can estimate the time to equilibration by examining the time for the diffusion field to become uniform. In the liquid, this will take $\mathcal{O}((10\text{ }\mu\text{m})^2/D_l) = 0.1\text{ s}$ and in the solid $\mathcal{O}((10\text{ }\mu\text{m})^2/D_s) = 1000\text{ s}$.

Not wanting to take a hundred-million steps, we employ adaptive time stepping, using the [steppyngstounes](#) package. This package takes care of many of the messy details of stepping, like overshoot, underflow, and step size adaptation, while keeping the structure of our solve loop largely intact.

```
>>> from steppyngstounes import SequenceStepper, PIDStepper
>>> from itertools import count
```

Assuming the process is dominated by diffusion, we can take steps that increase geometrically. Since we're unsure if diffusion is the only process controlling dynamics, we take each increasing step with an adaptive stepper that uses a [PID controller](#) to keep the equation residuals and mass conservation within acceptable limits. The total number of solves is not strongly sensitive to the number of sweeps, but two sweeps seems to be both sufficient and efficient.

We'll only advance the step if it's successful, so we need to update the old values before we get started.

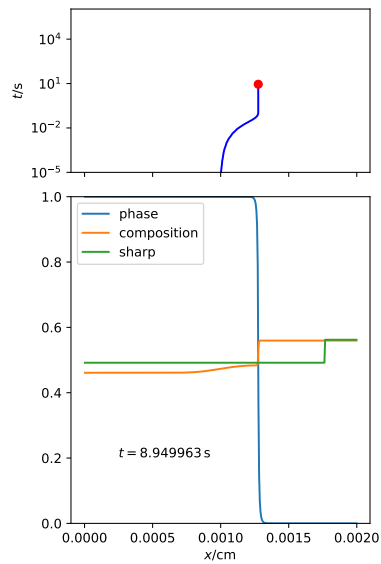
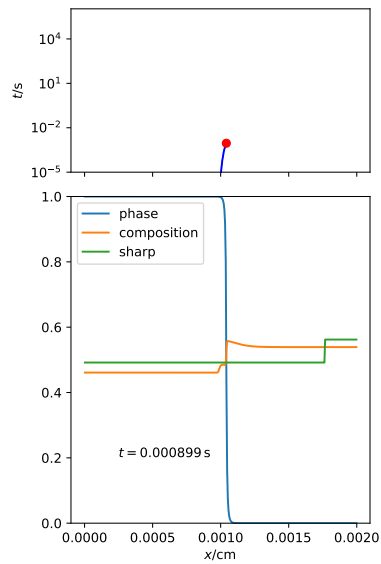
```
>>> phase.updateOld()
>>> C.updateOld()
```

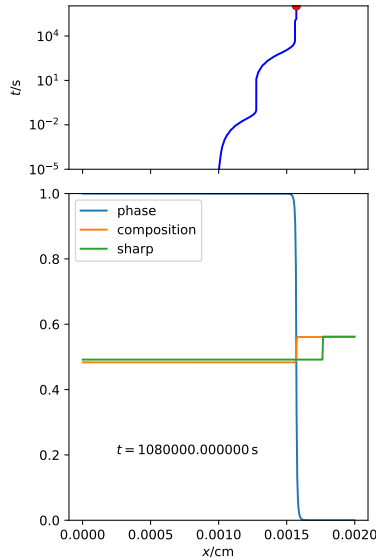
```
>>> if __name__ == '__main__':
...     totaltime = 300 * 3600 # 300 h
... else:
...     totaltime = 32e-5 # 320 us
```

```
>>> dt = dt0
```

```
>>> for checkpoint in SequenceStepper(start=float(elapsed), stop=totaltime,
...                                   sizes=(dt0 * 2**(n/2) for n in count(7))):
...     for step in PIDStepper(start=checkpoint.begin,
...                             stop=checkpoint.end,
...                             size=dt):
...         for sweep in range(2):
...             res = eq.sweep(dt=step.size, solver=solver)
...             err = max(res / 1e-3,
...                       abs(Cavg.value - 0.5) / 2e-6)
...             if step.succeeded(error=err):
...                 phase.updateOld()
...                 C.updateOld()
...                 elapsed.value = step.end
...             else:
...                 phase.value = phase.old
...                 C.value = C.old
...         # the last step might have been smaller than possible,
...         # if it was near the end of the checkpoint range
...         dt = step.want
...         _ = checkpoint.succeeded()
...         if __name__ == '__main__':
...             viewer.plot()
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Re-equilibrated phase field. Press <return> to proceed...")
```





The interface moves $\approx 2.8 \mu\text{m}$ in 70 ms, driven by diffusion in the liquid phase (compare the estimate above of 0.1 s). For the next 12 s, the interface stalls while the solute step trapped in the solid phase diffuses outward ($(2.8 \mu\text{m})^2/D_s = \mathcal{O}(80 \text{ s})$). Once the solute gradient in the solid reaches the new position of the interface, the solidification front begins to move, driven by diffusion in the solid. When the solute in the solid becomes uniform, the interface stalls again after $\approx 4000 \text{ s}$, having moved another $2.9 \mu\text{m}$ (recall the estimate of 1000 s for equilibration in the solid). After this point, there is essentially no further motion of the interface and barely perceptible changes in the concentration field.

Note: This evolution is qualitatively consistent with that seen in [examples.phase.binary](#), but the interface does not move as far and the bulk concentrations are further from the phase diagram values. The computation also takes substantially longer than the uncoupled variant.

24.11.5 examples.phase.impingement

Modules

<code>examples.phase.impingement.mesh20x20</code>	Solve for the impingement of four grains in two dimensions.
<code>examples.phase.impingement.mesh40x1</code>	Solve for the impingement of two grains in one dimension.
<code>examples.phase.impingement.test</code>	

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.
>>> 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
```

(continues on next page)

(continued from previous page)

```

...     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
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + '.gz').flat

```

We step the solution in time, plotting as we go if running interactively,

```

>>> 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.

```

>>> 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,

```

>>> 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

```

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 [13]

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> 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,

The theta equation is a bit more sensitive than the phase equation, so we require a stricter tolerance.

```
>>> solver = thetaEq.getDefaultSolver(tolerance=1e-9)
```

```
>>> steps = 10
>>> for i in range(steps):
...     theta.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration, solver=solver)
...     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
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + '.gz')
>>> testData = CellVariable(mesh=mesh, value=testData)
>>> print(theta.allclose(testData))
True
```

`examples.phase.impingement.test`

24.11.6 examples.phase.missOrientation

Modules

<code>examples.phase.missOrientation.circle</code>	In this example, a phase equation is solved in one dimension with a misorientation between two solid domains.
<code>examples.phase.missOrientation.mesh1D</code>	In this example a phase equation is solved in 1 dimension with a misorientation present.
<code>examples.phase.missOrientation.modCircle</code>	In this example a phase equation is solved in one dimension with a misorientation present.
<code>examples.phase.missOrientation.test</code>	

examples.phase.missOrientation.circle

In this example, a phase equation is solved in one dimension with a misorientation between two solid domains. 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

$$m_1(\phi, T) = \phi - \frac{1}{2} - T\phi(1 - \phi)$$

The initial conditions are:

$$\begin{aligned} \phi &= 1 & \forall x \\ \theta &= \begin{cases} 1 & \text{for } (x - L/2)^2 + (y - L/2)^2 > (L/4)^2 \\ 0 & \text{for } (x - L/2)^2 + (y - L/2)^2 \leq (L/4)^2 \end{cases} \\ T &= 1 & \forall x \end{aligned}$$

and boundary conditions $\phi = 1$ for $x = 0$ and $x = L$.

Here the phase equation is solved with an explicit technique.

The solution is allowed to evolve for `steps = 100` time steps.

```
>>> for step in range(steps):
...     phaseEq.solve(phase, dt=timeStepDuration, solver=solver)
```

The solution is compared with test data. The test data was created with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `circle.gz` extracts the data and compares it with the phase variable.

```
>>> import os
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + '.gz')
>>> print(phase.allclose(testData))
1
```

examples.phase.missOrientation.mesh1D

In this example a phase equation is solved in 1 dimension with a misorientation present. 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

$$m_1(\phi, T) = \phi - \frac{1}{2} - T\phi(1 - \phi)$$

The initial conditions are:

$$\begin{aligned} \phi &= 1 && \text{for } 0 \leq x \leq L \\ \theta &= \begin{cases} 1 & \text{for } 0 \leq x \leq L/2 \\ 0 & \text{for } L/2 < x \leq L \end{cases} \\ T &= 1 && \text{for } 0 \leq x \leq L \end{aligned}$$

and boundary conditions $\phi = 1$ for $x = 0$ and $x = L$.

Here the phase equation is solved with an explicit technique.

The solution is allowed to evolve for `steps = 100` time steps.

```
>>> for step in range(steps):
...     phaseEq.solve(phase, dt=timeStepDuration, solver=solver)
```

The solution is compared with test data. The test data was created with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `mesh1D.gz` extracts the data and compares it with the `theta` variable.

```
>>> import os
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + '.gz')
>>> print(phase.allclose(testData))
1
```

examples.phase.missOrientation.modCircle

In this example a phase equation is solved in one dimension with a misorientation present. 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

$$m_1(\phi, T) = \phi - \frac{1}{2} - T\phi(1 - \phi)$$

The initial conditions are:

$$\begin{aligned} \phi &= 1 && \forall x \\ \theta &= \begin{cases} 2\pi/3 & \text{for } (x - L/2)^2 + (y - L/2)^2 > (L/4)^2 \\ -2\pi/3 & \text{for } (x - L/2)^2 + (y - L/2)^2 \leq (L/4)^2 \end{cases} \\ T &= 1 && \forall x \end{aligned}$$

and boundary conditions $\phi = 1$ for $x = 0$ and $x = L$.

Here the phase equation is solved with an explicit technique.

The solution is allowed to evolve for `steps = 100` time steps.

```
>>> for step in range(steps):
...     phaseEq.solve(phase, dt=timeStepDuration, solver=solver)
```

The solution is compared with test data. The test data was created with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `modCircle.gz` extracts the data and compares it with the phase variable.

```
>>> import os
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + '.gz')
>>> print(phase.allclose(testData))
1
```

`examples.phase.missOrientation.test`

24.11.7 examples.phase.phaseViewer

1D Viewer that calculates and plots the position of the phase field interface.

Classes

`PhaseViewer`(phase, C, sharp, elapsed, L, deltaA)

```
>>> import fipy as fp
```

```
class examples.phase.phaseViewer.PhaseViewer(phase, C, sharp, elapsed, L, deltaA, title=None,
                                              tmin=None, tmax=None, **kwlimits)
```

Bases: `Matplotlib1DViewer`

```
>>> import fipy as fp
>>> mesh = fp.Grid1D(nx=100)
>>> x, = mesh.cellCenters
>>> xVar = fp.CellVariable(mesh=mesh, name="x", value=x)
>>> k = fp.Variable(name="k", value=0.)
>>> viewer = PhaseViewer(vars=(fp.numerix.sin(k * xVar) + 2,
...                               fp.numerix.cos(k * xVar / fp.numerix.pi) + 2),
...                       xmin=10, xmax=90,
...                       datamin=1.1, datamax=4.0,
...                       title="PhaseViewer test")
>>> for kval in fp.numerix.arange(0, 0.3, 0.03):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()
```

```
>>> viewer.cmap = "ocean"
>>> viewer.log = True
```

```
>>> viewer.title = "PhaseViewer changed"
>>> 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** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **xmax** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **datamin** (*float*, *optional*) – displayed range of data. Any limit set to a (default) value of *None* will autoscale. (*ymin* and *ymax* are synonyms for *datamin* and *datamax*).
- **datamax** (*float*, *optional*) – 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

property axes

The *Matplotlib Axes*.

property cmap

The *Matplotlib Colormap*.

property colorbar

The *Matplotlib Colorbar*.

property fig

The *Matplotlib Figure*.

property id

The *Matplotlib Figure* number.

property lines

The collection of *Matplotlib Line2D* objects representing the plotted data.

property log

Whether data has logarithmic 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.

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** (*float*, *optional*) – 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.
- **xmax** (*float*, *optional*) – 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.
- **ymin** (*float*, *optional*) – 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.
- **ymax** (*float*, *optional*) – 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.
- **zmin** (*float*, *optional*) – 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.
- **zmax** (*float*, *optional*) – 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.
- **datamin** (*float*, *optional*) – 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.
- **datamax** (*float*, *optional*) – 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.

property title

The text appearing at the top center.

(default: if `len(self.vars) == 1`, the name of the only *Variable*, otherwise `""`.)

property vars

The *Variable* or list of *Variable* objects to display.

24.11.8 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 [13] 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)
```

(continues on next page)

(continued from previous page)

```
...         + (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 $\beta = \frac{1-\Phi^2}{1+\Phi^2}$, $\Phi = \tan\left(\frac{N}{2}\psi\right)$, $\psi = \theta + \arctan\frac{\partial \phi / \partial y}{\partial \phi / \partial x}$, 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(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]):
...     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:

```

>>> 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)

```

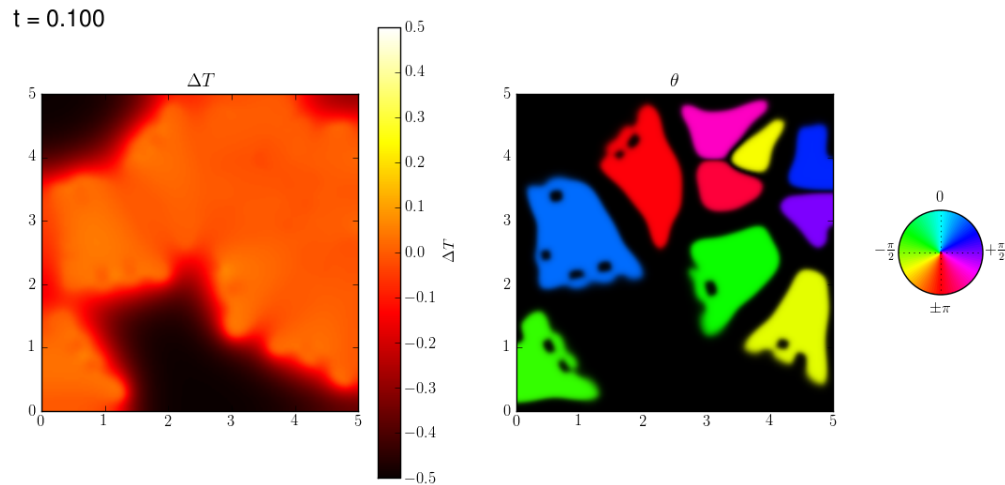
...         from matplotlib import colors
...
...         hsv = numerix.empty(orientation.shape + (3,))
...         hsv[..., 0] = (orientation / numerix.pi + 1) / 2.
...         hsv[..., 1] = 1.
...         hsv[..., 2] = phase
...
...         return colors.hsv_to_rgb(hsv)
...
...     @property
...     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,

```
>>> 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.

24.11.9 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 [13] 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
>>> heatEq = (TransientTerm(var=dT)
...           == DiffusionTerm(coeff=DT, var=dT)
```

(continues on next page)

(continued from previous page)

```
...         + 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:

```
>>> 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"
```

(continues on next page)

(continued from previous page)

```

...         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:
...     viewer = MultiViewer(viewers=(Viewer(vars=dT,
...                                         datamin=-0.5,
...                                         datamax=0.5),
...                                   Viewer(vars=phase,
...                                         datamin=0.,
...                                         datamax=1.),
...                                   Viewer(vars=theta,

```

(continues on next page)

(continued from previous page)

```

...         datamin=-numerix.pi,
...         datamax=numerix.pi)))
...     viewer.plot()

```

and iterate the solution in time, plotting as we go,

```
>>> eq = thetaEq & phaseEq & heatEq
```

```

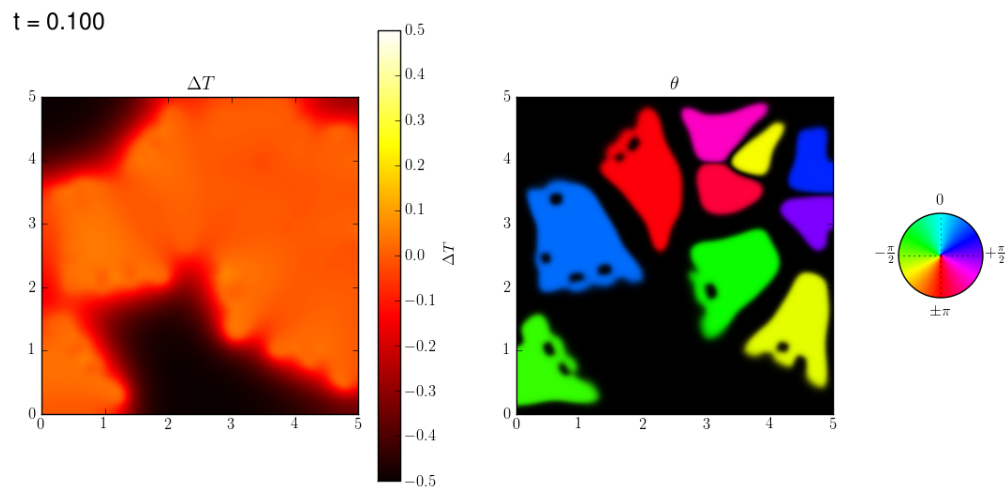
>>> if __name__ == "__main__":
...     total_time = 2.
...     else:
...         total_time = dt * 10
>>> elapsed = 0.
>>> save_interval = 0.002
>>> save_at = save_interval

```

```

>>> 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

```



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.

24.11.10 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 [22] [23]. 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

$$\frac{1}{M_\phi} \frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \sum_{j=0}^N C_j \left[\mu_j^{SL}(T) p'(\phi) + \frac{W_j}{2} g'(\phi) \right]$$

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:

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}} + \underbrace{D_j \nabla \cdot \frac{C_j}{1 + \sum_{\substack{k=0 \\ k \neq j}}^M C_k} \left\{ \overbrace{\frac{\rho}{RT} \left[\mu_j^{\circ SL} \nabla p(\phi) + \frac{W_j}{2} \nabla g(\phi) \right]}^{\text{phase transformation}} - \underbrace{\sum_{\substack{i=0 \\ i \neq j}}^M \nabla C_i}_{\text{counter diffusion}} \right\}}_{\text{convection}}$$

```

>>> 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

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}} + \underbrace{D_j \nabla \cdot \frac{C_j}{1 - \sum_{\substack{k=M+1 \\ k \neq j}}^{N-1} C_k} \left\{ \overbrace{\frac{C_N}{RT} \left[(\mu_j^{\circ SL} - \mu_N^{\circ SL}) \nabla p(\phi) + \frac{W_j - W_N}{2} \nabla g(\phi) \right]}^{\text{phase transformation}} + \underbrace{\sum_{\substack{i=M+1 \\ i \neq j}}^{N-1} \nabla C_i}_{\text{counter diffusion}} \right\}}_{\text{convection}}$$

```
>>> 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. The initial residual is much larger than the norm of the right-hand-side vector, so we use “initial” tolerance scaling.

```
>>> solver = DefaultAsymmetricSolver(criterion="initial", tolerance=1e-8)
```

```
>>> dt = 10000
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
```

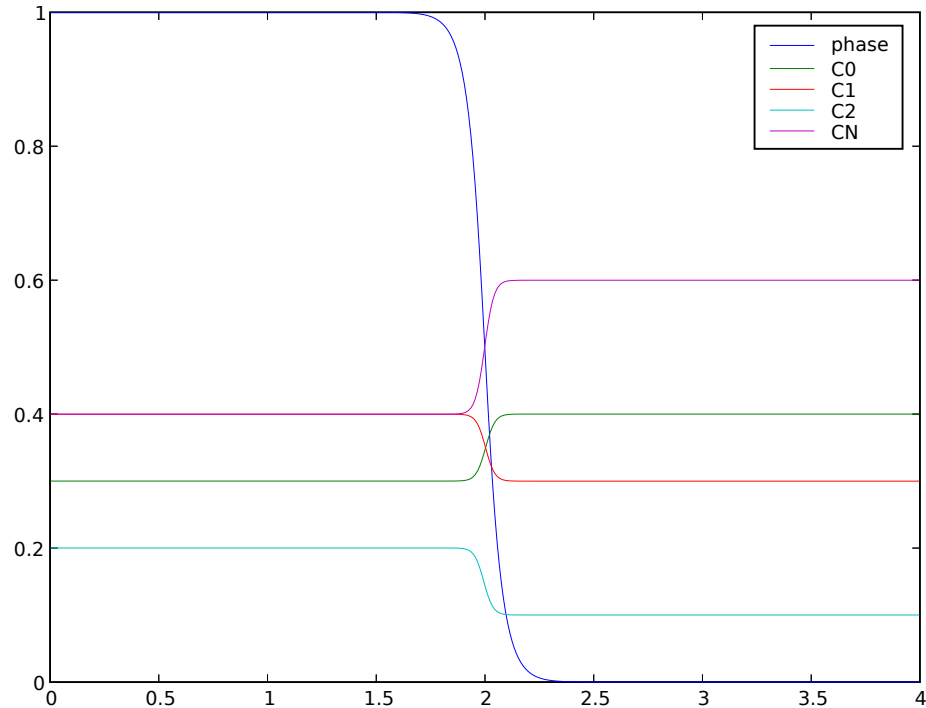
(continues on next page)

(continued from previous page)

```

...     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)

```

(continues on next page)

True

24.11.11 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,

$$\frac{1}{M_\phi} \frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \quad (24.11)$$

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)
```

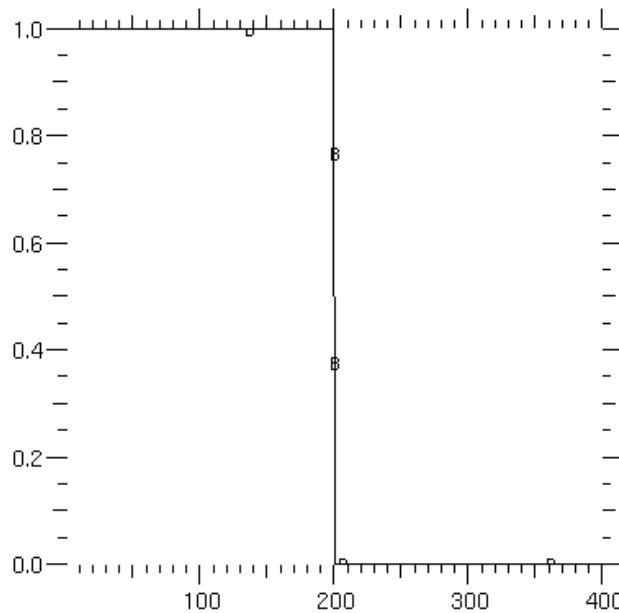
and set a step-function initial condition

$$\phi = \begin{cases} 1 & \text{for } x \leq L/2 \\ 0 & \text{for } x > L/2 \end{cases} \quad \text{at } t = 0$$

```
>>> 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

$$\phi = \frac{1}{2} \left[1 - \tanh \frac{x - L/2}{2\sqrt{\kappa/W}} \right] \quad (24.12)$$

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

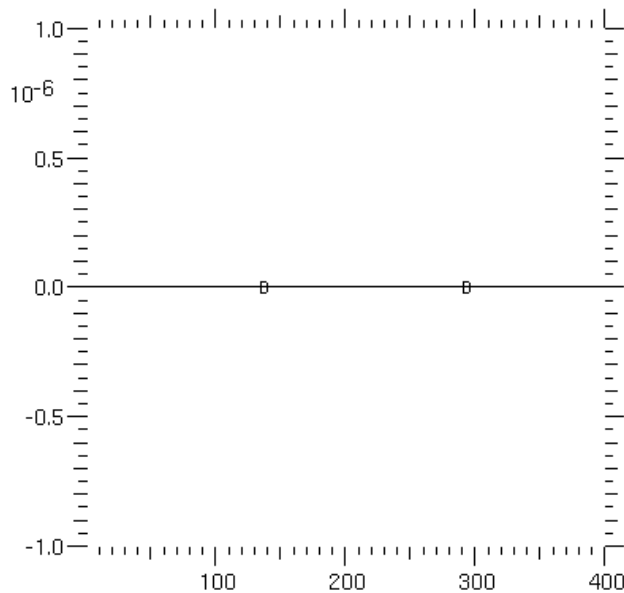
```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> S0 = mPhi * phase * (1 - phase)
>>> eq = S0 + DiffusionTerm(coeff=kappa)
```

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 (24.11)

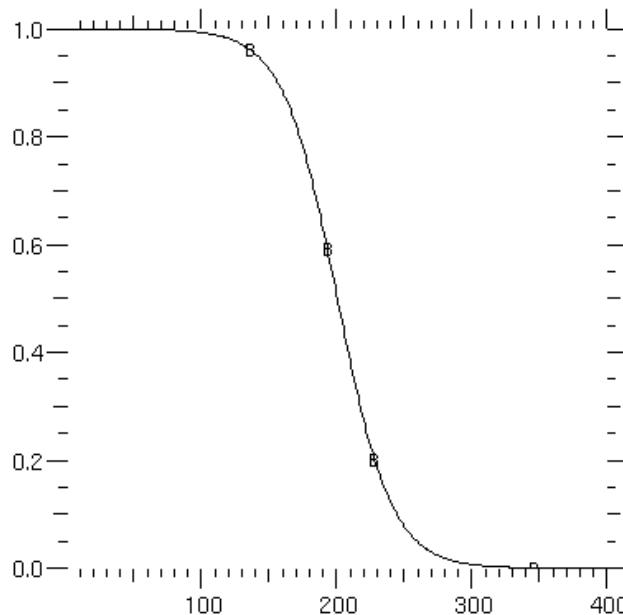
```
>>> eq = TransientTerm() == DiffusionTerm(coeff=kappa) + S0
```

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

```
>>> 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 (24.12) 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)
```

```
>>> 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)
```

```
>>> for i in range(5):
...     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("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 [31]

```
>>> 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
>>> 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.

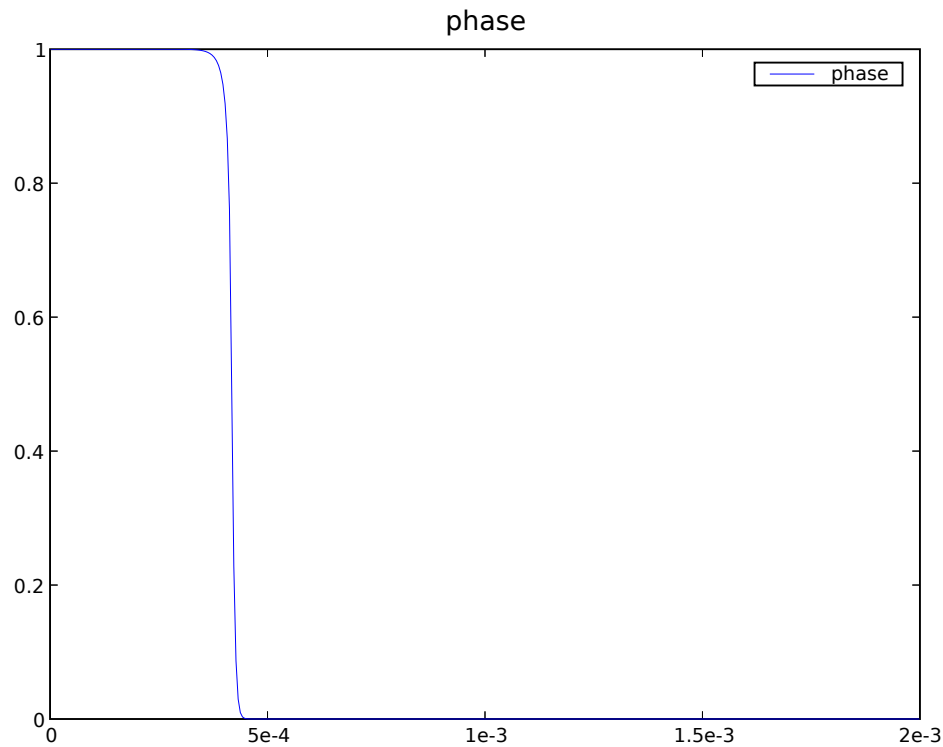
```
>>> print(phase.allclose(analyticalArray, rtol = 5, atol = 2e-3))
1
```

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) < 4.2e-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...")
```



24.11.12 examples.phase.symmetry

This example creates four symmetric quadrilateral regions in a box. We start with a [CellVariable](#) object that contains the following values:

$$\phi(x, y) = xy, 0 \leq x \leq L, 0 \leq y \leq L$$

We wish to create 4 symmetric regions such that

$$\phi(x, y) = \phi(L - x, y) = \phi(L - x, L - y) = \phi(L - x, L - y), 0 \leq x \leq L/2, 0 \leq y \leq L/2$$

We create a square domain

```
>>> from fipy import CellVariable, Grid2D, Viewer
>>> from fipy.tools import numerix
```

```
>>> N = 20
>>> L = 1.
>>> dx = L / N
>>> dy = L / N
```

```
>>> mesh = Grid2D(
...     dx = dx,
...     dy = dy,
...     nx = N,
...     ny = N)
```

```
>>> var = CellVariable(name = "test", mesh = mesh)
```

First set the values as given in the above equation:

```
>>> x, y = mesh.cellCenters
>>> var.setValue(x * y)
```

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=0, datamax=L * L / 4.)
...     viewer.plot()
```

The bottom-left quadrant is mirrored into each of the other three quadrants

```
>>> q = (x > L / 2.) & (y < L / 2.)
>>> var[q] = var(((L - x)[q], y[q]))
>>> q = (x < L / 2.) & (y > L / 2.)
>>> var[q] = var((x[q], (L - y)[q]))
>>> q = (x > L / 2.) & (y > L / 2.)
>>> var[q] = var(((L - x)[q], (L - y)[q]))
```

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

The following code tests the results with a different algorithm:

```
>>> testResult = numerix.zeros((N // 2, N // 2), 'd')
>>> bottomRight = numerix.zeros((N // 2, N // 2), 'd')
>>> topLeft = numerix.zeros((N // 2, N // 2), 'd')
>>> topRight = numerix.zeros((N // 2, N // 2), 'd')
>>> for j in range(N // 2):
...     for i in range(N // 2):
...         x = dx * (i + 0.5)
...         y = dx * (j + 0.5)
...         testResult[i, j] = x * y
...         bottomRight[i, j] = var(((L - x,), (y,)))[0]
...         topLeft[i, j] = var((x,), (L - y,))[0]
...         topRight[i, j] = var((L - x,), (L - y,))[0]
>>> numerix.allclose(testResult, bottomRight, atol = 1e-10)
1
>>> numerix.allclose(testResult, topLeft, atol = 1e-10)
1
>>> numerix.allclose(testResult, topRight, atol = 1e-10)
1
```

24.11.13 examples.phase.test

24.12 examples.reactiveWetting

Modules

<code>examples.reactiveWetting.liquidVapor1D</code>	Solve a single-component, liquid-vapor, van der Waals system.
<code>examples.reactiveWetting.liquidVapor2D</code>	A 2D version of the 1D example.
<code>examples.reactiveWetting.test</code>	Run all the test cases in <code>examples/ reactiveWetting/</code>

24.12.1 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.* [5]. The free energy for this system takes the form,

$$f = -\frac{e\rho^2}{m^2} + \frac{RT}{m} \left(\ln \frac{\rho}{m - \bar{v}\rho} \right) \quad (24.13)$$

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,

$$P(\rho^l) = P(\rho^v) \quad (24.14)$$

and

$$\mu(\rho^l) = \mu(\rho^v) \quad (24.15)$$

where μ is the chemical potential,

$$\mu = \frac{\partial f}{\partial \rho} \quad (24.16)$$

and P is the pressure,

$$P = \rho\mu - f \quad (24.17)$$

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. (24.14) and (24.15). Firstly, Eqs. (24.13), (24.16) and (24.17) 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,

$$\frac{\partial \rho}{\partial t} + \partial_j (\rho u_j) = 0 \quad (24.18)$$

and

$$\frac{\partial (\rho u_i)}{\partial t} + \partial_j (\rho u_i u_j) = \partial_j (\nu [\partial_j u_i + \partial_i u_j]) - \rho \partial_i \mu^{NC} \quad (24.19)$$

where the non-classical potential, μ^{NC} , is given by,

$$\mu^{NC} = \frac{\delta F}{\delta \rho} = \mu - \epsilon T \partial_j^2 \rho \quad (24.20)$$

As usual, to proceed, we define a mesh

```
>>> Lx = 1e-6
>>> nx = 100
>>> dx = Lx / nx
>>> mesh = Grid1D(nx=nx, dx=dx)
```

and the independent variables.

```
>>> density = CellVariable(mesh=mesh, hasOld=True, name=r'\rho$')
>>> velocity = CellVariable(mesh=mesh, hasOld=True, name=r'$u$')
>>> densityPrevious = density.copy()
>>> velocityPrevious = velocity.copy()
```

The system of equations is solved in a fully coupled manner using a block matrix. Defining μ^{NC} as an independent variable makes it easier to script the equations without using higher order terms.

```
>>> potentialNC = CellVariable(mesh=mesh, name=r'\mu^{NC}$')
```

```
>>> epsilon = 1e-16
>>> freeEnergy = (f(density) + epsilon * temperature / 2 * density.grad.mag**2).
    cellVolumeAverage
```

In order to solve the equations numerically, an interpolation method is used to prevent the velocity and density fields decoupling. The following velocity correction equation (expressed in discretized form) prevents decoupling from occurring,

$$u_{i,f}^c = \frac{A_f d_f}{\bar{d}_f} \left(\overline{\rho \partial_i \mu^{NC}}_f - \bar{\rho}_f \partial_{i,f} \mu^{NC} \right) \quad (24.21)$$

where A_f is the face area, d_f is the distance between the adjacent cell centers and $\bar{a}_f$ is the momentum conservation equation's matrix diagonal. The overbar refers to an averaged value between the two adjacent cells to the face. The notation $\partial_{i,f}$ refers to a derivative evaluated directly at the face (not averaged). The variable u_i^c is used to modify the velocity used in Eq. (24.18) such that,

$$\frac{\partial \rho}{\partial t} + \partial_j (\rho [u_j + u_i^c]) = 0 \quad (24.22)$$

Equation (24.22) becomes

```
>>> matrixDiagonal = CellVariable(mesh=mesh, name=r'$a_f$', value=1e+20, hasOld=True)
>>> correctionCoeff = mesh._faceAreas * mesh._cellDistances / matrixDiagonal.faceValue
>>> massEqn = TransientTerm(var=density) \
...     + VanLeerConvectionTerm(coeff=velocity.faceValue + correctionCoeff \
...                             * (density * potentialNC.grad).faceValue, \
...                             var=density) \
...     - DiffusionTerm(coeff=correctionCoeff * density.faceValue**2,
    var=potentialNC)
```

where the first term on the LHS of Eq. (24.21) is calculated in an explicit manner in the `VanLeerConvectionTerm` and the second term is calculated implicitly as a `DiffusionTerm` with μ^{NC} as the independent variable.

In order to write Eq. (24.19) as a *FiPy* expression, the last term is rewritten such that,

$$\rho \partial_i \mu^{NC} = \partial_i (\rho \mu^{NC}) - \mu^{NC} \partial_i \rho$$

which results in

```
>>> viscosity = 1e-3
>>> ConvectionTerm = CentralDifferenceConvectionTerm
>>> momentumEqn = TransientTerm(coeff=density, var=velocity) \
...     + ConvectionTerm(coeff=[[1]] * density.faceValue * velocity.faceValue,
    var=velocity) \
```

(continues on next page)

(continued from previous page)

```
... == 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. (24.20) is linearized in ρ such that

$$\mu^{NC} = \mu^* + \left(\frac{\partial \mu}{\partial \rho} \right)^* (\rho - \rho^*) - \epsilon T \partial_j^2 \rho \quad (24.23)$$

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. (24.23) 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)
...     for viewer in viewers:
...         viewer.plot()
```

(continues on next page)

(continued from previous page)

```
...     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*.

Beginning with *FiPy* 4, solver *Convergence* tolerance is normalized by the magnitude of the right-hand-side (RHS) vector. For this particular problem, the initial residual is much smaller than the RHS and so the solver gets “stuck”. Changing the normalization to use the initial residual at the beginning of each sweep allows the solution to progress. Another option would be to scale the tolerance appropriately, but the value is so small ($\sim 10^{-22}$) that this results in underflow issues for some solvers.

```
>>> solver = coupledEqn.getDefaultSolver(criterion="initial", tolerance=1e-12)
```

Note: *PETSc* intrinsically wants to use “preconditioned” normalization, which prevents the solver from getting “stuck” because the preconditioner effectively scales the RHS to be similar in magnitude to the residual. Unfortunately, this normalization is not available for the other solver suites, so we don't use it as the default.

```
>>> 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
...
...         density.updateOld()
...         velocity.updateOld()
...         matrixDiagonal.updateOld()
...         residual = coupledEqn.solve()
...         sweep += 1
...         if sweep > 20:
...             dt *= 0.1
...             residual = 1.
...             initialResidual = None
...             break
...     timestep += dt
```

(continues on next page)

(continued from previous page)

```

...     dt = min(dt, dx / max(abs(velocity)) * cfl)
...
...     coupledEqn.cacheMatrix()
...     residual = coupledEqn.sweep(dt=dt, solver=solver)
...
...     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.
...         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

```

24.12.2 examples.reactiveWetting.liquidVapor2D

A 2D version of the 1D example.

```
>>> molarWeight = 0.118
>>> ee = -0.455971
>>> gasConstant = 8.314
>>> temperature = 650.
>>> vbar = 1.3e-05
```

```
>>> liquidDensity = 7354.3402662299995
>>> vaporDensity = 82.855803327810008
```

```
>>> from fipy import CellVariable, Grid2D, 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))
```

```
>>> Lx = 1e-6
>>> nx = 100
>>> dx = Lx / nx
>>> mesh = Grid2D(nx=nx, ny=nx, dx=dx, dy=dx)
```

```
>>> density = CellVariable(mesh=mesh, hasOld=True, name=r'\rho$')
>>> velocityX = CellVariable(mesh=mesh, hasOld=True, name=r'$u_x$')
>>> velocityY = CellVariable(mesh=mesh, hasOld=True, name=r'$u_y$')
>>> velocityVector = CellVariable(mesh=mesh, name=r'\vec{u}$', rank=1)
>>> densityPrevious = density.copy()
>>> velocityXPrevious = velocityX.copy()
>>> velocityYPrevious = velocityY.copy()
```

```
>>> potentialNC = CellVariable(mesh=mesh, name=r'\mu^{NC}$')
```

```
>>> epsilon = 1e-16
>>> freeEnergy = (f(density) + epsilon * temperature / 2 * density.grad.mag**2).
↳ cellVolumeAverage
```

```
>>> matrixDiagonal = CellVariable(mesh=mesh, name=r'$a_f$', value=1e+20, hasOld=True)
>>> correctionCoeff = mesh._faceAreas * mesh._cellDistances / matrixDiagonal.faceValue
>>> massEqn = TransientTerm(var=density) \
```

(continues on next page)

(continued from previous page)

```

...         + VanLeerConvectionTerm(coeff=velocityVector.faceValue + correctionCoeff \
...                                   * (density * potentialNC.grad).faceValue, \
...                                   var=density) \
...         - DiffusionTerm(coeff=correctionCoeff * density.faceValue**2,
↪var=potentialNC)

```

```

>>> viscosity = 1e-3
>>> ConvectionTerm = CentralDifferenceConvectionTerm
>>> ##matXX = numerix.array([[2], [0]], [[0], [1]])
>>> ##matYY = numerix.array([[1], [0]], [[0], [2]])
>>> ##matXY = numerix.array([[0], [0.5]], [[0.5], [0]])
>>> ##matYX = matXY
>>> matXX = 1
>>> matYY = 1
>>> matXY = 0
>>> matYX = 0
>>> momentumXEqn = TransientTerm(coeff=density, var=velocityX) \
...                   + ConvectionTerm(coeff=density.faceValue * velocityVector.faceValue,
↪var=velocityX) \
...                   == DiffusionTerm(coeff=(viscosity * matXX,), var=velocityX) \
...                   + DiffusionTerm(coeff=(viscosity * matXY,), var=velocityY) \
...                   - ConvectionTerm(coeff=density.faceValue * [[1], [0]],
↪var=potentialNC) \
...                   + ImplicitSourceTerm(coeff=density.grad[0], var=potentialNC)

```

```

>>> momentumYEqn = TransientTerm(coeff=density, var=velocityY) \
...                   + ConvectionTerm(coeff=density.faceValue * velocityVector.faceValue,
↪var=velocityY) \
...                   == DiffusionTerm(coeff=(viscosity * matYY,), var=velocityY) \
...                   + DiffusionTerm(coeff=(viscosity * matYX,), var=velocityX) \
...                   - ConvectionTerm(coeff=density.faceValue * [[0], [1]],
↪var=potentialNC) \
...                   + ImplicitSourceTerm(coeff=density.grad[1], var=potentialNC)

```

```

>>> velocityX.constrain(0, mesh.facesLeft & mesh.facesRight)
>>> velocityY.constrain(0, mesh.facesTop & mesh.facesBottom)

```

```

>>> potentialDerivative = 2 * ee / molarWeight**2 + \
...                       gasConstant * temperature * molarWeight / density /
↪(molarWeight - vbar * density)**2

```

```

>>> potential = mu(density)

```

```

>>> potentialNCEqn = ImplicitSourceTerm(coeff=1, var=potentialNC) \
...                   == potential \
...                   + ImplicitSourceTerm(coeff=potentialDerivative, var=density) \
...                   - potentialDerivative * density \
...                   - DiffusionTerm(coeff=epsilon * temperature, var=density)

```

```

>>> potentialNC.faceGrad.constrain(value=[[0], [0]], where=mesh.exteriorFaces)

```

```
>>> coupledEqn = massEqn & momentumXEqn & momentumYEqn & potentialNEqn
```

```
>>> numerix.random.seed(2012)
>>> density[:] = (liquidDensity + vaporDensity) / 2 * \
...     (1 + 0.01 * (2 * numerix.random.random(mesh.numberOfCells) - 1))
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewers = Viewer(density), Viewer(velocityVector), Viewer(potentialNC)
...     for viewer in viewers:
...         viewer.plot()
...     input('arrange viewers')
...     for viewer in viewers:
...         viewer.plot()
```

```
>>> cfl = 0.1
>>> tolerance = 1e-1
>>> dt = 1e-14
>>> timestep = 0
>>> relaxation = 0.5
>>> sweeps = 0
>>> if __name__ == '__main__':
...     totalSteps = 1e+10
...     totalSweeps = 1e+10
... else:
...     totalSteps = 1
...     totalSweeps = 1
```

Beginning with *FiPy* 4, solver *Convergence* tolerance is normalized by the magnitude of the right-hand-side (RHS) vector. For this particular problem, the initial residual is much smaller than the RHS and so the solver gets “stuck”. Changing the normalization to use the initial residual at the beginning of each sweep allows the solution to progress.

```
>>> from fipy.solvers import solver_suite
>>> if solver_suite == "petsc":
...     # PETSc's default ILU preconditioner does not behave well
...     # for this problem
...     from fipy import SSORPreconditioner
...     precon = SSORPreconditioner()
... else:
...     precon = "default"
>>> solver = coupledEqn.getDefaultSolver(criterion="initial", precon=precon)
```

Note: *PETSc* intrinsically wants to use “preconditioned” normalization, which prevents the solver from getting “stuck” because the preconditioner effectively scales the RHS to be similar in magnitude to the residual. Unfortunately, this normalization is not available for the other solver suites, so we don’t use it as the default.

```
>>> while timestep < totalSteps:
...
...     sweep = 0
...     dt *= 1.1
```

(continues on next page)

(continued from previous page)

```

...     residual = 1.
...     initialResidual = None
...
...     density.updateOld()
...     velocityX.updateOld()
...     velocityY.updateOld()
...     matrixDiagonal.updateOld()
...
...     while residual > tolerance and sweeps < totalSweeps:
...         sweeps += 1
...         densityPrevious[:] = density
...         velocityXPrevious[:] = velocityX
...         velocityYPrevious[:] = velocityY
...         previousResidual = residual
...         velocityVector[0] = velocityX
...         velocityVector[1] = velocityY
...
...         dt = min(dt, dx / max(abs(velocityVector.mag)) * cfl)
...
...         coupledEqn.cacheMatrix()
...         residual = coupledEqn.sweep(dt=dt, solver=solver)
...
...         if initialResidual is None:
...             initialResidual = residual
...
...         residual = residual / initialResidual
...
...         if residual > previousResidual * 1.1 or sweep > 20:
...             density[:] = density.old
...             velocityX[:] = velocityX.old
...             velocityY[:] = velocityY.old
...             matrixDiagonal[:] = matrixDiagonal.old
...             dt = dt / 10.
...             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
...             velocityX[:] = relaxation * velocityX + (1 - relaxation) *
↪ velocityXPrevious
...             velocityY[:] = relaxation * velocityY + (1 - relaxation) *
↪ velocityYPrevious
...
...             sweep += 1
...
...         if __name__ == '__main__' and timestep % 1 == 0:
...             print('timestep: %e / %e, dt: %1.5e, free energy: %1.5e' % (timestep,
↪ totalSteps, dt, freeEnergy))
...             for viewer in viewers:

```

(continues on next page)

(continued from previous page)

```
...         viewer.plot()
...
...     timestep += 1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input('finished')
```

24.12.3 examples.reactiveWetting.test

Run all the test cases in *examples/reactiveWetting/*

24.13 examples.riemann

Modules

<code>examples.riemann.acoustics</code>	Test
<code>examples.riemann.test</code>	

24.13.1 examples.riemann.acoustics

Test

```
>>> print((0.4 < max(q.globalValue[0]) < 0.5))
True
```

24.13.2 examples.riemann.test

24.14 examples.test

Run all the test cases in *examples/*

24.15 examples.updating

Modules

<code>examples.updating.update0_1to1_0</code>	How to update scripts from version 0.1 to 1.0.
<code>examples.updating.update1_0to2_0</code>	How to update scripts from version 1.0 to 2.0.
<code>examples.updating.update2_0to3_0</code>	How to update scripts from version 2.0 to 3.0.

24.15.1 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(
```

(continues on next page)

(continued from previous page)

```
...     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,
...     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
```

(continues on next page)

(continued from previous page)

```
>>> 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 $n_y = 1$ 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 solver 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.

24.15.2 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, deltay)
```

- `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.linearCGSSolver import LinearCGSSolver
```

to either

```
>>> from fipy import LinearCGSSolver
```

or

```
>>> from fipy.solvers.petsc.linearCGSSolver import LinearCGSSolver
```

- 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`.

24.15.3 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*.

Bibliography

- [1] 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](https://doi.org/10.1146/annurev.matsci.32.101901.155803).
- [2] 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](https://doi.org/10.1146/annurev.matsci.32.112001.132041).
- [3] G. B. McFadden. Phase-field models of solidification. *Contemporary Mathematics*, 306:107–145, 2002.
- [4] 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](https://doi.org/10.1016/j.actbio.2010.09.019).
- [5] 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](https://doi.org/10.1103/PhysRevE.82.051601).
- [6] 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](https://doi.org/10.1063/1.3561487).
- [7] 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](https://doi.org/10.1103/PhysRevLett.87.016102).
- [8] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1996.
- [9] Scott Chacon. *Pro Git*. Apress, 2009. URL: <http://git-scm.com/book>.
- [10] Guido van Rossum. *Python Tutorial*. URL: <http://docs.python.org/tut/>.
- [11] Mark Pilgrim. *Dive Into Python*. Apress, 2004. ISBN 1590593561. URL: <http://diveintopython.org>.
- [12] Guido van Rossum. *Python Reference Manual*. URL: <http://docs.python.org/ref/>.
- [13] 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](https://doi.org/10.1016/S1359-6454(03)00388-4).
- [14] 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/>.
- [15] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow*. Taylor and Francis, 1980.
- [16] H. K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics*. Longman Scientific and Technical, 1995.

- [17] 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>.
- [18] 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.
- [19] John W. Cahn. Free energy of a nonuniform system. II. Thermodynamic basis. *Journal of Chemical Physics*, 30(5):1121–1124, 1959.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] 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.
- [24] 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.
- [25] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics*. Springer, 1996.
- [26] 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.
- [27] 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.
- [28] 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.
- [29] 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.
- [30] 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.
- [31] 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.
- [32] Greg Ward. *Installing Python Modules*. URL: <http://docs.python.org/inst/>.
- [33] 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>.

Python Module Index

e

- examples, 959
- examples.benchmarking, 960
 - examples.benchmarking.benchmarker, 960
 - examples.benchmarking.size, 960
 - examples.benchmarking.steps, 960
 - examples.benchmarking.utils, 960
 - examples.benchmarking.versions, 960
- examples.cahnHilliard, 960
 - examples.cahnHilliard.mesh2D, 961
 - examples.cahnHilliard.mesh2DCoupled, 963
 - examples.cahnHilliard.mesh3D, 966
 - examples.cahnHilliard.sphere, 968
 - examples.cahnHilliard.sphereDaemon, 970
 - examples.cahnHilliard.tanh1D, 970
 - examples.cahnHilliard.test, 972
- examples.chemotaxis, 972
 - examples.chemotaxis.input, 973
 - examples.chemotaxis.input2D, 975
 - examples.chemotaxis.parameters, 977
 - examples.chemotaxis.test, 977
- examples.convection, 977
 - examples.convection.advection, 978
 - examples.convection.advection.explicitUpwind, 978
 - examples.convection.advection.implicitUpwind, 978
 - examples.convection.advection.vanLeerUpwind, 979
 - examples.convection.exponential1D, 979
 - examples.convection.exponential1D.cylindricalMesh1D, 980
 - examples.convection.exponential1D.cylindricalMesh2DNonUniform, 981
 - examples.convection.exponential1D.mesh1D, 983
 - examples.convection.exponential1D.tri2D, 984
 - examples.convection.exponential1DBack, 985
 - examples.convection.exponential1DBack.mesh1D, 985
 - examples.convection.exponential1DSource, 986
 - examples.convection.exponential1DSource.mesh1D, 987
 - examples.convection.exponential1DSource.tri2D, 988
 - examples.convection.exponential2D, 989
 - examples.convection.exponential2D.cylindricalMesh2D, 989
 - examples.convection.exponential2D.cylindricalMesh2DNonUniform, 991
 - examples.convection.exponential2D.mesh2D, 992
 - examples.convection.exponential2D.tri2D, 993
 - examples.convection.peclet, 994
 - examples.convection.powerLaw1D, 996
 - examples.convection.powerLaw1D.mesh1D, 996
 - examples.convection.powerLaw1D.tri2D, 997
 - examples.convection.robin, 998
 - examples.convection.source, 1000
 - examples.convection.test, 1001
 - examples.diffusion, 1001
 - examples.diffusion.anisotropy, 1001
 - examples.diffusion.circle, 1003
 - examples.diffusion.circleQuad, 1008
 - examples.diffusion.coupled, 1013
 - examples.diffusion.electrostatics, 1015
 - examples.diffusion.explicit, 1019
 - examples.diffusion.explicit.mesh1D, 1020
 - examples.diffusion.explicit.mixedelement, 1021
 - examples.diffusion.explicit.test, 1022
 - examples.diffusion.explicit.tri2D, 1022
 - examples.diffusion.mesh1D, 1023
 - examples.diffusion.mesh20x20, 1043
 - examples.diffusion.mesh20x20Coupled, 1046
 - examples.diffusion.nthOrder, 1049
 - examples.diffusion.nthOrder.input4thOrder1D, 1050
 - examples.diffusion.nthOrder.input4thOrder_line, 1052
 - examples.diffusion.nthOrder.test, 1052
 - examples.diffusion.steadyState, 1052
 - examples.diffusion.steadyState.mesh1D, 1052
 - examples.diffusion.steadyState.mesh1D.inputPeriodic, 1052

1053	1101
examples.diffusion.steadyState.mesh1D.tri2Dinput,	examples.levelSet.distanceFunction.interior,
1054	1103
examples.diffusion.steadyState.mesh20x20,	examples.levelSet.distanceFunction.mesh1D,
1054	1103
examples.diffusion.steadyState.mesh20x20.gmshinput,	examples.levelSet.distanceFunction.square,
1055	1104
examples.diffusion.steadyState.mesh20x20.isotropic,	examples.levelSet.distanceFunction.test, 1105
1055	examples.levelSet.electroChem, 1105
examples.diffusion.steadyState.mesh20x20.modifiedMeshInput,	examples.levelSet.electroChem.adsorbingSurfactantEquation,
1055	1106
examples.diffusion.steadyState.mesh20x20.orthogonal,	examples.levelSet.electroChem.adsorption,
1057	1106
examples.diffusion.steadyState.mesh20x20.tri2Dinput,	examples.levelSet.electroChem.gapFillDistanceVariable,
1057	1107
examples.diffusion.steadyState.mesh50x50,	examples.levelSet.electroChem.gapFillMesh,
1057	1107
examples.diffusion.steadyState.mesh50x50.input,	examples.levelSet.electroChem.gold, 1107
1057	examples.levelSet.electroChem.howToWriteAScript,
examples.diffusion.steadyState.mesh50x50.tri2Dinput,	1108
1058	examples.levelSet.electroChem.leveler, 1114
examples.diffusion.steadyState.otherMeshes,	examples.levelSet.electroChem.lines, 1118
1058	examples.levelSet.electroChem.matplotlibSurfactantViewer,
examples.diffusion.steadyState.otherMeshes.cubicalProblem,	1118
1058	examples.levelSet.electroChem.mayaviSurfactantViewer,
examples.diffusion.steadyState.otherMeshes.grid3Dinput,	1122
1059	examples.levelSet.electroChem.metalIonDiffusionEquation,
examples.diffusion.steadyState.otherMeshes.prism,	1125
1059	examples.levelSet.electroChem.simpleTrenchSystem,
examples.diffusion.steadyState.test, 1060	1125
examples.diffusion.test, 1060	examples.levelSet.electroChem.surfactantBulkDiffusionEquation,
examples.diffusion.variable, 1060	1128
examples.elphf, 1060	examples.levelSet.electroChem.test, 1128
examples.elphf.diffusion, 1062	examples.levelSet.electroChem.trenchMesh,
examples.elphf.diffusion.mesh1D, 1062	1128
examples.elphf.diffusion.mesh1Ddimensional,	examples.levelSet.surfactant, 1128
1064	examples.levelSet.surfactant.circle, 1129
examples.elphf.diffusion.mesh2D, 1067	examples.levelSet.surfactant.expandingCircle,
examples.elphf.input, 1070	1129
examples.elphf.phase, 1076	examples.levelSet.surfactant.square, 1130
examples.elphf.phaseDiffusion, 1079	examples.levelSet.surfactant.test, 1131
examples.elphf.poisson, 1087	examples.levelSet.test, 1131
examples.elphf.test, 1090	examples.meshing, 1131
examples.flow, 1090	examples.meshing.gmshRefinement, 1131
examples.flow.stokesCavity, 1091	examples.meshing.inputGrid2D, 1131
examples.flow.test, 1096	examples.meshing.sphere, 1132
examples.levelSet, 1096	examples.meshing.test, 1133
examples.levelSet.advection, 1096	examples.parallel, 1133
examples.levelSet.advection.circle, 1096	examples.phase, 1133
examples.levelSet.advection.mesh1D, 1098	examples.phase.anisotropy, 1134
examples.levelSet.advection.test, 1099	examples.phase.anisotropyOLD, 1138
examples.levelSet.advection.trench, 1099	examples.phase.binary, 1141
examples.levelSet.distanceFunction, 1101	examples.phase.binaryCoupled, 1155
examples.levelSet.distanceFunction.circle,	examples.phase.impingement, 1169

- examples.phase.impingement.mesh20x20, 1170
- examples.phase.impingement.mesh40x1, 1174
- examples.phase.impingement.test, 1177
- examples.phase.missOrientation, 1177
- examples.phase.missOrientation.circle, 1177
- examples.phase.missOrientation.mesh1D, 1178
- examples.phase.missOrientation.modCircle, 1178
- examples.phase.missOrientation.test, 1179
- examples.phase.phaseViewer, 1179
- examples.phase.polyxtal, 1182
- examples.phase.polyxtalCoupled, 1188
- examples.phase.quaternary, 1194
- examples.phase.simple, 1200
- examples.phase.symmetry, 1208
- examples.phase.test, 1210
- examples.reactiveWetting, 1210
- examples.reactiveWetting.liquidVapor1D, 1210
- examples.reactiveWetting.liquidVapor2D, 1216
- examples.reactiveWetting.test, 1220
- examples.riemann, 1220
- examples.riemann.acoustics, 1220
- examples.riemann.test, 1220
- examples.test, 1220
- examples.updating, 1220
- examples.updating.update0_1to1_0, 1221
- examples.updating.update1_0to2_0, 1225
- examples.updating.update2_0to3_0, 1229
- f**
- fiPy, 139
- fiPy.boundaryConditions, 140
- fiPy.boundaryConditions.boundaryCondition, 141
- fiPy.boundaryConditions.constraint, 141
- fiPy.boundaryConditions.fixedFlux, 142
- fiPy.boundaryConditions.fixedValue, 142
- fiPy.boundaryConditions.nthOrderBoundaryCondition, 143
- fiPy.boundaryConditions.test, 143
- fiPy.matrices, 144
- fiPy.matrices.offsetSparseMatrix, 144
- fiPy.matrices.petscMatrix, 144
- fiPy.matrices.scipyMatrix, 144
- fiPy.matrices.sparseMatrix, 144
- fiPy.matrices.test, 144
- fiPy.matrices.trilinosMatrix, 144
- fiPy.meshes, 144
- fiPy.meshes.abstractMesh, 146
- fiPy.meshes.builders, 153
- fiPy.meshes.builders.abstractGridBuilder, 154
- fiPy.meshes.builders.grid1DBuilder, 154
- fiPy.meshes.builders.grid2DBuilder, 154
- fiPy.meshes.builders.grid3DBuilder, 154
- fiPy.meshes.builders.periodicGrid1DBuilder, 154
- fiPy.meshes.builders.utilityClasses, 154
- fiPy.meshes.cylindricalGrid1D, 154
- fiPy.meshes.cylindricalGrid2D, 154
- fiPy.meshes.cylindricalNonUniformGrid1D, 154
- fiPy.meshes.cylindricalNonUniformGrid2D, 163
- fiPy.meshes.cylindricalUniformGrid1D, 172
- fiPy.meshes.cylindricalUniformGrid2D, 179
- fiPy.meshes.factoryMeshes, 186
- fiPy.meshes.gmshMesh, 189
- fiPy.meshes.grid1D, 237
- fiPy.meshes.grid2D, 237
- fiPy.meshes.grid3D, 237
- fiPy.meshes.mesh, 237
- fiPy.meshes.mesh1D, 246
- fiPy.meshes.mesh2D, 254
- fiPy.meshes.nonUniformGrid1D, 263
- fiPy.meshes.nonUniformGrid2D, 271
- fiPy.meshes.nonUniformGrid3D, 280
- fiPy.meshes.periodicGrid1D, 288
- fiPy.meshes.periodicGrid2D, 297
- fiPy.meshes.periodicGrid3D, 323
- fiPy.meshes.representations, 379
- fiPy.meshes.representations.abstractRepresentation, 379
- fiPy.meshes.representations.gridRepresentation, 379
- fiPy.meshes.representations.meshRepresentation, 379
- fiPy.meshes.skewedGrid2D, 379
- fiPy.meshes.sphericalNonUniformGrid1D, 388
- fiPy.meshes.sphericalUniformGrid1D, 397
- fiPy.meshes.test, 404
- fiPy.meshes.topologies, 404
- fiPy.meshes.topologies.abstractTopology, 404
- fiPy.meshes.topologies.gridTopology, 404
- fiPy.meshes.topologies.meshTopology, 404
- fiPy.meshes.tri2D, 404
- fiPy.meshes.uniformGrid, 413
- fiPy.meshes.uniformGrid1D, 420
- fiPy.meshes.uniformGrid2D, 427
- fiPy.meshes.uniformGrid3D, 434
- fiPy.solvers, 441
- fiPy.solvers.convergence, 443
- fiPy.solvers.petsc, 448
- fiPy.solvers.petsc.comms, 449
- fiPy.solvers.petsc.comms.parallelPETScCommWrapper, 449
- fiPy.solvers.petsc.comms.petscCommWrapper, 450
- fiPy.solvers.petsc.comms.serialPETScCommWrapper, 450
- fiPy.solvers.petsc.dummySolver, 451

[fipy.solvers.petsc.linearBicgSolver](#), 451
[fipy.solvers.petsc.linearCGSolver](#), 453
[fipy.solvers.petsc.linearCGSSolver](#), 452
[fipy.solvers.petsc.linearGMRESSolver](#), 454
[fipy.solvers.petsc.linearLUSolver](#), 455
[fipy.solvers.petsc.petscConvergence](#), 456
[fipy.solvers.petsc.petscKrylovSolver](#), 461
[fipy.solvers.petsc.petscSolver](#), 462
[fipy.solvers.petsc.preconditioners](#), 463
[fipy.solvers.petsc.preconditioners.defaultPreconditioner](#), 464
[fipy.solvers.petsc.preconditioners.hyprePreconditioner](#), 465
[fipy.solvers.petsc.preconditioners.icPreconditioner](#), 465
[fipy.solvers.petsc.preconditioners.iluPreconditioner](#), 465
[fipy.solvers.petsc.preconditioners.jacobiPreconditioner](#), 465
[fipy.solvers.petsc.preconditioners.luPreconditioner](#), 466
[fipy.solvers.petsc.preconditioners.petscPreconditioner](#), 466
[fipy.solvers.petsc.preconditioners.ssorPreconditioner](#), 466
[fipy.solvers.preconditioner](#), 466
[fipy.solvers.pyAMG](#), 467
[fipy.solvers.pyAMG.preconditioners](#), 467
[fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner](#), 467
[fipy.solvers.pyamgx](#), 468
[fipy.solvers.pyamgx.aggregationAMGSolver](#), 468
[fipy.solvers.pyamgx.classicalAMGSolver](#), 469
[fipy.solvers.pyamgx.linearBiCGStabSolver](#), 470
[fipy.solvers.pyamgx.linearCGSolver](#), 471
[fipy.solvers.pyamgx.linearFGMRESSolver](#), 472
[fipy.solvers.pyamgx.linearGMRESSolver](#), 473
[fipy.solvers.pyamgx.linearLUSolver](#), 474
[fipy.solvers.pyamgx.preconditioners](#), 475
[fipy.solvers.pyamgx.pyamgxConvergence](#), 478
[fipy.solvers.pyamgx.pyAMGXSolver](#), 477
[fipy.solvers.pyamgx.smoothers](#), 478
[fipy.solvers.scipy](#), 479
[fipy.solvers.scipy.linearBicgstabSolver](#), 479
[fipy.solvers.scipy.linearCGSolver](#), 481
[fipy.solvers.scipy.linearCGSSolver](#), 480
[fipy.solvers.scipy.linearGMRESSolver](#), 482
[fipy.solvers.scipy.linearLUSolver](#), 483
[fipy.solvers.scipy.preconditioners](#), 484
[fipy.solvers.scipy.preconditioners.iluPreconditioner](#), 484
[fipy.solvers.scipy.preconditioners.jacobiPreconditioner](#), 484
[fipy.solvers.scipy.preconditioners.scipyPreconditioner](#), 485
[fipy.solvers.scipy.scipyConvergence](#), 485
[fipy.solvers.scipy.scipyKrylovSolver](#), 486
[fipy.solvers.scipy.scipySolver](#), 487
[fipy.solvers.solver](#), 488
[fipy.solvers.test](#), 497
[fipy.solvers.trilinos](#), 497
[fipy.solvers.trilinos.aztecConvergence](#), 497
[fipy.solvers.trilinos.comms](#), 499
[fipy.solvers.trilinos.comms.epetraCommWrapper](#), 499
[fipy.solvers.trilinos.comms.parallelEpetraCommWrapper](#), 499
[fipy.solvers.trilinos.comms.serialEpetraCommWrapper](#), 499
[fipy.solvers.trilinos.linearBicgstabSolver](#), 501
[fipy.solvers.trilinos.linearCGSolver](#), 503
[fipy.solvers.trilinos.linearCGSSolver](#), 502
[fipy.solvers.trilinos.linearGMRESSolver](#), 504
[fipy.solvers.trilinos.linearLUSolver](#), 505
[fipy.solvers.trilinos.preconditioners](#), 505
[fipy.solvers.trilinos.preconditioners.domDecompPreconditioner](#), 507
[fipy.solvers.trilinos.preconditioners.icPreconditioner](#), 508
[fipy.solvers.trilinos.preconditioners.iluPreconditioner](#), 508
[fipy.solvers.trilinos.preconditioners.jacobiPreconditioner](#), 508
[fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner](#), 508
[fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner](#), 509
[fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner](#), 509
[fipy.solvers.trilinos.preconditioners.multilevelPreconditioner](#), 509
[fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner](#), 510
[fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner](#), 510
[fipy.solvers.trilinos.preconditioners.multilevelSolverSmoothers](#), 510
[fipy.solvers.trilinos.preconditioners.trilinosPreconditioner](#), 511
[fipy.solvers.trilinos.trilinosAztecOOSolver](#), 511
[fipy.solvers.trilinos.trilinosMLTest](#), 512
[fipy.solvers.trilinos.trilinosNonlinearSolver](#), 513
[fipy.solvers.trilinos.trilinosSolver](#), 515
[fipy.steps](#), 516

- fiPy.steps.pidStepper, 518
- fiPy.steps.pseudoRKQSSStepper, 518
- fiPy.steps.stepper, 518
- fiPy.terms, 518
- fiPy.terms.abstractBinaryTerm, 525
- fiPy.terms.abstractConvectionTerm, 525
- fiPy.terms.abstractDiffusionTerm, 525
- fiPy.terms.abstractUpwindConvectionTerm, 525
- fiPy.terms.advectionTerm, 525
- fiPy.terms.asymmetricConvectionTerm, 532
- fiPy.terms.binaryTerm, 532
- fiPy.terms.cellTerm, 532
- fiPy.terms.centralDiffConvectionTerm, 536
- fiPy.terms.coupledBinaryTerm, 541
- fiPy.terms.diffusionTerm, 541
- fiPy.terms.diffusionTermCorrection, 545
- fiPy.terms.diffusionTermNoCorrection, 548
- fiPy.terms.explicitDiffusionTerm, 552
- fiPy.terms.explicitSourceTerm, 555
- fiPy.terms.explicitUpwindConvectionTerm, 555
- fiPy.terms.exponentialConvectionTerm, 560
- fiPy.terms.faceTerm, 566
- fiPy.terms.firstOrderAdvectionTerm, 570
- fiPy.terms.hybridConvectionTerm, 575
- fiPy.terms.implicitDiffusionTerm, 580
- fiPy.terms.implicitSourceTerm, 580
- fiPy.terms.nonDiffusionTerm, 584
- fiPy.terms.powerLawConvectionTerm, 584
- fiPy.terms.residualTerm, 590
- fiPy.terms.sourceTerm, 594
- fiPy.terms.term, 598
- fiPy.terms.test, 601
- fiPy.terms.transientTerm, 601
- fiPy.terms.unaryTerm, 606
- fiPy.terms.upwindConvectionTerm, 606
- fiPy.terms.vanLeerConvectionTerm, 611
- fiPy.testFiPy, 616
- fiPy.tests, 616
- fiPy.tests.doctestPlus, 617
- fiPy.tests.lateImportTest, 619
- fiPy.tests.test, 619
- fiPy.tests.testProgram, 619
- fiPy.tools, 620
- fiPy.tools.comms, 636
- fiPy.tools.comms.commWrapper, 637
- fiPy.tools.comms.dummyComm, 637
- fiPy.tools.copy_script, 637
- fiPy.tools.debug, 638
- fiPy.tools.decorators, 638
- fiPy.tools.dimensions, 638
- fiPy.tools.dimensions.DictWithDefault, 639
- fiPy.tools.dimensions.NumberDict, 639
- fiPy.tools.dimensions.physicalField, 639
- fiPy.tools.dump, 663
- fiPy.tools.inline, 664
- fiPy.tools.logging, 664
- fiPy.tools.logging.environment, 665
- fiPy.tools.numerix, 666
- fiPy.tools.parser, 671
- fiPy.tools.sharedtempfile, 672
- fiPy.tools.test, 673
- fiPy.tools.timer, 673
- fiPy.tools.vector, 673
- fiPy.variables, 674
- fiPy.variables.addOverFacesVariable, 676
- fiPy.variables.arithmeticCellToFaceVariable, 676
- fiPy.variables.betaNoiseVariable, 676
- fiPy.variables.binaryOperatorVariable, 692
- fiPy.variables.cellToFaceVariable, 692
- fiPy.variables.cellVariable, 692
- fiPy.variables.constant, 707
- fiPy.variables.constraintMask, 707
- fiPy.variables.coupledCellVariable, 707
- fiPy.variables.distanceVariable, 707
- fiPy.variables.exponentialNoiseVariable, 725
- fiPy.variables.faceGradContributionsVariable, 741
- fiPy.variables.faceGradVariable, 741
- fiPy.variables.faceVariable, 741
- fiPy.variables.gammaNoiseVariable, 752
- fiPy.variables.gaussCellGradVariable, 768
- fiPy.variables.gaussianNoiseVariable, 768
- fiPy.variables.harmonicCellToFaceVariable, 784
- fiPy.variables.histogramVariable, 784
- fiPy.variables.interfaceAreaVariable, 798
- fiPy.variables.interfaceFlagVariable, 798
- fiPy.variables.leastSquaresCellGradVariable, 798
- fiPy.variables.levelSetDiffusionVariable, 798
- fiPy.variables.meshVariable, 798
- fiPy.variables.minmodCellToFaceVariable, 809
- fiPy.variables.modCellGradVariable, 809
- fiPy.variables.modCellToFaceVariable, 809
- fiPy.variables.modFaceGradVariable, 809
- fiPy.variables.modPhysicalField, 809
- fiPy.variables.modularVariable, 809
- fiPy.variables.noiseVariable, 824
- fiPy.variables.operatorVariable, 839
- fiPy.variables.scharfetterGummelFaceVariable, 839
- fiPy.variables.surfactantConvectionVariable, 850
- fiPy.variables.surfactantVariable, 861
- fiPy.variables.test, 877
- fiPy.variables.unaryOperatorVariable, 877
- fiPy.variables.uniformNoiseVariable, 877

- [fipy.variables.variable](#), 892
- [fipy.viewers](#), 901
- [fipy.viewers.matplotlibViewer](#), 905
- [fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer](#), 908
- [fipy.viewers.matplotlibViewer.abstractMatplotlibViewer](#), 910
- [fipy.viewers.matplotlibViewer.matplotlib1DViewer](#), 913
- [fipy.viewers.matplotlibViewer.matplotlib2DContourViewer](#), 915
- [fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer](#), 918
- [fipy.viewers.matplotlibViewer.matplotlib2DGridViewer](#), 921
- [fipy.viewers.matplotlibViewer.matplotlib2DViewer](#), 924
- [fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer](#), 926
- [fipy.viewers.matplotlibViewer.matplotlibStreamViewer](#), 927
- [fipy.viewers.matplotlibViewer.matplotlibVectorViewer](#), 931
- [fipy.viewers.matplotlibViewer.test](#), 934
- [fipy.viewers.mayaviViewer](#), 934
- [fipy.viewers.mayaviViewer.mayaviClient](#), 938
- [fipy.viewers.mayaviViewer.mayaviDaemon](#), 941
- [fipy.viewers.mayaviViewer.test](#), 942
- [fipy.viewers.multiViewer](#), 942
- [fipy.viewers.test](#), 943
- [fipy.viewers.testinteractive](#), 944
- [fipy.viewers.tsvViewer](#), 944
- [fipy.viewers.viewer](#), 946
- [fipy.viewers.vtkViewer](#), 948
- [fipy.viewers.vtkViewer.test](#), 952
- [fipy.viewers.vtkViewer.vtkCellViewer](#), 952
- [fipy.viewers.vtkViewer.vtkFaceViewer](#), 954
- [fipy.viewers.vtkViewer.vtkViewer](#), 956

p

- [package](#), 135
- [package.subpackage](#), 135
- [package.subpackage.base](#), 136
- [package.subpackage.object](#), 137

Index

Symbols

- `:math: '\pi'`, 1135, 1140, 1171, 1176
- `__abs__` () (*fipy.tools.PhysicalField* method), 621
- `__abs__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 642
- `__abs__` () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 678
- `__abs__` () (*fipy.variables.cellVariable.CellVariable* method), 692
- `__abs__` () (*fipy.variables.distanceVariable.DistanceVariable* method), 710
- `__abs__` () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 727
- `__abs__` () (*fipy.variables.faceVariable.FaceVariable* method), 741
- `__abs__` () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 753
- `__abs__` () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 770
- `__abs__` () (*fipy.variables.histogramVariable.HistogramVariable* method), 784
- `__abs__` () (*fipy.variables.meshVariable.MeshVariable* method), 799
- `__abs__` () (*fipy.variables.modularVariable.ModularVariable* method), 810
- `__abs__` () (*fipy.variables.noiseVariable.NoiseVariable* method), 825
- `__abs__` () (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 839
- `__abs__` () (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 851
- `__abs__` () (*fipy.variables.surfactantVariable.SurfactantVariable* method), 862
- `__abs__` () (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 878
- `__abs__` () (*fipy.variables.variable.Variable* method), 892
- `__add__` () (*fipy.meshes.abstractMesh.AbstractMesh* method), 146
- `__add__` () (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 155
- `__add__` () (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 163
- `__add__` () (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* method), 172
- `__add__` () (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* method), 179
- `__add__` () (*fipy.meshes.gmshMesh.Gmsh2D* method), 193
- `__add__` () (*fipy.meshes.gmshMesh.Gmsh2DIn3Dspace* method), 202
- `__add__` () (*fipy.meshes.gmshMesh.Gmsh3D* method), 210
- `__add__` () (*fipy.meshes.gmshMesh.GmshGrid2D* method), 219
- `__add__` () (*fipy.meshes.gmshMesh.GmshGrid3D* method), 227
- `__add__` () (*fipy.meshes.mesh.Mesh* method), 237
- `__add__` () (*fipy.meshes.mesh1D.Mesh1D* method), 246
- `__add__` () (*fipy.meshes.mesh2D.Mesh2D* method), 254
- `__add__` () (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 263
- `__add__` () (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 271
- `__add__` () (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 280
- `__add__` () (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 289
- `__add__` () (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 298
- `__add__` () (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 306
- `__add__` () (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 315
- `__add__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 325
- `__add__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 332
- `__add__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 340
- `__add__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 348
- `__add__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 356
- `__add__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 364
- `__add__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 371
- `__add__` () (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 380
- `__add__` () (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 389
- `__add__` () (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* method), 397
- `__add__` () (*fipy.meshes.tri2D.Tri2D* method), 405
- `__add__` () (*fipy.meshes.uniformGrid.UniformGrid* method), 413
- `__add__` () (*fipy.meshes.uniformGrid1D.UniformGrid1D* method), 420
- `__add__` () (*fipy.meshes.uniformGrid2D.UniformGrid2D* method), 427
- `__add__` () (*fipy.meshes.uniformGrid3D.UniformGrid3D* method), 435
- `__add__` () (*fipy.tools.PhysicalField* method), 621
- `__add__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 642

`__and__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 678
`__and__()` (*fipy.variables.cellVariable.CellVariable* method), 693
`__and__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 710
`__and__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 727
`__and__()` (*fipy.variables.faceVariable.FaceVariable* method), 741
`__and__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 754
`__and__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 770
`__and__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 784
`__and__()` (*fipy.variables.meshVariable.MeshVariable* method), 799
`__and__()` (*fipy.variables.modularVariable.ModularVariable* method), 810
`__and__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 825
`__and__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 839
`__and__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 851
`__and__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 862
`__and__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 878
`__and__()` (*fipy.variables.variable.Variable* method), 893
`__array__()` (*fipy.tools.PhysicalField* method), 621
`__array__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 643
`__array__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 678
`__array__()` (*fipy.variables.cellVariable.CellVariable* method), 693
`__array__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 710
`__array__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 727
`__array__()` (*fipy.variables.faceVariable.FaceVariable* method), 742
`__array__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 754
`__array__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 770
`__array__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 784
`__array__()` (*fipy.variables.meshVariable.MeshVariable* method), 799
`__array__()` (*fipy.variables.modularVariable.ModularVariable* method), 811
`__array__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 825
`__array__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 840
`__array__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 851
`__array__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 863
`__array__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 879
`__array__()` (*fipy.variables.variable.Variable* method), 893
`__bool__()` (*fipy.tools.PhysicalField* method), 621
`__bool__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 643
`__bool__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 679
`__bool__()` (*fipy.variables.cellVariable.CellVariable* method), 693
`__bool__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 711
`__bool__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 728
`__bool__()` (*fipy.variables.faceVariable.FaceVariable* method), 742
`__bool__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 754
`__bool__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 771
`__bool__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 785
`__bool__()` (*fipy.variables.meshVariable.MeshVariable* method), 800
`__bool__()` (*fipy.variables.modularVariable.ModularVariable* method), 811
`__bool__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 826
`__bool__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 840
`__bool__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 852
`__bool__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 863
`__bool__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 879
`__bool__()` (*fipy.variables.variable.Variable* method), 893
`__array_wrap__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 678
`__array_wrap__()` (*fipy.variables.cellVariable.CellVariable* method), 693
`__array_wrap__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 710
`__array_wrap__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 728
`__array_wrap__()` (*fipy.variables.faceVariable.FaceVariable* method), 742
`__array_wrap__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 754
`__array_wrap__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 771
`__array_wrap__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 785
`__array_wrap__()` (*fipy.variables.meshVariable.MeshVariable* method), 800
`__array_wrap__()` (*fipy.variables.modularVariable.ModularVariable* method), 811
`__array_wrap__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 826
`__array_wrap__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 840
`__array_wrap__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 852
`__array_wrap__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 863
`__array_wrap__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 879
`__array_wrap__()` (*fipy.variables.variable.Variable* method), 893

__bool__ () (fipy.variables.noiseVariable.NoiseVariable method), 826
 __bool__ () (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 840
 __bool__ () (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 852
 __bool__ () (fipy.variables.surfactantVariable.SurfactantVariable method), 863
 __bool__ () (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 879
 __bool__ () (fipy.variables.variable.Variable method), 893
 __call__ () (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 679
 __call__ () (fipy.variables.cellVariable.CellVariable method), 694
 __call__ () (fipy.variables.distanceVariable.DistanceVariable method), 711
 __call__ () (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 728
 __call__ () (fipy.variables.faceVariable.FaceVariable method), 742
 __call__ () (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 755
 __call__ () (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 771
 __call__ () (fipy.variables.histogramVariable.HistogramVariable method), 785
 __call__ () (fipy.variables.meshVariable.MeshVariable method), 800
 __call__ () (fipy.variables.modularVariable.ModularVariable method), 811
 __call__ () (fipy.variables.noiseVariable.NoiseVariable method), 826
 __call__ () (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 840
 __call__ () (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 852
 __call__ () (fipy.variables.surfactantVariable.SurfactantVariable method), 863
 __call__ () (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 879
 __call__ () (fipy.variables.variable.Variable method), 894
 __cause__ (fipy.meshes.abstractMesh.MeshAdditionError attribute), 153
 __cause__ (fipy.meshes.gmshMesh.GmshException attribute), 218
 __cause__ (fipy.meshes.gmshMesh.MeshExportError attribute), 236
 __cause__ (fipy.meshes.mesh.MeshAdditionError attribute), 245
 __cause__ (fipy.solvers.SerialSolverError attribute), 442
 __cause__ (fipy.solvers.convergence.DivergenceWarning attribute), 445
 __cause__ (fipy.solvers.solver.BreakdownWarning attribute), 488
 __cause__ (fipy.solvers.solver.IllConditionedPreconditionerWarning attribute), 489
 __cause__ (fipy.solvers.solver.IllegalInputOrBreakdownWarning attribute), 490
 __cause__ (fipy.solvers.solver.LossOfPrecisionWarning attribute), 490
 __cause__ (fipy.solvers.solver.MatrixIllConditionedWarning attribute), 491
 __cause__ (fipy.solvers.solver.MaximumIterationWarning attribute), 491
 __cause__ (fipy.solvers.solver.NormalConvergence attribute), 492
 __cause__ (fipy.solvers.solver.ParameterWarning attribute), 493
 __cause__ (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning attribute), 493
 __cause__ (fipy.solvers.solver.PreconditionerWarning attribute), 494
 __cause__ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning attribute), 494
 __cause__ (fipy.solvers.solver.SolverConvergenceWarning attribute), 496
 __cause__ (fipy.solvers.solver.StagnatedSolverWarning attribute), 496
 __cause__ (fipy.terms.AbstractBaseClassError attribute), 519
 __cause__ (fipy.terms.ExplicitVariableError attribute), 519
 __cause__ (fipy.terms.IncorrectSolutionVariable attribute), 520
 __cause__ (fipy.terms.SolutionVariableNumberError attribute), 521
 __cause__ (fipy.terms.SolutionVariableRequiredError attribute), 521
 __cause__ (fipy.terms.TermMultiplyError attribute), 522
 __cause__ (fipy.terms.TransientTermError attribute), 522
 __cause__ (fipy.terms.VectorCoeffError attribute), 523
 __cause__ (fipy.viewers.MeshDimensionError attribute), 903
 __delattr__ () (fipy.meshes.abstractMesh.MeshAdditionError method), 153
 __delattr__ () (fipy.meshes.gmshMesh.GmshException method), 218
 __delattr__ () (fipy.meshes.gmshMesh.MeshExportError method), 236
 __delattr__ () (fipy.meshes.mesh.MeshAdditionError method), 245
 __delattr__ () (fipy.solvers.SerialSolverError method), 442
 __delattr__ (fipy.solvers.solver.PreconditionerWarning attribute), 494
 __delattr__ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning attribute), 494
 __delattr__ (fipy.solvers.solver.SolverConvergenceWarning attribute), 496
 __delattr__ (fipy.solvers.solver.StagnatedSolverWarning attribute), 496
 __delattr__ (fipy.terms.AbstractBaseClassError attribute), 519
 __delattr__ (fipy.terms.ExplicitVariableError attribute), 519
 __delattr__ (fipy.terms.IncorrectSolutionVariable attribute), 520
 __delattr__ (fipy.terms.SolutionVariableNumberError attribute), 521
 __delattr__ (fipy.terms.SolutionVariableRequiredError attribute), 521
 __delattr__ (fipy.terms.TermMultiplyError attribute), 522
 __delattr__ (fipy.terms.TransientTermError attribute), 522
 __delattr__ (fipy.terms.VectorCoeffError attribute), 523
 __delattr__ (fipy.viewers.MeshDimensionError attribute), 903
 __delattr__ (fipy.viewers.MeshDimensionError attribute), 903

[__delattr__\(\)](#) (*fipy.solvers.convergence.DivergenceWarning* method), 445
[__delattr__\(\)](#) (*fipy.solvers.solver.BreakdownWarning* method), 489
[__delattr__\(\)](#) (*fipy.solvers.solver.IllConditionedPreconditionerWarning* method), 489
[__delattr__\(\)](#) (*fipy.solvers.solver.IllegalInputOrBreakdownWarning* method), 490
[__delattr__\(\)](#) (*fipy.solvers.solver.LossOfPrecisionWarning* method), 490
[__delattr__\(\)](#) (*fipy.solvers.solver.MatrixIllConditionedWarning* method), 491
[__delattr__\(\)](#) (*fipy.solvers.solver.MaximumIterationWarning* method), 491
[__delattr__\(\)](#) (*fipy.solvers.solver.NormalConvergence* method), 492
[__delattr__\(\)](#) (*fipy.solvers.solver.ParameterWarning* method), 493
[__delattr__\(\)](#) (*fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning* method), 493
[__delattr__\(\)](#) (*fipy.solvers.solver.PreconditionerWarning* method), 494
[__delattr__\(\)](#) (*fipy.solvers.solver.ScalarQuantityOutOfRangeWarning* method), 494
[__delattr__\(\)](#) (*fipy.solvers.solver.SolverConvergenceWarning* method), 496
[__delattr__\(\)](#) (*fipy.solvers.solver.StagnatedSolverWarning* method), 496
[__delattr__\(\)](#) (*fipy.terms.AbstractBaseClassError* method), 519
[__delattr__\(\)](#) (*fipy.terms.ExplicitVariableError* method), 520
[__delattr__\(\)](#) (*fipy.terms.IncorrectSolutionVariable* method), 520
[__delattr__\(\)](#) (*fipy.terms.SolutionVariableNumberError* method), 521
[__delattr__\(\)](#) (*fipy.terms.SolutionVariableRequiredError* method), 521
[__delattr__\(\)](#) (*fipy.terms.TermMultiplyError* method), 522
[__delattr__\(\)](#) (*fipy.terms.TransientTermError* method), 522
[__delattr__\(\)](#) (*fipy.terms.VectorCoeffError* method), 523
[__delattr__\(\)](#) (*fipy.viewers.MeshDimensionError* method), 903
[__div__\(\)](#) (*fipy.meshes.abstractMesh.AbstractMesh* method), 148
[__div__\(\)](#) (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 157
[__div__\(\)](#) (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 165
[__div__\(\)](#) (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* method), 174
[__div__\(\)](#) (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* method), 181
[__div__\(\)](#) (*fipy.meshes.gmshMesh.Gmsh2D* method), 195
[__div__\(\)](#) (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* method), 203
[__div__\(\)](#) (*fipy.meshes.gmshMesh.Gmsh3D* method), 212
[__div__\(\)](#) (*fipy.meshes.gmshMesh.GmshGrid2D* method), 220
[__div__\(\)](#) (*fipy.meshes.gmshMesh.GmshGrid3D* method), 229
[__div__\(\)](#) (*fipy.meshes.mesh.Mesh* method), 239
[__div__\(\)](#) (*fipy.meshes.mesh1D.Mesh1D* method), 248
[__div__\(\)](#) (*fipy.meshes.mesh2D.Mesh2D* method), 256
[__div__\(\)](#) (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 265
[__div__\(\)](#) (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 273
[__div__\(\)](#) (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 282
[__div__\(\)](#) (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 290
[__div__\(\)](#) (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 300
[__div__\(\)](#) (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 308
[__div__\(\)](#) (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 316
[__div__\(\)](#) (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 326
[__div__\(\)](#) (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 334
[__div__\(\)](#) (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 342
[__div__\(\)](#) (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 350
[__div__\(\)](#) (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 357
[__div__\(\)](#) (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 365
[__div__\(\)](#) (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 373
[__div__\(\)](#) (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 381
[__div__\(\)](#) (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 390
[__div__\(\)](#) (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* method), 399
[__div__\(\)](#) (*fipy.meshes.tri2D.Tri2D* method), 406
[__div__\(\)](#) (*fipy.meshes.uniformGrid.UniformGrid* method), 415
[__div__\(\)](#) (*fipy.meshes.uniformGrid1D.UniformGrid1D* method), 422
[__div__\(\)](#) (*fipy.meshes.uniformGrid2D.UniformGrid2D* method), 429
[__div__\(\)](#) (*fipy.meshes.uniformGrid3D.UniformGrid3D* method), 436
[__div__\(\)](#) (*fipy.tools.PhysicalField* method), 622
[__div__\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 643
[__div__\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 657
[__eq__\(\)](#) (*fipy.terms.advectionTerm.AdvectionTerm* method), 528
[__eq__\(\)](#) (*fipy.terms.cellTerm.CellTerm* method), 532
[__eq__\(\)](#) (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 537
[__eq__\(\)](#) (*fipy.terms.diffusionTerm.DiffusionTerm* method), 542
[__eq__\(\)](#) (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* method), 545
[__eq__\(\)](#) (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* method), 548
[__eq__\(\)](#) (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* method), 552
[__eq__\(\)](#) (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 557
[__eq__\(\)](#) (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 562
[__eq__\(\)](#) (*fipy.terms.faceTerm.FaceTerm* method), 566
[__eq__\(\)](#) (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 571
[__eq__\(\)](#) (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 576
[__eq__\(\)](#) (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 581

`__eq__()` (*fiPy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 586
`__eq__()` (*fiPy.terms.residualTerm.ResidualTerm* method), 590
`__eq__()` (*fiPy.terms.sourceTerm.SourceTerm* method), 594
`__eq__()` (*fiPy.terms.term.Term* method), 598
`__eq__()` (*fiPy.terms.transientTerm.TransientTerm* method), 602
`__eq__()` (*fiPy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 608
`__eq__()` (*fiPy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 613
`__eq__()` (*fiPy.tools.physicalField.physicalField* method), 622
`__eq__()` (*fiPy.tools.dimensions.physicalField.physicalField* method), 644
`__eq__()` (*fiPy.tools.dimensions.physicalField.physicalUnit* method), 657
`__eq__()` (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* method), 679
`__eq__()` (*fiPy.variables.cellVariable.CellVariable* method), 694
`__eq__()` (*fiPy.variables.distanceVariable.DistanceVariable* method), 711
`__eq__()` (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 729
`__eq__()` (*fiPy.variables.faceVariable.FaceVariable* method), 743
`__eq__()` (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 755
`__eq__()` (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 772
`__eq__()` (*fiPy.variables.histogramVariable.HistogramVariable* method), 786
`__eq__()` (*fiPy.variables.meshVariable.MeshVariable* method), 800
`__eq__()` (*fiPy.variables.modularVariable.ModularVariable* method), 812
`__eq__()` (*fiPy.variables.noiseVariable.NoiseVariable* method), 827
`__eq__()` (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 840
`__eq__()` (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 852
`__eq__()` (*fiPy.variables.surfactantVariable.SurfactantVariable* method), 864
`__eq__()` (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 880
`__eq__()` (*fiPy.variables.variable.Variable* method), 894
`__float__()` (*fiPy.tools.physicalField.physicalField* method), 622
`__float__()` (*fiPy.tools.dimensions.physicalField.physicalField* method), 644
`__ge__()` (*fiPy.tools.physicalField.physicalField* method), 622
`__ge__()` (*fiPy.tools.dimensions.physicalField.physicalField* method), 644
`__ge__()` (*fiPy.tools.dimensions.physicalField.physicalUnit* method), 658
`__ge__()` (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* method), 680
`__ge__()` (*fiPy.variables.cellVariable.CellVariable* method), 694
`__ge__()` (*fiPy.variables.distanceVariable.DistanceVariable* method), 712
`__ge__()` (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 729
`__ge__()` (*fiPy.variables.faceVariable.FaceVariable* method), 743
`__ge__()` (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 755
`__ge__()` (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 772
`__ge__()` (*fiPy.variables.histogramVariable.HistogramVariable* method), 786
`__ge__()` (*fiPy.variables.meshVariable.MeshVariable* method), 800
`__ge__()` (*fiPy.variables.modularVariable.ModularVariable* method), 812
`__ge__()` (*fiPy.variables.noiseVariable.NoiseVariable* method), 827
`__ge__()` (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 841
`__ge__()` (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 852
`__ge__()` (*fiPy.variables.surfactantVariable.SurfactantVariable* method), 864
`__ge__()` (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 880
`__ge__()` (*fiPy.variables.variable.Variable* method), 894
`__getattr__()` (*fiPy.meshes.abstractMesh.MeshAdditionError* method), 153
`__getattr__()` (*fiPy.meshes.gmshMesh.GmshException* method), 218
`__getattr__()` (*fiPy.meshes.gmshMesh.MeshExportError* method), 236
`__getattr__()` (*fiPy.meshes.mesh.MeshAdditionError* method), 245
`__getattr__()` (*fiPy.solvers.serialSolver.SerialSolverError* method), 442
`__getattr__()` (*fiPy.solvers.convergence.DivergenceWarning* method), 445
`__getattr__()` (*fiPy.solvers.solver.BreakdownWarning* method), 489
`__getattr__()` (*fiPy.solvers.solver.IllConditionedPreconditionerWarning* method), 489
`__getattr__()` (*fiPy.solvers.solver.IllegalInputOrBreakdownWarning* method), 490
`__getattr__()` (*fiPy.solvers.solver.LossOfPrecisionWarning* method), 490
`__getattr__()` (*fiPy.solvers.solver.MatrixIllConditionedWarning* method), 491
`__getattr__()` (*fiPy.solvers.solver.MaximumIterationWarning* method), 492
`__getattr__()` (*fiPy.solvers.solver.NormalConvergence* method), 492
`__getattr__()` (*fiPy.solvers.solver.ParameterWarning* method), 493
`__getattr__()` (*fiPy.solvers.solver.PreconditionerNotPositiveDefiniteWarning* method), 493
`__getattr__()` (*fiPy.solvers.solver.PreconditionerWarning* method), 494
`__getattr__()` (*fiPy.solvers.solver.ScalarQuantityOutOfRangeWarning* method), 494
`__getattr__()` (*fiPy.solvers.solver.SolverConvergenceWarning* method), 496
`__getattr__()` (*fiPy.solvers.solver.StagnatedSolverWarning* method), 496
`__getattr__()` (*fiPy.terms.AbstractBaseClassError* method), 519
`__getattr__()` (*fiPy.terms.ExplicitVariableError* method), 520
`__getattr__()` (*fiPy.terms.IncorrectSolutionVariable* method), 520
`__getattr__()` (*fiPy.terms.SolutionVariableNumberError* method), 521

`__getattr__` (fipy.terms.SolutionVariableRequiredError method), 521
`__getattr__` (fipy.terms.TermMultiplyError method), 522
`__getattr__` (fipy.terms.TransientTermError method), 523
`__getattr__` (fipy.terms.VectorCoeffError method), 523
`__getattr__` (fipy.viewers.MeshDimensionError method), 903
`__getitem__` (fipy.tools.PhysicalField method), 622
`__getitem__` (fipy.tools.dimensions.physicalField.PhysicalField method), 644
`__getitem__` (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 680
`__getitem__` (fipy.variables.cellVariable.CellVariable method), 695
`__getitem__` (fipy.variables.distanceVariable.DistanceVariable method), 712
`__getitem__` (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 729
`__getitem__` (fipy.variables.faceVariable.FaceVariable method), 743
`__getitem__` (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 756
`__getitem__` (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 772
`__getitem__` (fipy.variables.histogramVariable.HistogramVariable method), 786
`__getitem__` (fipy.variables.meshVariable.MeshVariable method), 801
`__getitem__` (fipy.variables.modularVariable.ModularVariable method), 812
`__getitem__` (fipy.variables.noiseVariable.NoiseVariable method), 827
`__getitem__` (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 841
`__getitem__` (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 853
`__getitem__` (fipy.variables.surfactantVariable.SurfactantVariable method), 865
`__getitem__` (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 880
`__getitem__` (fipy.variables.variable.Variable method), 894
`__getstate__` (fipy.meshes.abstractMesh.AbstractMesh method), 148
`__getstate__` (fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D method), 157
`__getstate__` (fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D method), 165
`__getstate__` (fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D method), 174
`__getstate__` (fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D method), 181
`__getstate__` (fipy.meshes.gmshMesh.Gmsh2D method), 195
`__getstate__` (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace method), 203
`__getstate__` (fipy.meshes.gmshMesh.Gmsh3D method), 212
`__getstate__` (fipy.meshes.gmshMesh.GmshGrid2D method), 221
`__getstate__` (fipy.meshes.gmshMesh.GmshGrid3D method), 229
`__getstate__` (fipy.meshes.mesh.Mesh method), 239
`__getstate__` (fipy.meshes.mesh1D.Mesh1D method), 248
`__getstate__` (fipy.meshes.mesh2D.Mesh2D method), 256
`__getstate__` (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D method), 265
`__getstate__` (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D method), 273
`__getstate__` (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D method), 282
`__getstate__` (fipy.meshes.periodicGrid1D.PeriodicGrid1D method), 290
`__getstate__` (fipy.meshes.periodicGrid2D.PeriodicGrid2D method), 300
`__getstate__` (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight method), 308
`__getstate__` (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom method), 317
`__getstate__` (fipy.meshes.periodicGrid3D.PeriodicGrid3D method), 326
`__getstate__` (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack method), 334
`__getstate__` (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight method), 342
`__getstate__` (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack method), 350
`__getstate__` (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom method), 358
`__getstate__` (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom method), 365
`__getstate__` (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack method), 373
`__getstate__` (fipy.meshes.skewedGrid2D.SkewedGrid2D method), 381
`__getstate__` (fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D method), 391
`__getstate__` (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D method), 399
`__getstate__` (fipy.meshes.tri2D.Tri2D method), 407
`__getstate__` (fipy.meshes.uniformGrid1D.UniformGrid1D method), 415
`__getstate__` (fipy.meshes.uniformGrid2D.UniformGrid2D method), 422
`__getstate__` (fipy.meshes.uniformGrid2D.UniformGrid2D method), 429
`__getstate__` (fipy.meshes.uniformGrid3D.UniformGrid3D method), 436
`__getstate__` (fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper method), 449
`__getstate__` (fipy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper method), 450
`__getstate__` (fipy.solvers.petsc.comms.serialPETScCommWrapper.SerialPETScCommWrapper method), 450
`__getstate__` (fipy.solvers.trilinos.comms.epetraCommWrapper.EpetraCommWrapper method), 450

method), 499

__getstate__() (fipy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper method), 500

__getstate__() (fipy.solvers.trilinos.comms.serialEpetraCommWrapper.SerialEpetraCommWrapper method), 500

__getstate__() (fipy.tools.comms.commWrapper.CommWrapper method), 637

__getstate__() (fipy.tools.comms.dummyComm.DummyComm method), 637

__getstate__() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 680

__getstate__() (fipy.variables.cellVariable.CellVariable method), 695

__getstate__() (fipy.variables.distanceVariable.DistanceVariable method), 712

__getstate__() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 729

__getstate__() (fipy.variables.faceVariable.FaceVariable method), 743

__getstate__() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 756

__getstate__() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 772

__getstate__() (fipy.variables.histogramVariable.HistogramVariable method), 786

__getstate__() (fipy.variables.meshVariable.MeshVariable method), 801

__getstate__() (fipy.variables.modularVariable.ModularVariable method), 813

__getstate__() (fipy.variables.noiseVariable.NoiseVariable method), 827

__getstate__() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 841

__getstate__() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 853

__getstate__() (fipy.variables.surfactantVariable.SurfactantVariable method), 865

__getstate__() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 880

__getstate__() (fipy.variables.variable.Variable method), 895

__gt__() (fipy.tools.PhysicalField method), 623

__gt__() (fipy.tools.dimensions.physicalField.PhysicalField method), 644

__gt__() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 658

__gt__() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 680

__gt__() (fipy.variables.cellVariable.CellVariable method), 695

__gt__() (fipy.variables.distanceVariable.DistanceVariable method), 712

__gt__() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 729

__gt__() (fipy.variables.faceVariable.FaceVariable method), 743

__gt__() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 756

__gt__() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 772

__gt__() (fipy.variables.histogramVariable.HistogramVariable method), 786

__gt__() (fipy.variables.meshVariable.MeshVariable method), 801

__gt__() (fipy.variables.modularVariable.ModularVariable method), 813

__gt__() (fipy.variables.noiseVariable.NoiseVariable method), 827

__gt__() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 841

__gt__() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 853

__gt__() (fipy.variables.surfactantVariable.SurfactantVariable method), 865

__gt__() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 880

__gt__() (fipy.variables.variable.Variable method), 895

__hash__() (fipy.tools.dimensions.physicalField.PhysicalUnit attribute), 658

__hash__() (fipy.terms.advectionTerm.AdvectionTerm method), 528

__hash__() (fipy.terms.cellTerm.CellTerm method), 532

__hash__() (fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm method), 538

__hash__() (fipy.terms.diffusionTerm.DiffusionTerm method), 542

__hash__() (fipy.terms.diffusionTermCorrection.DiffusionTermCorrection method), 545

__hash__() (fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection method), 549

__hash__() (fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm method), 552

__hash__() (fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm method), 557

__hash__() (fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm method), 562

__hash__() (fipy.terms.faceTerm.FaceTerm method), 566

__hash__() (fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm method), 571

__hash__() (fipy.terms.hybridConvectionTerm.HybridConvectionTerm method), 577

__hash__() (fipy.terms.implicitSourceTerm.ImplicitSourceTerm method), 581

__hash__() (fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm method), 586

__hash__() (fipy.terms.residualTerm.ResidualTerm method), 590

__hash__() (fipy.terms.sourceTerm.SourceTerm method), 594

__hash__() (fipy.terms.term.Term method), 598

__hash__() (fipy.terms.transientTerm.TransientTerm method), 602

__hash__() (fipy.terms.upwindConvectionTerm.UpwindConvectionTerm method), 608

__hash__() (fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm method), 613

__hash__() (fipy.tools.PhysicalField method), 623

__hash__() (fipy.tools.dimensions.physicalField.PhysicalField method), 645

__hash__() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 680

__hash__() (fipy.variables.cellVariable.CellVariable method), 695

<code>__hash__()</code> (<i>fipy.variables.distanceVariable.DistanceVariable</i> method), 712	<code>__le__()</code> (<i>fipy.tools.dimensions.physicalField.PhysicalUnit</i> method), 658
<code>__hash__()</code> (<i>fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable</i> method), 730	<code>__le__()</code> (<i>fipy.variables.betaNoiseVariable.BetaNoiseVariable</i> method), 681
<code>__hash__()</code> (<i>fipy.variables.faceVariable.FaceVariable</i> method), 744	<code>__le__()</code> (<i>fipy.variables.cellVariable.CellVariable</i> method), 695
<code>__hash__()</code> (<i>fipy.variables.gammaNoiseVariable.GammaNoiseVariable</i> method), 756	<code>__le__()</code> (<i>fipy.variables.distanceVariable.DistanceVariable</i> method), 713
<code>__hash__()</code> (<i>fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable</i> method), 773	<code>__le__()</code> (<i>fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable</i> method), 730
<code>__hash__()</code> (<i>fipy.variables.histogramVariable.HistogramVariable</i> method), 787	<code>__le__()</code> (<i>fipy.variables.faceVariable.FaceVariable</i> method), 744
<code>__hash__()</code> (<i>fipy.variables.meshVariable.MeshVariable</i> method), 801	<code>__le__()</code> (<i>fipy.variables.gammaNoiseVariable.GammaNoiseVariable</i> method), 756
<code>__hash__()</code> (<i>fipy.variables.modularVariable.ModularVariable</i> method), 813	<code>__le__()</code> (<i>fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable</i> method), 773
<code>__hash__()</code> (<i>fipy.variables.noiseVariable.NoiseVariable</i> method), 828	<code>__le__()</code> (<i>fipy.variables.histogramVariable.HistogramVariable</i> method), 787
<code>__hash__()</code> (<i>fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable</i> method), 842	<code>__le__()</code> (<i>fipy.variables.meshVariable.MeshVariable</i> method), 801
<code>__hash__()</code> (<i>fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable</i> method), 853	<code>__le__()</code> (<i>fipy.variables.modularVariable.ModularVariable</i> method), 813
<code>__hash__()</code> (<i>fipy.variables.surfactantVariable.SurfactantVariable</i> method), 865	<code>__le__()</code> (<i>fipy.variables.noiseVariable.NoiseVariable</i> method), 828
<code>__hash__()</code> (<i>fipy.variables.uniformNoiseVariable.UniformNoiseVariable</i> method), 881	<code>__le__()</code> (<i>fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable</i> method), 842
<code>__hash__()</code> (<i>fipy.variables.variable.Variable</i> method), 895	<code>__le__()</code> (<i>fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable</i> method), 853
<code>__invert__()</code> (<i>fipy.variables.betaNoiseVariable.BetaNoiseVariable</i> method), 681	<code>__le__()</code> (<i>fipy.variables.surfactantVariable.SurfactantVariable</i> method), 865
<code>__invert__()</code> (<i>fipy.variables.cellVariable.CellVariable</i> method), 695	<code>__le__()</code> (<i>fipy.variables.uniformNoiseVariable.UniformNoiseVariable</i> method), 881
<code>__invert__()</code> (<i>fipy.variables.distanceVariable.DistanceVariable</i> method), 713	<code>__le__()</code> (<i>fipy.variables.variable.Variable</i> method), 895
<code>__invert__()</code> (<i>fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable</i> method), 730	<code>__lt__()</code> (<i>fipy.tools.PhysicalField</i> method), 623
<code>__invert__()</code> (<i>fipy.variables.faceVariable.FaceVariable</i> method), 744	<code>__lt__()</code> (<i>fipy.tools.dimensions.physicalField.PhysicalField</i> method), 645
<code>__invert__()</code> (<i>fipy.variables.gammaNoiseVariable.GammaNoiseVariable</i> method), 756	<code>__lt__()</code> (<i>fipy.tools.dimensions.physicalField.PhysicalUnit</i> method), 658
<code>__invert__()</code> (<i>fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable</i> method), 773	<code>__lt__()</code> (<i>fipy.variables.betaNoiseVariable.BetaNoiseVariable</i> method), 681
<code>__invert__()</code> (<i>fipy.variables.histogramVariable.HistogramVariable</i> method), 787	<code>__lt__()</code> (<i>fipy.variables.cellVariable.CellVariable</i> method), 696
<code>__invert__()</code> (<i>fipy.variables.meshVariable.MeshVariable</i> method), 801	<code>__lt__()</code> (<i>fipy.variables.distanceVariable.DistanceVariable</i> method), 713
<code>__invert__()</code> (<i>fipy.variables.modularVariable.ModularVariable</i> method), 813	<code>__lt__()</code> (<i>fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable</i> method), 730
<code>__invert__()</code> (<i>fipy.variables.noiseVariable.NoiseVariable</i> method), 828	<code>__lt__()</code> (<i>fipy.variables.faceVariable.FaceVariable</i> method), 744
<code>__invert__()</code> (<i>fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable</i> method), 842	<code>__lt__()</code> (<i>fipy.variables.gammaNoiseVariable.GammaNoiseVariable</i> method), 757
<code>__invert__()</code> (<i>fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable</i> method), 853	<code>__lt__()</code> (<i>fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable</i> method), 773
<code>__invert__()</code> (<i>fipy.variables.surfactantVariable.SurfactantVariable</i> method), 865	<code>__lt__()</code> (<i>fipy.variables.histogramVariable.HistogramVariable</i> method), 787
<code>__invert__()</code> (<i>fipy.variables.uniformNoiseVariable.UniformNoiseVariable</i> method), 881	<code>__lt__()</code> (<i>fipy.variables.meshVariable.MeshVariable</i> method), 802
<code>__invert__()</code> (<i>fipy.variables.variable.Variable</i> method), 895	<code>__lt__()</code> (<i>fipy.variables.modularVariable.ModularVariable</i> method), 813
<code>__le__()</code> (<i>fipy.tools.PhysicalField</i> method), 623	<code>__lt__()</code> (<i>fipy.variables.noiseVariable.NoiseVariable</i> method), 828
<code>__le__()</code> (<i>fipy.tools.dimensions.physicalField.PhysicalField</i> method), 645	<code>__lt__()</code> (<i>fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable</i> method), 842
	<code>__lt__()</code> (<i>fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable</i> method), 854
	<code>__lt__()</code> (<i>fipy.variables.surfactantVariable.SurfactantVariable</i> method), 866
	<code>__lt__()</code> (<i>fipy.variables.uniformNoiseVariable.UniformNoiseVariable</i> method), 881

- `__lt__()` (*fipy.variables.variable.Variable* method), 895
- `__mod__()` (*fipy.tools.PhysicalField* method), 623
- `__mod__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 645
- `__mul__()` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 157
- `__mul__()` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 165
- `__mul__()` (*fipy.meshes.gmshMesh.Gmsh2D* method), 195
- `__mul__()` (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* method), 203
- `__mul__()` (*fipy.meshes.gmshMesh.Gmsh3D* method), 212
- `__mul__()` (*fipy.meshes.gmshMesh.GmshGrid2D* method), 221
- `__mul__()` (*fipy.meshes.gmshMesh.GmshGrid3D* method), 229
- `__mul__()` (*fipy.meshes.mesh.Mesh* method), 239
- `__mul__()` (*fipy.meshes.mesh1D.Mesh1D* method), 248
- `__mul__()` (*fipy.meshes.mesh2D.Mesh2D* method), 256
- `__mul__()` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 265
- `__mul__()` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 273
- `__mul__()` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 282
- `__mul__()` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 290
- `__mul__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 300
- `__mul__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 308
- `__mul__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 317
- `__mul__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 326
- `__mul__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 334
- `__mul__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 342
- `__mul__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 350
- `__mul__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 358
- `__mul__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 365
- `__mul__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 373
- `__mul__()` (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 381
- `__mul__()` (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 391
- `__mul__()` (*fipy.meshes.tri2D.Tri2D* method), 407
- `__mul__()` (*fipy.terms.advectionTerm.AdvectionTerm* method), 528
- `__mul__()` (*fipy.terms.cellTerm.CellTerm* method), 532
- `__mul__()` (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 538
- `__mul__()` (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 557
- `__mul__()` (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 562
- `__mul__()` (*fipy.terms.faceTerm.FaceTerm* method), 566
- `__mul__()` (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 571
- `__mul__()` (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 577
- `__mul__()` (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 581
- `__mul__()` (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 586
- `__mul__()` (*fipy.terms.residualTerm.ResidualTerm* method), 590
- `__mul__()` (*fipy.terms.sourceTerm.SourceTerm* method), 594
- `__mul__()` (*fipy.terms.transientTerm.TransientTerm* method), 603
- `__mul__()` (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 608
- `__mul__()` (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 613
- `__mul__()` (*fipy.tools.PhysicalField* method), 623
- `__mul__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 645
- `__mul__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 658
- `__ne__()` (*fipy.tools.PhysicalField* method), 624
- `__ne__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 645
- `__ne__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 659
- `__ne__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 681
- `__ne__()` (*fipy.variables.cellVariable.CellVariable* method), 696
- `__ne__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 713
- `__ne__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 730
- `__ne__()` (*fipy.variables.faceVariable.FaceVariable* method), 745
- `__ne__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 757
- `__ne__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 774
- `__ne__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 788
- `__ne__()` (*fipy.variables.meshVariable.MeshVariable* method), 802
- `__ne__()` (*fipy.variables.modularVariable.ModularVariable* method), 814
- `__ne__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 829
- `__ne__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 842
- `__ne__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 854
- `__ne__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 866
- `__ne__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 882
- `__ne__()` (*fipy.variables.variable.Variable* method), 896
- `__neg__()` (*fipy.terms.advectionTerm.AdvectionTerm* method), 528
- `__neg__()` (*fipy.terms.cellTerm.CellTerm* method), 533
- `__neg__()` (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 538
- `__neg__()` (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 557
- `__neg__()` (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 562
- `__neg__()` (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 571
- `__neg__()` (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 577
- `__neg__()` (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 581
- `__neg__()` (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 586
- `__neg__()` (*fipy.terms.residualTerm.ResidualTerm* method), 590
- `__neg__()` (*fipy.terms.sourceTerm.SourceTerm* method), 594
- `__neg__()` (*fipy.terms.transientTerm.TransientTerm* method), 603
- `__neg__()` (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 608
- `__neg__()` (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 613
- `__neg__()` (*fipy.tools.PhysicalField* method), 623
- `__neg__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 645
- `__neg__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 658
- `__neg__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 681
- `__neg__()` (*fipy.variables.cellVariable.CellVariable* method), 696
- `__neg__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 713
- `__neg__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 730
- `__neg__()` (*fipy.variables.faceVariable.FaceVariable* method), 745
- `__neg__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 757
- `__neg__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 774
- `__neg__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 788
- `__neg__()` (*fipy.variables.meshVariable.MeshVariable* method), 802
- `__neg__()` (*fipy.variables.modularVariable.ModularVariable* method), 814
- `__neg__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 829
- `__neg__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 842
- `__neg__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 854
- `__neg__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 866
- `__neg__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 882
- `__neg__()` (*fipy.variables.variable.Variable* method), 896

method), 562
`__neg__()` (*fipy.terms.faceTerm.FaceTerm* method), 567
`__neg__()` (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 572
`__neg__()` (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 577
`__neg__()` (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 581
`__neg__()` (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 586
`__neg__()` (*fipy.terms.residualTerm.ResidualTerm* method), 590
`__neg__()` (*fipy.terms.sourceTerm.SourceTerm* method), 595
`__neg__()` (*fipy.terms.transientTerm.TransientTerm* method), 603
`__neg__()` (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 608
`__neg__()` (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 613
`__neg__()` (*fipy.tools.PhysicalField* method), 624
`__neg__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 646
`__new__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* static method), 682
`__new__()` (*fipy.variables.cellVariable.CellVariable* static method), 696
`__new__()` (*fipy.variables.distanceVariable.DistanceVariable* static method), 714
`__new__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* static method), 731
`__new__()` (*fipy.variables.faceVariable.FaceVariable* static method), 745
`__new__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* static method), 757
`__new__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* static method), 774
`__new__()` (*fipy.variables.histogramVariable.HistogramVariable* static method), 788
`__new__()` (*fipy.variables.meshVariable.MeshVariable* static method), 802
`__new__()` (*fipy.variables.modularVariable.ModularVariable* static method), 814
`__new__()` (*fipy.variables.noiseVariable.NoiseVariable* static method), 829
`__new__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* static method), 843
`__new__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* static method), 854
`__new__()` (*fipy.variables.surfactantVariable.SurfactantVariable* static method), 866
`__new__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* static method), 882
`__new__()` (*fipy.variables.variable.Variable* static method), 896
`__nonzero__()` (*fipy.tools.PhysicalField* method), 624
`__nonzero__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 646
`__nonzero__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 682
`__nonzero__()` (*fipy.variables.cellVariable.CellVariable* method), 696
`__nonzero__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 714
`__nonzero__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 731
`__nonzero__()` (*fipy.variables.faceVariable.FaceVariable* method), 745
`__nonzero__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 757
`__nonzero__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 774
`__nonzero__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 788
`__nonzero__()` (*fipy.variables.meshVariable.MeshVariable* method), 802
`__nonzero__()` (*fipy.variables.modularVariable.ModularVariable* method), 814
`__nonzero__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 829
`__nonzero__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 843
`__nonzero__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 854
`__nonzero__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 866
`__nonzero__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 882
`__nonzero__()` (*fipy.variables.variable.Variable* method), 896
`__or__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 682
`__or__()` (*fipy.variables.cellVariable.CellVariable* method), 697
`__or__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 714
`__or__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 731
`__or__()` (*fipy.variables.faceVariable.FaceVariable* method), 745
`__or__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 758
`__or__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 774
`__or__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 788
`__or__()` (*fipy.variables.meshVariable.MeshVariable* method), 803
`__or__()` (*fipy.variables.modularVariable.ModularVariable* method), 814
`__or__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 829
`__or__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 843
`__or__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 855
`__or__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 866
`__or__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 882
`__or__()` (*fipy.variables.variable.Variable* method), 896
`__pow__()` (*fipy.tools.PhysicalField* method), 624
`__pow__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 646
`__pow__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 659

__pow__ (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 682
__pow__ (fipy.variables.cellVariable.CellVariable method), 697
__pow__ (fipy.variables.distanceVariable.DistanceVariable method), 714
__pow__ (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 731
__pow__ (fipy.variables.faceVariable.FaceVariable method), 745
__pow__ (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 758
__pow__ (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 774
__pow__ (fipy.variables.histogramVariable.HistogramVariable method), 788
__pow__ (fipy.variables.meshVariable.MeshVariable method), 803
__pow__ (fipy.variables.modularVariable.ModularVariable method), 815
__pow__ (fipy.variables.noiseVariable.NoiseVariable method), 829
__pow__ (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 843
__pow__ (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 855
__pow__ (fipy.variables.surfactantVariable.SurfactantVariable method), 867
__pow__ (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 882
__pow__ (fipy.variables.variable.Variable method), 897
__radd__ (fipy.meshes.abstractMesh.AbstractMesh method), 148
__radd__ (fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D method), 157
__radd__ (fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D method), 165
__radd__ (fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D method), 174
__radd__ (fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D method), 181
__radd__ (fipy.meshes.gmshMesh.Gmsh2D method), 195
__radd__ (fipy.meshes.gmshMesh.Gmsh2DIn3Dspace method), 204
__radd__ (fipy.meshes.gmshMesh.Gmsh3D method), 213
__radd__ (fipy.meshes.gmshMesh.GmshGrid2D method), 221
__radd__ (fipy.meshes.gmshMesh.GmshGrid3D method), 229
__radd__ (fipy.meshes.mesh.Mesh method), 240
__radd__ (fipy.meshes.mesh1D.Mesh1D method), 248
__radd__ (fipy.meshes.mesh2D.Mesh2D method), 256
__radd__ (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D method), 266
__radd__ (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D method), 274
__radd__ (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D method), 282
__radd__ (fipy.meshes.periodicGrid1D.PeriodicGrid1D method), 291
__radd__ (fipy.meshes.periodicGrid2D.PeriodicGrid2D method), 300
__radd__ (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight method), 309
__radd__ (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom method), 317
__radd__ (fipy.meshes.periodicGrid3D.PeriodicGrid3D method), 327
__radd__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack method), 335
__radd__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight method), 343
__radd__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack method), 350
__radd__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom method), 358
__radd__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom method), 366
__radd__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack method), 374
__radd__ (fipy.meshes.skewedGrid2D.SkewedGrid2D method), 382
__radd__ (fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D method), 391
__radd__ (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D method), 399
__radd__ (fipy.meshes.tri2D.Tri2D method), 407
__radd__ (fipy.meshes.uniformGrid.UniformGrid method), 415
__radd__ (fipy.meshes.uniformGrid1D.UniformGrid1D method), 422
__radd__ (fipy.meshes.uniformGrid2D.UniformGrid2D method), 429
__radd__ (fipy.meshes.uniformGrid3D.UniformGrid3D method), 436
__radd__ (fipy.tools.physicalField.physicalField method), 624
__radd__ (fipy.tools.dimensions.physicalField.PhysicalField method), 646
__rdiv__ (fipy.tools.dimensions.physicalField.PhysicalUnit method), 659
__reduce__ (fipy.meshes.abstractMesh.MeshAdditionError method), 153
__reduce__ (fipy.meshes.gmshMesh.GmshException method), 218
__reduce__ (fipy.meshes.gmshMesh.MeshExportError method), 236
__reduce__ (fipy.meshes.mesh.MeshAdditionError method), 245
__reduce__ (fipy.solvers.SerialSolverError method), 442
__reduce__ (fipy.solvers.convergence.DivergenceWarning method), 445
__reduce__ (fipy.solvers.solver.BreakdownWarning method), 489
__reduce__ (fipy.solvers.solver.IllConditionedPreconditionerWarning method), 489
__reduce__ (fipy.solvers.solver.IllegalInputOrBreakdownWarning method), 490
__reduce__ (fipy.solvers.solver.LossOfPrecisionWarning method), 490
__reduce__ (fipy.solvers.solver.MatrixIllConditionedWarning method), 491
__reduce__ (fipy.solvers.solver.MaximumIterationWarning method), 492
__reduce__ (fipy.solvers.solver.NormalConvergence method), 492
__reduce__ (fipy.solvers.solver.ParameterWarning method), 493
__reduce__ (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning method), 493
__reduce__ (fipy.solvers.solver.PreconditionerWarning method), 494
__reduce__ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning method), 494

`__reduce__()` (*fipy.solvers.solver.SolverConvergenceWarning* method), 496
`__reduce__()` (*fipy.solvers.solver.StagnatedSolverWarning* method), 496
`__reduce__()` (*fipy.terms.AbstractBaseClassError* method), 519
`__reduce__()` (*fipy.terms.ExplicitVariableError* method), 520
`__reduce__()` (*fipy.terms.IncorrectSolutionVariable* method), 520
`__reduce__()` (*fipy.terms.SolutionVariableNumberError* method), 521
`__reduce__()` (*fipy.terms.SolutionVariableRequiredError* method), 521
`__reduce__()` (*fipy.terms.TermMultiplyError* method), 522
`__reduce__()` (*fipy.terms.TransientTermError* method), 523
`__reduce__()` (*fipy.terms.VectorCoeffError* method), 523
`__reduce__()` (*fipy.viewers.MeshDimensionError* method), 903
`__repr__()` (*fipy.boundaryConditions.boundaryCondition.BoundaryCondition* method), 141
`__repr__()` (*fipy.boundaryConditions.constraint.Constraint* method), 141
`__repr__()` (*fipy.boundaryConditions.fixedFlux.FixedFlux* method), 142
`__repr__()` (*fipy.boundaryConditions.fixedValue.FixedValue* method), 143
`__repr__()` (*fipy.boundaryConditions.nthOrderBoundaryCondition.NthOrderBoundaryCondition* method), 143
`__repr__()` (*fipy.meshes.abstractMesh.AbstractMesh* method), 150
`__repr__()` (*fipy.meshes.abstractMesh.MeshAdditionError* method), 153
`__repr__()` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 159
`__repr__()` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 167
`__repr__()` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* method), 175
`__repr__()` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* method), 183
`__repr__()` (*fipy.meshes.gmshMesh.Gmsh2D* method), 197
`__repr__()` (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* method), 206
`__repr__()` (*fipy.meshes.gmshMesh.Gmsh3D* method), 214
`__repr__()` (*fipy.meshes.gmshMesh.GmshException* method), 218
`__repr__()` (*fipy.meshes.gmshMesh.GmshGrid2D* method), 223
`__repr__()` (*fipy.meshes.gmshMesh.GmshGrid3D* method), 231
`__repr__()` (*fipy.meshes.gmshMesh.MeshExportError* method), 236
`__repr__()` (*fipy.meshes.mesh.Mesh* method), 241
`__repr__()` (*fipy.meshes.mesh.MeshAdditionError* method), 245
`__repr__()` (*fipy.meshes.mesh1D.Mesh1D* method), 250
`__repr__()` (*fipy.meshes.mesh2D.Mesh2D* method), 258
`__repr__()` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 267
`__repr__()` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 275
`__repr__()` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 284
`__repr__()` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 293
`__repr__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 302
`__repr__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 310
`__repr__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 319
`__repr__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 329
`__repr__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 336
`__repr__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 344
`__repr__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 352
`__repr__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 360
`__repr__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 367
`__repr__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 375
`__repr__()` (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 384
`__repr__()` (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 393
`__repr__()` (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* method), 400
`__repr__()` (*fipy.meshes.tri2D.Tri2D* method), 409
`__repr__()` (*fipy.meshes.uniformGrid.UniformGrid* method), 417
`__repr__()` (*fipy.meshes.uniformGrid1D.UniformGrid1D* method), 424
`__repr__()` (*fipy.meshes.uniformGrid2D.UniformGrid2D* method), 431
`__repr__()` (*fipy.meshes.uniformGrid3D.UniformGrid3D* method), 438
`__repr__()` (*fipy.solvers.SerialSolverError* method), 442
`__repr__()` (*fipy.solvers.convergence.DivergenceWarning* method), 446
`__repr__()` (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 450
`__repr__()` (*fipy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper* method), 450
`__repr__()` (*fipy.solvers.petsc.comms.serialPETScCommWrapper.SerialPETScCommWrapper* method), 450
`__repr__()` (*fipy.solvers.petsc.dummySolver.DummySolver* method), 451
`__repr__()` (*fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver* method), 452
`__repr__()` (*fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver* method), 453
`__repr__()` (*fipy.solvers.petsc.linearCGSolver.LinearCGSolver* method), 454
`__repr__()` (*fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver* method), 455
`__repr__()` (*fipy.solvers.petsc.linearLUSolver.LinearLUSolver* method), 456
`__repr__()` (*fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver* method), 462
`__repr__()` (*fipy.solvers.petsc.petscSolver.PETScSolver* method), 463
`__repr__()` (*fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver* method), 469
`__repr__()` (*fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver* method), 470
`__repr__()` (*fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver* method), 471
`__repr__()` (*fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver* method), 471

```

method), 472
__repr__ ( (fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver
method), 473
__repr__ ( (fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver
method), 474
__repr__ ( (fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver
method), 474
__repr__ ( (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver
method), 477
__repr__ ( (fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver
method), 480
__repr__ ( (fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver
method), 481
__repr__ ( (fipy.solvers.scipy.linearCGSolver.LinearCGSolver
method), 481
__repr__ ( (fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver
method), 482
__repr__ ( (fipy.solvers.scipy.linearLUSolver.LinearLUSolver
method), 483
__repr__ ( (fipy.solvers.scipy.scipyKrylovSolver.ScipyKrylovSolver
method), 486
__repr__ ( (fipy.solvers.scipy.scipySolver.ScipySolver method), 487
__repr__ ( (fipy.solvers.solver.BreakdownWarning method), 489
__repr__ ( (fipy.solvers.solver.IllConditionedPreconditionerWarning
method), 489
__repr__ ( (fipy.solvers.solver.IllegalInputOrBreakdownWarning
method), 490
__repr__ ( (fipy.solvers.solver.LossOfPrecisionWarning method), 490
__repr__ ( (fipy.solvers.solver.MatrixIllConditionedWarning
method), 491
__repr__ ( (fipy.solvers.solver.MaximumIterationWarning method),
492
__repr__ ( (fipy.solvers.solver.NormalConvergence method), 492
__repr__ ( (fipy.solvers.solver.ParameterWarning method), 493
__repr__ ( (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning
method), 493
__repr__ ( (fipy.solvers.solver.PreconditionerWarning method), 494
__repr__ ( (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning
method), 495
__repr__ ( (fipy.solvers.solver.Solver method), 495
__repr__ ( (fipy.solvers.solver.SolverConvergenceWarning method),
496
__repr__ ( (fipy.solvers.solver.StagnatedSolverWarning method), 496
__repr__ ( (fipy.solvers.trilinos.comms.epetraCommWrapper.EpetraCommWrapper
method), 499
__repr__ ( (fipy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper
method), 500
__repr__ ( (fipy.solvers.trilinos.comms.serialEpetraCommWrapper.SerialEpetraCommWrapper
method), 500
__repr__ ( (fipy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver
method), 501
__repr__ ( (fipy.solvers.trilinos.linearCGSSolver.LinearCGSSolver
method), 502
__repr__ ( (fipy.solvers.trilinos.linearCGSolver.LinearCGSolver
method), 503
__repr__ ( (fipy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver
method), 504
__repr__ ( (fipy.solvers.trilinos.linearLUSolver.LinearLUSolver
method), 505
__repr__ ( (fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver
method), 512
__repr__ ( (fipy.solvers.trilinos.trilinosMLTest.TrilinosMLTest
method), 513
__repr__ ( (fipy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver
method), 514
__repr__ ( (fipy.solvers.trilinos.trilinosSolver.TrilinosSolver method),
516
__repr__ ( (fipy.terms.AbstractBaseClassError method), 519
__repr__ ( (fipy.terms.ExplicitVariableError method), 520
__repr__ ( (fipy.terms.IncorrectSolutionVariable method), 520
__repr__ ( (fipy.terms.SolutionVariableNumberError method), 521
__repr__ ( (fipy.terms.SolutionVariableRequiredError method), 521
__repr__ ( (fipy.terms.TermMultiplyError method), 522
__repr__ ( (fipy.terms.TransientTermError method), 523
__repr__ ( (fipy.terms.VectorCoeffError method), 523
__repr__ ( (fipy.terms.advectionTerm.AdvectionTerm method), 528
__repr__ ( (fipy.terms.cellTerm.CellTerm method), 533
__repr__ ( (fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm
method), 538
__repr__ ( (fipy.terms.diffusionTerm.DiffusionTerm method), 542
__repr__ ( (fipy.terms.diffusionTermCorrection.DiffusionTermCorrection
method), 545
__repr__ ( (fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection
method), 549
__repr__ ( (fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm
method), 552
__repr__ ( (fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm
method), 557
__repr__ ( (fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm
method), 562
__repr__ ( (fipy.terms.faceTerm.FaceTerm method), 567
__repr__ ( (fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm
method), 572
__repr__ ( (fipy.terms.hybridConvectionTerm.HybridConvectionTerm
method), 577
__repr__ ( (fipy.terms.implicitSourceTerm.ImplicitSourceTerm
method), 581
__repr__ ( (fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm
method), 586
__repr__ ( (fipy.terms.residualTerm.ResidualTerm method), 590
__repr__ ( (fipy.terms.sourceTerm.SourceTerm method), 595
__repr__ ( (fipy.terms.term.Term method), 598
__repr__ ( (fipy.terms.transientTerm.TransientTerm method), 603
__repr__ ( (fipy.terms.upwindConvectionTerm.UpwindConvectionTerm
method), 608
__repr__ ( (fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm
method), 613
__repr__ ( (fipy.tools.PhysicalField method), 624
__repr__ ( (fipy.tools.comms.commWrapper.CommWrapper method),
637
__repr__ ( (fipy.tools.comms.dummyComm.DummyComm method),
637
__repr__ ( (fipy.tools.dimensions.physicalField.PhysicalField
method), 646

```

`__repr__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 659
`__repr__` () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 682
`__repr__` () (*fipy.variables.cellVariable.CellVariable* method), 697
`__repr__` () (*fipy.variables.distanceVariable.DistanceVariable* method), 714
`__repr__` () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 731
`__repr__` () (*fipy.variables.faceVariable.FaceVariable* method), 745
`__repr__` () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 758
`__repr__` () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 774
`__repr__` () (*fipy.variables.histogramVariable.HistogramVariable* method), 789
`__repr__` () (*fipy.variables.meshVariable.MeshVariable* method), 803
`__repr__` () (*fipy.variables.modularVariable.ModularVariable* method), 815
`__repr__` () (*fipy.variables.noiseVariable.NoiseVariable* method), 829
`__repr__` () (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 843
`__repr__` () (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 855
`__repr__` () (*fipy.variables.surfactantVariable.SurfactantVariable* method), 867
`__repr__` () (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 882
`__repr__` () (*fipy.variables.variable.Variable* method), 897
`__repr__` () (*fipy.viewers.MeshDimensionError* method), 904
`__rmul__` () (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 159
`__rmul__` () (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 167
`__rmul__` () (*fipy.meshes.gmshMesh.Gmsh2D* method), 197
`__rmul__` () (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* method), 206
`__rmul__` () (*fipy.meshes.gmshMesh.Gmsh3D* method), 214
`__rmul__` () (*fipy.meshes.gmshMesh.GmshGrid2D* method), 223
`__rmul__` () (*fipy.meshes.gmshMesh.GmshGrid3D* method), 231
`__rmul__` () (*fipy.meshes.mesh.Mesh* method), 241
`__rmul__` () (*fipy.meshes.mesh1D.Mesh1D* method), 250
`__rmul__` () (*fipy.meshes.mesh2D.Mesh2D* method), 258
`__rmul__` () (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 267
`__rmul__` () (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 275
`__rmul__` () (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 284
`__rmul__` () (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 293
`__rmul__` () (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 302
`__rmul__` () (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 310
`__rmul__` () (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 319
`__rmul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 329
`__rmul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 337
`__rmul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 344
`__rmul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 352
`__rmul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 360
`__rmul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 368
`__rmul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 375
`__rmul__` () (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 384
`__rmul__` () (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 393
`__rmul__` () (*fipy.meshes.tri2D.Tri2D* method), 409
`__rmul__` () (*fipy.terms.advectionTerm.AdvectionTerm* method), 528
`__rmul__` () (*fipy.terms.cellTerm.CellTerm* method), 533
`__rmul__` () (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 538
`__rmul__` () (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 557
`__rmul__` () (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 562
`__rmul__` () (*fipy.terms.faceTerm.FaceTerm* method), 567
`__rmul__` () (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 572
`__rmul__` () (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 577
`__rmul__` () (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 581
`__rmul__` () (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 586
`__rmul__` () (*fipy.terms.residualTerm.ResidualTerm* method), 590
`__rmul__` () (*fipy.terms.sourceTerm.SourceTerm* method), 595
`__rmul__` () (*fipy.terms.transientTerm.TransientTerm* method), 603
`__rmul__` () (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 608
`__rmul__` () (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 613
`__rmul__` () (*fipy.tools.PhysicalField* method), 624
`__rmul__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 646
`__rmul__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 660
`__rtruediv__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 660
`__setattr__` () (*fipy.meshes.abstractMesh.MeshAdditionError* method), 153
`__setattr__` () (*fipy.meshes.gmshMesh.GmshException* method), 218
`__setattr__` () (*fipy.meshes.gmshMesh.MeshExportError* method), 236
`__setattr__` () (*fipy.meshes.mesh.MeshAdditionError* method), 245
`__setattr__` () (*fipy.solvers.SerialSolverError* method), 442
`__setattr__` () (*fipy.solvers.convergence.DivergenceWarning* method), 446
`__setattr__` () (*fipy.solvers.solver.BreakdownWarning* method), 489
`__setattr__` () (*fipy.solvers.solver.IllConditionedPreconditionerWarning* method), 495

method), 489

__setattr__() (fipy.solvers.solver.IllegalInputOrBreakdownWarning method), 490

__setattr__() (fipy.solvers.solver.LossOfPrecisionWarning method), 491

__setattr__() (fipy.solvers.solver.MatrixIllConditionedWarning method), 491

__setattr__() (fipy.solvers.solver.MaximumIterationWarning method), 492

__setattr__() (fipy.solvers.solver.NormalConvergence method), 492

__setattr__() (fipy.solvers.solver.ParameterWarning method), 493

__setattr__() (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning method), 493

__setattr__() (fipy.solvers.solver.PreconditionerWarning method), 494

__setattr__() (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning method), 495

__setattr__() (fipy.solvers.solver.SolverConvergenceWarning method), 496

__setattr__() (fipy.solvers.solver.StagnatedSolverWarning method), 496

__setattr__() (fipy.terms.AbstractBaseClassError method), 519

__setattr__() (fipy.terms.ExplicitVariableError method), 520

__setattr__() (fipy.terms.IncorrectSolutionVariable method), 520

__setattr__() (fipy.terms.SolutionVariableNumberError method), 521

__setattr__() (fipy.terms.SolutionVariableRequiredError method), 522

__setattr__() (fipy.terms.TermMultiplyError method), 522

__setattr__() (fipy.terms.TransientTermError method), 523

__setattr__() (fipy.terms.VectorCoeffError method), 523

__setattr__() (fipy.viewers.MeshDimensionError method), 904

__setitem__() (fipy.tools.PhysicalField method), 625

__setitem__() (fipy.tools.dimensions.physicalField.PhysicalField method), 646

__setstate__() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 682

__setstate__() (fipy.variables.cellVariable.CellVariable method), 697

__setstate__() (fipy.variables.distanceVariable.DistanceVariable method), 714

__setstate__() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 731

__setstate__() (fipy.variables.faceVariable.FaceVariable method), 746

__setstate__() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 758

__setstate__() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 775

__setstate__() (fipy.variables.histogramVariable.HistogramVariable method), 789

__setstate__() (fipy.variables.meshVariable.MeshVariable method), 803

__setstate__() (fipy.variables.modularVariable.ModularVariable method), 815

__setstate__() (fipy.variables.noiseVariable.NoiseVariable method), 830

__setstate__() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 843

__setstate__() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 855

__setstate__() (fipy.variables.surfactantVariable.SurfactantVariable method), 867

__setstate__() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 883

__setstate__() (fipy.variables.variable.Variable method), 897

__str__() (fipy.meshes.abstractMesh.MeshAdditionError method), 153

__str__() (fipy.meshes.gmshMesh.GmshException method), 218

__str__() (fipy.meshes.gmshMesh.MeshExportError method), 236

__str__() (fipy.meshes.mesh.MeshAdditionError method), 245

__str__() (fipy.solvers.SerialSolverError method), 442

__str__() (fipy.solvers.convergence.AbsoluteToleranceConvergence method), 444

__str__() (fipy.solvers.convergence.BreakdownDivergence method), 444

__str__() (fipy.solvers.convergence.Convergence method), 444

__str__() (fipy.solvers.convergence.ConvergenceBase method), 445

__str__() (fipy.solvers.convergence.Divergence method), 445

__str__() (fipy.solvers.convergence.DivergenceWarning method), 446

__str__() (fipy.solvers.convergence.HappyBreakdownConvergence method), 446

__str__() (fipy.solvers.convergence.IllConditionedDivergence method), 446

__str__() (fipy.solvers.convergence.IllConditionedPreconditionerDivergence method), 446

__str__() (fipy.solvers.convergence.IteratingConvergence method), 446

__str__() (fipy.solvers.convergence.IterationConvergence method), 447

__str__() (fipy.solvers.convergence.IterationDivergence method), 447

__str__() (fipy.solvers.convergence.LossOfAccuracyConvergence method), 447

__str__() (fipy.solvers.convergence.NullDivergence method), 447

__str__() (fipy.solvers.convergence.OutOfRangeDivergence method), 447

__str__() (fipy.solvers.convergence.PreconditioningDivergence method), 447

__str__() (fipy.solvers.convergence.RHSZeroConvergence method), 447

__str__() (fipy.solvers.convergence.RelativeToleranceConvergence method), 448

__str__() (fipy.solvers.convergence.StagnatedDivergence method), 448

__str__() (fipy.solvers.convergence.ToleranceDivergence method), 448

__str__() (fipy.solvers.petsc.petscConvergence.KSP_AbsoluteToleranceConvergence method), 457

__str__() (fipy.solvers.petsc.petscConvergence.KSP_BreakdownBICG Divergence method), 458

__str__() (fipy.solvers.petsc.petscConvergence.KSP_BreakdownDivergence method), 458

__str__() (fipy.solvers.petsc.petscConvergence.KSP_HappyBreakdownConvergence method), 458

__str__() (fipy.solvers.petsc.petscConvergence.KSP_IndefiniteMatrixDivergence method), 458

__str__() (fipy.solvers.petsc.petscConvergence.KSP_IndefinitePreconditionerDivergence method), 458

(fipy.solvers.petsc.petscConvergence.KSP_IteratingConvergence_str__O (fipy.solvers.solver.StagnatedSolverWarning method), 496
method), 458 __str__O

(fipy.solvers.petsc.petscConvergence.KSP_IterationConvergence (fipy.solvers.trilinos.aztecConvergence.AZ_BreakdownDivergence
method), 459 __str__O method), 498

(fipy.solvers.petsc.petscConvergence.KSP_IterationDivergence (fipy.solvers.trilinos.aztecConvergence.AZ_IllConditionedDivergence
method), 459 __str__O method), 498

(fipy.solvers.petsc.petscConvergence.KSP_NanOrInfDivergence (fipy.solvers.trilinos.aztecConvergence.AZ_IterationDivergence
method), 459 __str__O method), 498

(fipy.solvers.petsc.petscConvergence.KSP_NonSymmetricDivergence (fipy.solvers.trilinos.aztecConvergence.AZ_LossOfAccuracyConvergence
method), 459 __str__O method), 498

(fipy.solvers.petsc.petscConvergence.KSP_NormalAbsoluteToleranceConvergence (fipy.solvers.trilinos.aztecConvergence.AZ_NormalConvergence
method), 459 __str__O method), 499

(fipy.solvers.petsc.petscConvergence.KSP_NormalRelativeToleranceConvergence (fipy.solvers.trilinos.aztecConvergence.AZ_ParameterDivergence
method), 460 __str__O method), 499

(fipy.solvers.petsc.petscConvergence.KSP_NullDivergence (fipy.terms.AbstractBaseClassError method), 519
method), 460 __str__O (fipy.terms.ExplicitVariableError method), 520

(fipy.solvers.petsc.petscConvergence.KSP_PreconditioningDivergence (fipy.terms.IncorrectSolutionVariable method), 520
method), 460 __str__O (fipy.terms.SolutionVariableNumberError method), 521

(fipy.solvers.petsc.petscConvergence.KSP_RelativeToleranceConvergence (fipy.terms.SolutionVariableRequiredError method), 522
method), 460 __str__O (fipy.terms.TermMultiplyError method), 522

(fipy.solvers.petsc.petscConvergence.KSP_ToleranceDivergence (fipy.terms.TransientTermError method), 523
method), 460 __str__O (fipy.terms.VectorCoeffError method), 523

(fipy.solvers.petsc.petscConvergence.KSP_ToleranceDivergence (fipy.tools.PhysicalField method), 625
method), 460 __str__O (fipy.tools.dimensions.physicalField.PhysicalField
method), 647

(fipy.solvers.pyamgx.pyamgxConvergence.pyamgx_BreakdownDivergence (fipy.tools.dimensions.physicalField.PhysicalUnit method),
method), 478 __str__O 660

(fipy.solvers.pyamgx.pyamgxConvergence.pyamgx_Convergence_str__O (fipy.variables.betaNoiseVariable.BetaNoiseVariable
method), 478 __str__O method), 682

(fipy.solvers.pyamgx.pyamgxConvergence.pyamgx_IterationDivergence (fipy.variables.cellVariable.CellVariable method), 697
method), 478 __str__O (fipy.variables.distanceVariable.DistanceVariable method),
714

(fipy.solvers.pyamgx.pyamgxConvergence.pyamgx_IterationDivergence (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable
method), 478 __str__O method), 731

(fipy.solvers.scipy.scipyConvergence.SciPy_BreakdownDivergence (fipy.variables.faceVariable.FaceVariable method), 746
method), 485 __str__O (fipy.variables.gammaNoiseVariable.GammaNoiseVariable
method), 758

(fipy.solvers.scipy.scipyConvergence.SciPy_Convergence_str__O (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable
method), 485 __str__O method), 775

(fipy.solvers.scipy.scipyConvergence.SciPy_IterationDivergence_str__O (fipy.variables.histogramVariable.HistogramVariable
method), 486 __str__O method), 789

(fipy.solvers.solver.BreakdownWarning method), 489 __str__O (fipy.variables.meshVariable.MeshVariable method), 803

(fipy.solvers.solver.IllConditionedPreconditionerWarning (fipy.variables.modularVariable.ModularVariable method),
method), 489 __str__O 815

(fipy.solvers.solver.IllegalInputOrBreakdownWarning (fipy.variables.noiseVariable.NoiseVariable method), 830
method), 490 __str__O

(fipy.solvers.solver.LossOfPrecisionWarning method), 491 (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable
method), 843

(fipy.solvers.solver.MatrixIllConditionedWarning method), 491 __str__O

(fipy.solvers.solver.MaximumIterationWarning method), 492 (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable
method), 855

(fipy.solvers.solver.NormalConvergence method), 492 __str__O (fipy.variables.surfactantVariable.SurfactantVariable
method), 867

(fipy.solvers.solver.ParameterWarning method), 493 __str__O

(fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning (fipy.variables.uniformNoiseVariable.UniformNoiseVariable
method), 493 __str__O method), 883

(fipy.solvers.solver.PreconditionerWarning method), 494 __str__O (fipy.variables.variable.Variable method), 897

(fipy.solvers.solver.ScalarQuantityOutOfRangeWarning (fipy.viewers.MeshDimensionError method), 904
method), 495 __str__O

(fipy.solvers.solver.SolverConvergenceWarning method), 496 __sub__O (fipy.meshes.abstractMesh.AbstractMesh method), 150

(fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D

```

method), 160
__sub__ (fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D
method), 168
__sub__ (fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D
method), 176
__sub__ (fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D
method), 183
__sub__ (fipy.meshes.gmshMesh.Gmsh2D method), 198
__sub__ (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace method), 206
__sub__ (fipy.meshes.gmshMesh.Gmsh3D method), 215
__sub__ (fipy.meshes.gmshMesh.GmshGrid2D method), 223
__sub__ (fipy.meshes.gmshMesh.GmshGrid3D method), 232
__sub__ (fipy.meshes.mesh.Mesh method), 242
__sub__ (fipy.meshes.mesh1D.Mesh1D method), 251
__sub__ (fipy.meshes.mesh2D.Mesh2D method), 259
__sub__ (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D
method), 268
__sub__ (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D
method), 276
__sub__ (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D
method), 285
__sub__ (fipy.meshes.periodicGrid1D.PeriodicGrid1D method),
293
__sub__ (fipy.meshes.periodicGrid2D.PeriodicGrid2D method),
303
__sub__ (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight
method), 311
__sub__ (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom
method), 319
__sub__ (fipy.meshes.periodicGrid3D.PeriodicGrid3D method),
329
__sub__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack
method), 337
__sub__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight
method), 345
__sub__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack
method), 353
__sub__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom
method), 360
__sub__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom
method), 368
__sub__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack
method), 376
__sub__ (fipy.meshes.skewedGrid2D.SkewedGrid2D method), 384
__sub__ (fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D
method), 393
__sub__ (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D
method), 401
__sub__ (fipy.meshes.tri2D.Tri2D method), 409
__sub__ (fipy.meshes.uniformGrid.UniformGrid method), 417
__sub__ (fipy.meshes.uniformGrid1D.UniformGrid1D method), 424
__sub__ (fipy.meshes.uniformGrid2D.UniformGrid2D method), 431
__sub__ (fipy.meshes.uniformGrid3D.UniformGrid3D method), 438
__sub__ (fipy.tools.physicalField.physicalField method), 647
__truediv__ (fipy.meshes.abstractMesh.AbstractMesh method),
150
__truediv__ (fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D
method), 160
__truediv__ (fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D
method), 168
__truediv__ (fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D
method), 176
__truediv__ (fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D
method), 183
__truediv__ (fipy.meshes.gmshMesh.Gmsh2D method), 198
__truediv__ (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace method),
206
__truediv__ (fipy.meshes.gmshMesh.Gmsh3D method), 215
__truediv__ (fipy.meshes.gmshMesh.GmshGrid2D method), 223
__truediv__ (fipy.meshes.gmshMesh.GmshGrid3D method), 232
__truediv__ (fipy.meshes.mesh.Mesh method), 242
__truediv__ (fipy.meshes.mesh1D.Mesh1D method), 251
__truediv__ (fipy.meshes.mesh2D.Mesh2D method), 259
__truediv__ (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D
method), 268
__truediv__ (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D
method), 276
__truediv__ (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D
method), 285
__truediv__ (fipy.meshes.periodicGrid1D.PeriodicGrid1D
method), 293
__truediv__ (fipy.meshes.periodicGrid2D.PeriodicGrid2D
method), 303
__truediv__ (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight
method), 311
__truediv__ (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom
method), 319
__truediv__ (fipy.meshes.periodicGrid3D.PeriodicGrid3D
method), 329
__truediv__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack
method), 337
__truediv__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight
method), 345
__truediv__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack
method), 353
__truediv__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom
method), 360
__truediv__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom
method), 368
__truediv__ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack
method), 376
__truediv__ (fipy.meshes.skewedGrid2D.SkewedGrid2D method),
384
__truediv__ (fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D
method), 393
__truediv__ (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D
method), 401
__truediv__ (fipy.meshes.tri2D.Tri2D method), 409
__truediv__ (fipy.meshes.uniformGrid.UniformGrid method), 417
__truediv__ (fipy.meshes.uniformGrid1D.UniformGrid1D
method), 424

```

`__truediv__()` (*fipy.meshes.uniformGrid2D.UniformGrid2D* method), 431
`__truediv__()` (*fipy.meshes.uniformGrid3D.UniformGrid3D* method), 438
`__truediv__()` (*fipy.tools.PhysicalField* method), 625
`__truediv__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 647
`__truediv__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 661
`--cache`
 command line option, 17
`--inline`
 command line option, 17
`--lsmllib`
 command line option, 18
`--no-cache`
 command line option, 17
`--petsc`
 command line option, 18
`--pyamgx`
 command line option, 18
`--scipy`
 command line option, 18
`--skfmm`
 command line option, 18
`--trilinos`
 command line option, 18

A

`AbsoluteToleranceConvergence` (class in *fipy.solvers.convergence*), 444
`AbstractBaseClassError`, 519
`AbstractMatplotlib2DViewer` (class in *fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer*), 908
`AbstractMatplotlibViewer` (class in *fipy.viewers.matplotlibViewer.abstractMatplotlibViewer*), 910
`AbstractMesh` (class in *fipy.meshes.abstractMesh*), 146
`AbstractViewer` (class in *fipy.viewers.viewer*), 946
`actual_code` (*fipy.solvers.convergence.ConvergenceBase* attribute), 445
`add()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 647
`add()` (*fipy.tools.PhysicalField* method), 626
`add_note()` (*fipy.meshes.abstractMesh.MeshAdditionError* method), 153
`add_note()` (*fipy.meshes.gmshMesh.GmshException* method), 218
`add_note()` (*fipy.meshes.gmshMesh.MeshExportError* method), 236
`add_note()` (*fipy.meshes.mesh.MeshAdditionError* method), 245
`add_note()` (*fipy.solvers.convergence.DivergenceWarning* method), 446
`add_note()` (*fipy.solvers.SerialSolverError* method), 442
`add_note()` (*fipy.solvers.solver.BreakdownWarning* method), 489
`add_note()` (*fipy.solvers.solver.IllConditionedPreconditionerWarning* method), 489
`add_note()` (*fipy.solvers.solver.IllegalInputOrBreakdownWarning* method), 490
`add_note()` (*fipy.solvers.solver.LossOfPrecisionWarning* method), 491
`add_note()` (*fipy.solvers.solver.MatrixIllConditionedWarning* method), 491
`add_note()` (*fipy.solvers.solver.MaximumIterationWarning* method), 492
`add_note()` (*fipy.solvers.solver.NormalConvergence* method), 492
`add_note()` (*fipy.solvers.solver.ParameterWarning* method), 493
`add_note()`
 (*fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning* method), 494

`add_note()` (*fipy.solvers.solver.PreconditionerWarning* method), 494
`add_note()` (*fipy.solvers.solver.ScalarQuantityOutOfRangeWarning* method), 495
`add_note()` (*fipy.solvers.solver.SolverConvergenceWarning* method), 496
`add_note()` (*fipy.solvers.solver.StagnatedSolverWarning* method), 497
`add_note()` (*fipy.terms.AbstractBaseClassError* method), 519
`add_note()` (*fipy.terms.ExplicitVariableError* method), 520
`add_note()` (*fipy.terms.IncorrectSolutionVariable* method), 520
`add_note()` (*fipy.terms.SolutionVariableNumberError* method), 521
`add_note()` (*fipy.terms.SolutionVariableRequiredError* method), 522
`add_note()` (*fipy.terms.TermMultiplyError* method), 522
`add_note()` (*fipy.terms.TransientTermError* method), 523
`add_note()` (*fipy.terms.VectorCoeffError* method), 523
`add_note()` (*fipy.viewers.MeshDimensionError* method), 904
`AdvectionTerm`, 1112
`AdvectionTerm` (class in *fipy.terms.advectionTerm*), 525
`AggregationAMGPreconditioner` (class in *fipy.solvers.pyamgx.preconditioners*), 475
`AggregationAMGSolver` (class in *fipy.solvers.pyamgx.aggregationAMGSolver*), 468
`all()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 682
`all()` (*fipy.variables.cellVariable.CellVariable* method), 697
`all()` (*fipy.variables.distanceVariable.DistanceVariable* method), 714
`all()`
 (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 732
`all()` (*fipy.variables.faceVariable.FaceVariable* method), 746
`all()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 758
`all()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 775
`all()` (*fipy.variables.histogramVariable.HistogramVariable* method), 789
`all()` (*fipy.variables.meshVariable.MeshVariable* method), 803
`all()` (*fipy.variables.modularVariable.ModularVariable* method), 815
`all()` (*fipy.variables.noiseVariable.NoiseVariable* method), 830
`all()`
 (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 843
`all()`
 (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 855
`all()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 867
`all()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 883
`all()` (*fipy.variables.variable.Variable* method), 897
`all()` (in module *fipy.tools.numerix*), 668
`allclose`, 1141, 1199
`allclose()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 647
`allclose()` (*fipy.tools.PhysicalField* method), 626
`allclose()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 683
`allclose()` (*fipy.variables.cellVariable.CellVariable* method), 697
`allclose()` (*fipy.variables.distanceVariable.DistanceVariable* method), 715
`allclose()`
 (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 732
`allclose()` (*fipy.variables.faceVariable.FaceVariable* method), 746
`allclose()`
 (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 758
`allclose()`
 (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 775

- method*), 775
- `allclose()` (*fipy.variables.histogramVariable.HistogramVariable method*), 789
- `allclose()` (*fipy.variables.meshVariable.MeshVariable method*), 803
- `allclose()` (*fipy.variables.modularVariable.ModularVariable method*), 815
- `allclose()` (*fipy.variables.noiseVariable.NoiseVariable method*), 830
- `allclose()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 844
- `allclose()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 855
- `allclose()` (*fipy.variables.surfactantVariable.SurfactantVariable method*), 867
- `allclose()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 883
- `allclose()` (*fipy.variables.variable.Variable method*), 897
- `allclose()` (in module *fipy.tools.numerix*), 668
- `allequal()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 648
- `allequal()` (*fipy.tools.PhysicalField method*), 626
- `allequal()` (in module *fipy.tools.numerix*), 668
- `allgather()` (*fipy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper method*), 500
- AMG**, 121
- AMGPreconditioner** (class in *fipy.solvers.pyamgx.preconditioners*), 475
- `any()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable method*), 683
- `any()` (*fipy.variables.cellVariable.CellVariable method*), 698
- `any()` (*fipy.variables.distanceVariable.DistanceVariable method*), 715
- `any()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 732
- `any()` (*fipy.variables.faceVariable.FaceVariable method*), 746
- `any()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 759
- `any()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 775
- `any()` (*fipy.variables.histogramVariable.HistogramVariable method*), 789
- `any()` (*fipy.variables.meshVariable.MeshVariable method*), 804
- `any()` (*fipy.variables.modularVariable.ModularVariable method*), 815
- `any()` (*fipy.variables.noiseVariable.NoiseVariable method*), 830
- `any()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 844
- `any()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 856
- `any()` (*fipy.variables.surfactantVariable.SurfactantVariable method*), 867
- `any()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 883
- `any()` (*fipy.variables.variable.Variable method*), 897
- AppVeyor**, 121
- `arccos()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 648
- `arccos()` (*fipy.tools.PhysicalField method*), 626
- `arccosh()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 648
- `arccosh()` (*fipy.tools.PhysicalField method*), 626
- `arcsin()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 648
- `arcsin()` (*fipy.tools.PhysicalField method*), 626
- `arctan`, 1135, 1140
- `arctan()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 648
- `arctan()` (*fipy.tools.PhysicalField method*), 627
- `arctan2`, 1135, 1140
- `arctan2()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 649
- `arctan2()` (*fipy.tools.PhysicalField method*), 627
- `arctanh()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 649
- `arctanh()` (*fipy.tools.PhysicalField method*), 627
- `arithmeticFaceValue` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable property*), 683
- `arithmeticFaceValue` (*fipy.variables.cellVariable.CellVariable property*), 698
- `arithmeticFaceValue` (*fipy.variables.distanceVariable.DistanceVariable property*), 715
- `arithmeticFaceValue` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable property*), 732
- `arithmeticFaceValue` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable property*), 759
- `arithmeticFaceValue` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable property*), 775
- `arithmeticFaceValue` (*fipy.variables.histogramVariable.HistogramVariable property*), 789
- `arithmeticFaceValue` (*fipy.variables.modularVariable.ModularVariable property*), 815
- `arithmeticFaceValue` (*fipy.variables.noiseVariable.NoiseVariable property*), 830
- `arithmeticFaceValue` (*fipy.variables.surfactantVariable.SurfactantVariable property*), 868
- `arithmeticFaceValue` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable property*), 883
- array**, 1146, 1160
- `aspect2D` (*fipy.meshes.abstractMesh.AbstractMesh property*), 150
- `aspect2D` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D property*), 160
- `aspect2D` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D property*), 168
- `aspect2D` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property*), 176
- `aspect2D` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property*), 183
- `aspect2D` (*fipy.meshes.gmshMesh.Gmsh2D property*), 198
- `aspect2D` (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property*), 206
- `aspect2D` (*fipy.meshes.gmshMesh.Gmsh3D property*), 215
- `aspect2D` (*fipy.meshes.gmshMesh.GmshGrid2D property*), 224
- `aspect2D` (*fipy.meshes.gmshMesh.GmshGrid3D property*), 232
- `aspect2D` (*fipy.meshes.mesh.Mesh property*), 242
- `aspect2D` (*fipy.meshes.mesh1D.Mesh1D property*), 251
- `aspect2D` (*fipy.meshes.mesh2D.Mesh2D property*), 259
- `aspect2D` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D property*), 268
- `aspect2D` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D property*), 276

- aspect2D (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 285
- aspect2D (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 293
- aspect2D (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 303
- aspect2D (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 311
- aspect2D (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 320
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 329
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 337
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 345
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 353
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 361
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 368
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 376
- aspect2D (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 384
- aspect2D (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 394
- aspect2D (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 401
- aspect2D (*fipy.meshes.tri2D.Tri2D* property), 410
- aspect2D (*fipy.meshes.uniformGrid.UniformGrid* property), 417
- aspect2D (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 424
- aspect2D (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 431
- aspect2D (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 438
- axes (examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer property), 1121
- axes (examples.phase.phaseViewer.PhaseViewer property), 1180
- axes (*fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* property), 909
- axes (*fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* property), 911
- axes (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 914
- axes (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* property), 916
- axes (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* property), 919
- axes (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* property), 922
- axes (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* property), 925
- axes (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 929
- axes (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* property), 933
- AZ_BreakdownDivergence (class in *fipy.solvers.trilinos.aztecConvergence*), 498
- AZ_IllConditionedDivergence (class in *fipy.solvers.trilinos.aztecConvergence*), 498
- AZ_IterationDivergence (class in *fipy.solvers.trilinos.aztecConvergence*), 498
- AZ_LossOfAccuracyConvergence (class in *fipy.solvers.trilinos.aztecConvergence*), 498
- AZ_NormalConvergence (class in *fipy.solvers.trilinos.aztecConvergence*), 498
- AZ_ParameterDivergence (class in *fipy.solvers.trilinos.aztecConvergence*), 499
- Azure, 121
- B**
- Base (class in package.subpackage.base), 136
- BetaNoiseVariable (class in *fipy.variables.betaNoiseVariable*), 676
- BiCG, 121
- BiCGSTAB, 121
- BiCGStabPreconditioner (class in *fipy.solvers.pyamgx.preconditioners*), 475
- BoundaryCondition (class in *fipy.boundaryConditions.boundaryCondition*), 141
- BreakdownDivergence (class in *fipy.solvers.convergence*), 444
- BreakdownWarning, 488
- C**
- cacheMatrix, 1094
- cacheMatrix() (*fipy.terms.advectionTerm.AdvectionTerm* method), 529
- cacheMatrix() (*fipy.terms.cellTerm.CellTerm* method), 533
- cacheMatrix() (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 538
- cacheMatrix() (*fipy.terms.diffusionTerm.DiffusionTerm* method), 542
- cacheMatrix() (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* method), 545
- cacheMatrix() (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* method), 549
- cacheMatrix() (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* method), 552
- cacheMatrix() (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 557
- cacheMatrix() (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 562
- cacheMatrix() (*fipy.terms.faceTerm.FaceTerm* method), 567
- cacheMatrix() (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 572
- cacheMatrix() (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 577
- cacheMatrix() (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 581
- cacheMatrix() (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 587
- cacheMatrix() (*fipy.terms.residualTerm.ResidualTerm* method), 591
- cacheMatrix() (*fipy.terms.sourceTerm.SourceTerm* method), 595
- cacheMatrix() (*fipy.terms.term.Term* method), 598
- cacheMatrix() (*fipy.terms.transientTerm.TransientTerm* method), 603

cacheMatrix()
(fipy.terms.upwindConvectionTerm.UpwindConvectionTerm method), 608

cacheMatrix()
(fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm method), 613

cacheRHSvector, 1094

cacheRHSvector() *(fipy.terms.advectionTerm.AdvectionTerm method)*, 529

cacheRHSvector() *(fipy.terms.cellTerm.CellTerm method)*, 533

cacheRHSvector()
(fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm method), 538

cacheRHSvector() *(fipy.terms.diffusionTerm.DiffusionTerm method)*, 542

cacheRHSvector()
(fipy.terms.diffusionTermCorrection.DiffusionTermCorrection method), 545

cacheRHSvector()
(fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection method), 549

cacheRHSvector()
(fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm method), 552

cacheRHSvector()
(fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm method), 557

cacheRHSvector()
(fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm method), 562

cacheRHSvector() *(fipy.terms.faceTerm.FaceTerm method)*, 567

cacheRHSvector()
(fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm method), 572

cacheRHSvector()
(fipy.terms.hybridConvectionTerm.HybridConvectionTerm method), 577

cacheRHSvector()
(fipy.terms.implicitSourceTerm.ImplicitSourceTerm method), 581

cacheRHSvector()
(fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm method), 587

cacheRHSvector() *(fipy.terms.residualTerm.ResidualTerm method)*, 591

cacheRHSvector() *(fipy.terms.sourceTerm.SourceTerm method)*, 595

cacheRHSvector() *(fipy.terms.term.Term method)*, 598

cacheRHSvector() *(fipy.terms.transientTerm.TransientTerm method)*, 603

cacheRHSvector()
(fipy.terms.upwindConvectionTerm.UpwindConvectionTerm method), 608

cacheRHSvector()
(fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm method), 613

calcDistanceFunction()
(fipy.variables.distanceVariable.DistanceVariable method), 716

ceil() *(fipy.tools.dimensions.physicalField.PhysicalField method)*, 649

ceil() *(fipy.tools.physicalField.PhysicalField method)*, 628

cellCenters *(fipy.meshes.abstractMesh.AbstractMesh property)*, 150

cellCenters
(fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D property), 160

cellCenters
(fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D property), 168

cellCenters
(fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property), 176

cellCenters
(fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property), 183

cellCenters *(fipy.meshes.gmshMesh.Gmsh2D property)*, 198

cellCenters *(fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property)*, 207

cellCenters *(fipy.meshes.gmshMesh.Gmsh3D property)*, 215

cellCenters *(fipy.meshes.gmshMesh.GmshGrid2D property)*, 224

cellCenters *(fipy.meshes.gmshMesh.GmshGrid3D property)*, 232

cellCenters *(fipy.meshes.mesh.Mesh property)*, 242

cellCenters *(fipy.meshes.mesh1D.Mesh1D property)*, 251

cellCenters *(fipy.meshes.mesh2D.Mesh2D property)*, 259

cellCenters *(fipy.meshes.nonUniformGrid1D.NonUniformGrid1D property)*, 268

cellCenters *(fipy.meshes.nonUniformGrid2D.NonUniformGrid2D property)*, 276

cellCenters *(fipy.meshes.nonUniformGrid3D.NonUniformGrid3D property)*, 285

cellCenters *(fipy.meshes.periodicGrid1D.PeriodicGrid1D property)*, 294

cellCenters *(fipy.meshes.periodicGrid2D.PeriodicGrid2D property)*, 303

cellCenters *(fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight property)*, 311

cellCenters *(fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom property)*, 320

cellCenters *(fipy.meshes.periodicGrid3D.PeriodicGrid3D property)*, 330

cellCenters *(fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property)*, 337

cellCenters *(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property)*, 345

cellCenters
(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property), 353

cellCenters
(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property), 361

cellCenters *(fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property)*, 368

cellCenters
(fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property), 376

cellCenters *(fipy.meshes.skewedGrid2D.SkewedGrid2D property)*, 385

cellCenters
(fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property), 394

cellCenters
(fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property), 401

cellCenters *(fipy.meshes.tri2D.Tri2D property)*, 410

cellCenters *(fipy.meshes.uniformGrid.UniformGrid property)*, 417

cellCenters *(fipy.meshes.uniformGrid1D.UniformGrid1D property)*, 424

cellCenters *(fipy.meshes.uniformGrid2D.UniformGrid2D property)*, 431

cellCenters *(fipy.meshes.uniformGrid3D.UniformGrid3D property)*, 438

cellFaceIDs *(fipy.meshes.abstractMesh.AbstractMesh property)*, 150

cellFaceIDs
(fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D property), 160

cellFaceIDs
(fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D property), 168

- property), 168
- cellFaceIDs (fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property), 176
- cellFaceIDs (fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property), 183
- cellFaceIDs (fipy.meshes.gmshMesh.Gmsh2D property), 198
- cellFaceIDs (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property), 207
- cellFaceIDs (fipy.meshes.gmshMesh.Gmsh3D property), 215
- cellFaceIDs (fipy.meshes.gmshMesh.GmshGrid2D property), 224
- cellFaceIDs (fipy.meshes.gmshMesh.GmshGrid3D property), 232
- cellFaceIDs (fipy.meshes.mesh.Mesh property), 242
- cellFaceIDs (fipy.meshes.mesh1D.Mesh1D property), 251
- cellFaceIDs (fipy.meshes.mesh2D.Mesh2D property), 259
- cellFaceIDs (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D property), 268
- cellFaceIDs (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D property), 276
- cellFaceIDs (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D property), 285
- cellFaceIDs (fipy.meshes.periodicGrid1D.PeriodicGrid1D property), 294
- cellFaceIDs (fipy.meshes.periodicGrid2D.PeriodicGrid2D property), 303
- cellFaceIDs (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight property), 311
- cellFaceIDs (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom property), 320
- cellFaceIDs (fipy.meshes.periodicGrid3D.PeriodicGrid3D property), 330
- cellFaceIDs (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property), 337
- cellFaceIDs (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property), 345
- cellFaceIDs (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property), 353
- cellFaceIDs (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property), 361
- cellFaceIDs (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property), 368
- cellFaceIDs (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property), 376
- cellFaceIDs (fipy.meshes.skewedGrid2D.SkewedGrid2D property), 385
- cellFaceIDs (fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property), 394
- cellFaceIDs (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property), 401
- cellFaceIDs (fipy.meshes.tri2D.Tri2D property), 410
- cellFaceIDs (fipy.meshes.uniformGrid.UniformGrid property), 417
- cellFaceIDs (fipy.meshes.uniformGrid1D.UniformGrid1D property), 424
- cellFaceIDs (fipy.meshes.uniformGrid2D.UniformGrid2D property), 431
- cellFaceIDs (fipy.meshes.uniformGrid3D.UniformGrid3D property), 438
- cellInterfaceAreas (fipy.variables.distanceVariable.DistanceVariable property), 716
- CellTerm (class in fipy.terms.cellTerm), 532
- CellVariable, 971, 987, 1050, 1092, 1111, 1139, 1142, 1155, 1170, 1175, 1194, 1200, 1221, 1223
- CellVariable (class in fipy.variables.cellVariable), 692
- cellVolumeAverage (fipy.variables.betaNoiseVariable.BetaNoiseVariable property), 684
- cellVolumeAverage (fipy.variables.cellVariable.CellVariable property), 698
- cellVolumeAverage (fipy.variables.distanceVariable.DistanceVariable property), 717
- cellVolumeAverage (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable property), 733
- cellVolumeAverage (fipy.variables.gammaNoiseVariable.GammaNoiseVariable property), 759
- cellVolumeAverage (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable property), 776
- cellVolumeAverage (fipy.variables.histogramVariable.HistogramVariable property), 790
- cellVolumeAverage (fipy.variables.modularVariable.ModularVariable property), 816
- cellVolumeAverage (fipy.variables.noiseVariable.NoiseVariable property), 831
- cellVolumeAverage (fipy.variables.surfactantVariable.SurfactantVariable property), 868
- cellVolumeAverage (fipy.variables.uniformNoiseVariable.UniformNoiseVariable property), 884
- CentralDifferenceConvectionTerm (class in fipy.terms.centralDiffConvectionTerm), 536
- CG, 121
- CGPreconditioner (class in fipy.solvers.pyamgx.preconditioners), 475
- CGS, 121
- CircleCI, 121
- ClassicalAMGSolver (class in fipy.solvers.pyamgx.classicalAMGSolver), 469
- clock_ns() (fipy.tools.timer.Timer static method), 673
- cmap (examples.phase.phaseViewer.PhaseViewer property), 1180
- cmap (fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer property), 909
- cmap (fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer property), 911
- cmap (fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer property), 914
- cmap (fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer property), 916
- cmap (fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer property), 919
- cmap (fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer property), 922
- cmap (fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer property), 922

property), 925

cmap (fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer property), 929

cmap (fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer property), 933

collection (fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer property), 925

colorbar (examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer property), 1121

colorbar (examples.phase.phaseViewer.PhaseViewer property), 1180

colorbar (fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer property), 909

colorbar (fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer property), 911

colorbar (fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer property), 914

colorbar (fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer property), 916

colorbar (fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer property), 919

colorbar (fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer property), 922

colorbar (fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer property), 925

colorbar (fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer property), 929

colorbar (fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer property), 933

command line option

- cache, 17
- inline, 17
- lsmlib, 18
- no-cache, 17
- petsc, 18
- pyamgx, 18
- scipy, 18
- skfmm, 18
- trilinos, 18

CommWrapper (class in fipy.tools.comms.commWrapper), 637

conda, 121

conda_info() (in module fipy.tools.logging.environment), 665

conjugate() (fipy.tools.dimensions.physicalField.PhysicalField method), 649

conjugate() (fipy.tools.PhysicalField method), 628

constrain() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 684

constrain() (fipy.variables.cellVariable.CellVariable method), 699

constrain() (fipy.variables.distanceVariable.DistanceVariable method), 717

constrain() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 733

constrain() (fipy.variables.faceVariable.FaceVariable method), 746

constrain() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 760

constrain() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 776

constrain() (fipy.variables.histogramVariable.HistogramVariable method), 790

constrain() (fipy.variables.meshVariable.MeshVariable method), 804

constrain() (fipy.variables.modularVariable.ModularVariable method), 816

constrain() (fipy.variables.noiseVariable.NoiseVariable method), 832

constrain() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 845

constrain() (fipy.viewers.matplotlib2DViewer.Matplotlib2DViewer property), 909

constrain() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 856

constrain() (fipy.variables.surfactantVariable.SurfactantVariable method), 868

constrain() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 884

constrain() (fipy.variables.variable.Variable method), 897

constraintMask (fipy.variables.betaNoiseVariable.BetaNoiseVariable property), 685

constraintMask (fipy.viewers.matplotlib2DGridViewer.Matplotlib2DGridViewer property), 718

constraintMask (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable property), 734

constraintMask (fipy.variables.faceVariable.FaceVariable property), 747

constraintMask (fipy.viewers.matplotlibStreamViewer.MatplotlibStreamViewer property), 760

constraintMask (fipy.variables.gammaNoiseVariable.GammaNoiseVariable property), 760

constraintMask (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable property), 777

constraintMask (fipy.variables.histogramVariable.HistogramVariable property), 791

constraintMask (fipy.variables.meshVariable.MeshVariable property), 805

constraintMask (fipy.variables.modularVariable.ModularVariable property), 817

constraintMask (fipy.variables.noiseVariable.NoiseVariable property), 832

constraintMask (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable property), 845

constraintMask (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable property), 857

constraintMask (fipy.variables.surfactantVariable.SurfactantVariable property), 869

constraintMask (fipy.variables.uniformNoiseVariable.UniformNoiseVariable property), 885

Continuous Integration, 121

Convergence (class in fipy.solvers.convergence), 444

ConvergenceBase (class in fipy.solvers.convergence), 444

conversionFactorTo() (fipy.tools.dimensions.physicalField.PhysicalUnit method),

- 661
conversionTupleTo() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 662
convertToUnit() (fipy.tools.dimensions.physicalField.PhysicalField method), 649
convertToUnit() (fipy.tools.PhysicalField method), 628
copy() (fipy.tools.dimensions.physicalField.PhysicalField method), 650
copy() (fipy.tools.PhysicalField method), 628
copy() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 685
copy() (fipy.variables.cellVariable.CellVariable method), 700
copy() (fipy.variables.distanceVariable.DistanceVariable method), 718
copy() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 734
copy() (fipy.variables.faceVariable.FaceVariable method), 748
copy() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 761
copy() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 777
copy() (fipy.variables.histogramVariable.HistogramVariable method), 792
copy() (fipy.variables.meshVariable.MeshVariable method), 805
copy() (fipy.variables.modularVariable.ModularVariable method), 817
copy() (fipy.variables.noiseVariable.NoiseVariable method), 832
copy() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 845
copy() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 857
copy() (fipy.variables.surfactantVariable.SurfactantVariable method), 870
copy() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 885
copy() (fipy.variables.variable.Variable method), 898
cos() (fipy.tools.dimensions.physicalField.PhysicalField method), 650
cos() (fipy.tools.PhysicalField method), 628
cosh() (fipy.tools.dimensions.physicalField.PhysicalField method), 650
cosh() (fipy.tools.PhysicalField method), 629
criterion (fipy.solvers.convergence.ConvergenceBase attribute), 444
CylindricalGrid1D() (in module fipy.meshes.factoryMeshes), 186
CylindricalGrid2D() (in module fipy.meshes.factoryMeshes), 186
CylindricalNonUniformGrid1D (class in fipy.meshes.cylindricalNonUniformGrid1D), 154
CylindricalNonUniformGrid2D (class in fipy.meshes.cylindricalNonUniformGrid2D), 163
CylindricalUniformGrid1D (class in fipy.meshes.cylindricalUniformGrid1D), 172
CylindricalUniformGrid2D (class in fipy.meshes.cylindricalUniformGrid2D), 179
- ## D
- DEFAULT_ITERATIONS (fipy.solvers.petsc.dummySolver.DummySolver attribute), 451
DEFAULT_ITERATIONS (fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver attribute), 452
DEFAULT_ITERATIONS (fipy.solvers.petsc.linearCGSolver.LinearCGSolver attribute), 454
DEFAULT_ITERATIONS (fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver attribute), 453
DEFAULT_ITERATIONS (fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver attribute), 455
DEFAULT_ITERATIONS (fipy.solvers.petsc.linearLUSolver.LinearLUSolver attribute), 456
DEFAULT_ITERATIONS (fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver attribute), 461
DEFAULT_ITERATIONS (fipy.solvers.petsc.petscSolver.PETScSolver attribute), 463
DEFAULT_ITERATIONS (fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver attribute), 469
DEFAULT_ITERATIONS (fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver attribute), 470
DEFAULT_ITERATIONS (fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver attribute), 470
DEFAULT_ITERATIONS (fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver attribute), 471
DEFAULT_ITERATIONS (fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver attribute), 472
DEFAULT_ITERATIONS (fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver attribute), 473
DEFAULT_ITERATIONS (fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver attribute), 474
DEFAULT_ITERATIONS (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver attribute), 477
DEFAULT_ITERATIONS (fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver attribute), 480
DEFAULT_ITERATIONS (fipy.solvers.scipy.linearCGSolver.LinearCGSolver attribute), 481
DEFAULT_ITERATIONS (fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver attribute), 480
DEFAULT_ITERATIONS (fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver attribute), 482
DEFAULT_ITERATIONS (fipy.solvers.scipy.linearLUSolver.LinearLUSolver attribute), 483
DEFAULT_ITERATIONS (fipy.solvers.scipy.scipyKrylovSolver.ScipyKrylovSolver attribute), 486
DEFAULT_ITERATIONS (fipy.solvers.scipy.scipySolver.ScipySolver attribute), 487
DEFAULT_ITERATIONS (fipy.solvers.solver.Solver attribute), 495
DEFAULT_ITERATIONS (fipy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver attribute), 501
DEFAULT_ITERATIONS (fipy.solvers.trilinos.linearCGSolver.LinearCGSolver attribute), 503
DEFAULT_ITERATIONS (fipy.solvers.trilinos.linearCGSSolver.LinearCGSSolver attribute), 502
DEFAULT_ITERATIONS (fipy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver attribute), 502

- attribute), 504
- DEFAULT_ITERATIONS (fipy.solvers.trilinos.linearLUSolver.LinearLUSolver attribute), 505
- DEFAULT_ITERATIONS (fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver attribute), 512
- DEFAULT_ITERATIONS (fipy.solvers.trilinos.trilinosMLTest.TrilinosMLTest attribute), 513
- DEFAULT_ITERATIONS (fipy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver attribute), 514
- DEFAULT_ITERATIONS (fipy.solvers.trilinos.trilinosSolver.TrilinosSolver attribute), 515
- DEFAULT_PRECONDITIONER (fipy.solvers.petsc.dummySolver.DummySolver attribute), 451
- DEFAULT_PRECONDITIONER (fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver attribute), 452
- DEFAULT_PRECONDITIONER (fipy.solvers.petsc.linearCGSolver.LinearCGSolver attribute), 454
- DEFAULT_PRECONDITIONER (fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver attribute), 453
- DEFAULT_PRECONDITIONER (fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver attribute), 455
- DEFAULT_PRECONDITIONER (fipy.solvers.petsc.linearLUSolver.LinearLUSolver attribute), 456
- DEFAULT_PRECONDITIONER (fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver attribute), 462
- DEFAULT_PRECONDITIONER (fipy.solvers.petsc.petscSolver.PETScSolver attribute), 463
- DEFAULT_PRECONDITIONER (fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver attribute), 469
- DEFAULT_PRECONDITIONER (fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver attribute), 470
- DEFAULT_PRECONDITIONER (fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver attribute), 471
- DEFAULT_PRECONDITIONER (fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver attribute), 471
- DEFAULT_PRECONDITIONER (fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver attribute), 472
- DEFAULT_PRECONDITIONER (fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver attribute), 473
- DEFAULT_PRECONDITIONER (fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver attribute), 474
- DEFAULT_PRECONDITIONER (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver attribute), 477
- DEFAULT_PRECONDITIONER (fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver attribute), 480
- DEFAULT_PRECONDITIONER (fipy.solvers.scipy.linearCGSolver.LinearCGSolver attribute), 481
- DEFAULT_PRECONDITIONER (fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver attribute), 480
- DEFAULT_PRECONDITIONER (fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver attribute), 482
- DEFAULT_PRECONDITIONER (fipy.solvers.scipy.linearLUSolver.LinearLUSolver attribute), 483
- DEFAULT_PRECONDITIONER (fipy.solvers.scipy.scipyKrylovSolver.ScipyKrylovSolver attribute), 486
- DEFAULT_PRECONDITIONER (fipy.solvers.scipy.scipySolver.ScipySolver attribute), 487
- DEFAULT_PRECONDITIONER (fipy.solvers.solver.Solver attribute), 495
- DEFAULT_PRECONDITIONER (fipy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver attribute), 501
- DEFAULT_PRECONDITIONER (fipy.solvers.trilinos.linearCGSolver.LinearCGSolver attribute), 503
- DEFAULT_PRECONDITIONER (fipy.solvers.trilinos.linearCGSSolver.LinearCGSSolver attribute), 502
- DEFAULT_PRECONDITIONER (fipy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver attribute), 504
- DEFAULT_PRECONDITIONER (fipy.solvers.trilinos.linearLUSolver.LinearLUSolver attribute), 505
- DEFAULT_PRECONDITIONER (fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver attribute), 512
- DEFAULT_PRECONDITIONER (fipy.solvers.trilinos.trilinosMLTest.TrilinosMLTest attribute), 513
- DEFAULT_PRECONDITIONER (fipy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver attribute), 514
- DEFAULT_PRECONDITIONER (fipy.solvers.trilinos.trilinosSolver.TrilinosSolver attribute), 515
- DEFAULT_SMOOTHER (fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver attribute), 469
- DEFAULT_SMOOTHER (fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver attribute), 470
- DEFAULT_SMOOTHER (fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver attribute), 471
- DEFAULT_SMOOTHER (fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver attribute), 472
- DEFAULT_SMOOTHER (fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver attribute), 473
- DEFAULT_SMOOTHER (fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver attribute), 473
- DEFAULT_SMOOTHER (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver attribute), 477
- DEFAULT_TOLERANCE (fipy.solvers.petsc.dummySolver.DummySolver attribute), 451
- default_tolerance (fipy.solvers.petsc.dummySolver.DummySolver property), 451

DEFAULT_TOLERANCE (fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver attribute), 452	(fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver property), 473
default_tolerance (fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver property), 452	DEFAULT_TOLERANCE (fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver attribute), 474
DEFAULT_TOLERANCE (fipy.solvers.petsc.linearCGSolver.LinearCGSolver attribute), 454	default_tolerance (fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver property), 474
default_tolerance (fipy.solvers.petsc.linearCGSolver.LinearCGSolver property), 454	DEFAULT_TOLERANCE (fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver attribute), 474
DEFAULT_TOLERANCE (fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver attribute), 453	default_tolerance (fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver property), 474
default_tolerance (fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver property), 453	DEFAULT_TOLERANCE (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver attribute), 477
DEFAULT_TOLERANCE (fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver attribute), 455	default_tolerance (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver property), 477
default_tolerance (fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver property), 455	DEFAULT_TOLERANCE (fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver attribute), 480
DEFAULT_TOLERANCE (fipy.solvers.petsc.linearLUSolver.LinearLUSolver attribute), 456	default_tolerance (fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver property), 480
default_tolerance (fipy.solvers.petsc.linearLUSolver.LinearLUSolver property), 456	DEFAULT_TOLERANCE (fipy.solvers.scipy.linearCGSolver.LinearCGSolver attribute), 481
DEFAULT_TOLERANCE (fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver attribute), 462	default_tolerance (fipy.solvers.scipy.linearCGSolver.LinearCGSolver property), 482
default_tolerance (fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver property), 462	DEFAULT_TOLERANCE (fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver attribute), 480
DEFAULT_TOLERANCE (fipy.solvers.petsc.petscSolver.PETScSolver attribute), 463	default_tolerance (fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver property), 481
default_tolerance (fipy.solvers.petsc.petscSolver.PETScSolver property), 463	DEFAULT_TOLERANCE (fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver attribute), 482
DEFAULT_TOLERANCE (fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver attribute), 469	default_tolerance (fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver property), 482
default_tolerance (fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver property), 469	DEFAULT_TOLERANCE (fipy.solvers.scipy.linearLUSolver.LinearLUSolver attribute), 483
DEFAULT_TOLERANCE (fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver attribute), 470	default_tolerance (fipy.solvers.scipy.linearLUSolver.LinearLUSolver property), 483
default_tolerance (fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver property), 470	DEFAULT_TOLERANCE (fipy.solvers.scipy.scipyKrylovSolver.ScipyKrylovSolver attribute), 486
DEFAULT_TOLERANCE (fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver attribute), 471	default_tolerance (fipy.solvers.scipy.scipyKrylovSolver.ScipyKrylovSolver property), 487
default_tolerance (fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver property), 471	DEFAULT_TOLERANCE (fipy.solvers.scipy.scipySolver.ScipySolver attribute), 487
DEFAULT_TOLERANCE (fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver attribute), 472	default_tolerance (fipy.solvers.scipy.scipySolver.ScipySolver property), 487
default_tolerance (fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver property), 472	DEFAULT_TOLERANCE (fipy.solvers.solver.Solver attribute), 495
DEFAULT_TOLERANCE (fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver attribute), 473	default_tolerance (fipy.solvers.solver.Solver property), 495
default_tolerance	DEFAULT_TOLERANCE (fipy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver attribute), 501
	default_tolerance (fipy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver property), 501

DEFAULT_TOLERANCE (*fiPy.solvers.trilinos.linearCGSolver.LinearCGSolver* attribute), 503
default_tolerance (*fiPy.solvers.trilinos.linearCGSolver.LinearCGSolver* property), 503
DEFAULT_TOLERANCE (*fiPy.solvers.trilinos.linearCGSSolver.LinearCGSSolver* attribute), 502
default_tolerance (*fiPy.solvers.trilinos.linearCGSSolver.LinearCGSSolver* property), 502
DEFAULT_TOLERANCE (*fiPy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver* attribute), 504
default_tolerance (*fiPy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver* property), 504
DEFAULT_TOLERANCE (*fiPy.solvers.trilinos.linearLUSolver.LinearLUSolver* attribute), 505
default_tolerance (*fiPy.solvers.trilinos.linearLUSolver.LinearLUSolver* property), 505
DEFAULT_TOLERANCE (*fiPy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver* attribute), 512
default_tolerance (*fiPy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver* property), 512
DEFAULT_TOLERANCE (*fiPy.solvers.trilinos.trilinosMLTest.TrilinosMLTest* attribute), 513
default_tolerance (*fiPy.solvers.trilinos.trilinosMLTest.TrilinosMLTest* property), 513
DEFAULT_TOLERANCE (*fiPy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver* attribute), 514
default_tolerance (*fiPy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver* property), 514
DEFAULT_TOLERANCE (*fiPy.solvers.trilinos.trilinosSolver.TrilinosSolver* attribute), 515
default_tolerance (*fiPy.solvers.trilinos.trilinosSolver.TrilinosSolver* property), 516
DefaultAsymmetricSolver, 987, 1162, 1198
DefaultAsymmetricSolver (in module *fiPy.solvers*), 441
DefaultPreconditioner (class in *fiPy.solvers.petsc.preconditioners*), 463
DefaultPreconditioner (class in *fiPy.solvers.petsc.preconditioners.defaultPreconditioner*), 464
DefaultSolver, 971
DefaultSolver (in module *fiPy.solvers*), 441
deprecate() (in module *fiPy.tools.decorators*), 638
DiffusionTerm (class in *fiPy.terms.diffusionTerm*), 541
DiffusionTermCorrection (class in *fiPy.terms.diffusionTermCorrection*), 545
DiffusionTermNoCorrection (class in *fiPy.terms.diffusionTermNoCorrection*), 548
DILUPreconditioner (class in *fiPy.solvers.pyamgx.preconditioners*), 475
DistanceVariable, 1110
DistanceVariable (class in *fiPy.variables.distanceVariable*), 707
Divergence (class in *fiPy.solvers.convergence*), 445
divergence (*fiPy.variables.faceVariable.FaceVariable* property), 748
divergence (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 846
divergence (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 858
DivergenceWarning, 445
divide() (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 651
divide() (*fiPy.tools.PhysicalField* method), 629
doctest_raw_input() (in module *fiPy*), 139
DomDecompPreconditioner (class in *fiPy.solvers.trilinos.preconditioners*), 505
DomDecompPreconditioner (class in *fiPy.solvers.trilinos.preconditioners.domDecompPreconditioner*), 507
dot() (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 651
dot() (*fiPy.tools.PhysicalField* method), 629
dot() (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* method), 685
dot() (*fiPy.variables.cellVariable.CellVariable* method), 700
dot() (*fiPy.variables.distanceVariable.DistanceVariable* method), 719
dot() (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 734
dot() (*fiPy.variables.faceVariable.FaceVariable* method), 748
dot() (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 761
dot() (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 778
dot() (*fiPy.variables.histogramVariable.HistogramVariable* method), 792
dot() (*fiPy.variables.meshVariable.MeshVariable* method), 806
dot() (*fiPy.variables.modularVariable.ModularVariable* method), 818
dot() (*fiPy.variables.noiseVariable.NoiseVariable* method), 833
dot() (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 846
dot() (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 858
dot() (*fiPy.variables.surfactantVariable.SurfactantVariable* method), 870
dot() (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 886
dot() (in module *fiPy.tools.numerix*), 668
dtype (*fiPy.tools.dimensions.physicalField.PhysicalField* property), 651
dtype (*fiPy.tools.PhysicalField* property), 629
dtype (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* property), 685
dtype (*fiPy.variables.cellVariable.CellVariable* property), 701
dtype (*fiPy.variables.distanceVariable.DistanceVariable* property), 719
dtype (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 735
dtype (*fiPy.variables.faceVariable.FaceVariable* property), 748
dtype (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 761
dtype (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 778
dtype (*fiPy.variables.histogramVariable.HistogramVariable* property), 792
dtype (*fiPy.variables.meshVariable.MeshVariable* property), 806
dtype (*fiPy.variables.modularVariable.ModularVariable* property), 818
dtype (*fiPy.variables.noiseVariable.NoiseVariable* property), 833
dtype (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 846

property), 846

dtype
 (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 858

dtype (*fipy.variables.surfactantVariable.SurfactantVariable* property), 870

dtype (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 886

dtype (*fipy.variables.variable.Variable* property), 899

DummyComm (class in *fipy.tools.comms.dummyComm*), 637

DummySolver (class in *fipy.solvers.petsc.dummySolver*), 451

DummySolver (in module *fipy.solvers*), 442

DummyViewer (class in *fipy.viewers*), 902

E

elapsed (*fipy.tools.timer.Timer* property), 673

environment variable
 FIPY_CACHE, 19
 FIPY_DEFAULT_CRITERION, 18
 FIPY_DISPLAY_MATRIX, 18
 FIPY_INCLUDE_NUMERIX_ALL, 19, 1230
 FIPY_INLINE, 18
 FIPY_INLINE_COMMENT, 18
 FIPY_LOG_CONFIG, 16, 18
 FIPY_SOLVERS, 18, 31, 32, 82
 FIPY_VIEWER, 18
 PETSC_OPTIONS, 19

EpetraCommWrapper (class in *fipy.solvers.trilinos.comms.epetraCommWrapper*), 499

error() (in module *fipy.steps*), 516

examples
 module, 959

examples.benchmarking
 module, 960

examples.benchmarking.benchmarker
 module, 960

examples.benchmarking.size
 module, 960

examples.benchmarking.steps
 module, 960

examples.benchmarking.utils
 module, 960

examples.benchmarking.versions
 module, 960

examples.cahnHilliard
 module, 960

examples.cahnHilliard.mesh2D
 module, 961

examples.cahnHilliard.mesh2DCoupled
 module, 963

examples.cahnHilliard.mesh3D
 module, 966

examples.cahnHilliard.sphere
 module, 968

examples.cahnHilliard.sphereDaemon
 module, 970

examples.cahnHilliard.tanh1D
 module, 970

examples.cahnHilliard.test
 module, 972

examples.chemotaxis
 module, 972

examples.chemotaxis.input
 module, 973

examples.chemotaxis.input2D
 module, 975

examples.chemotaxis.parameters

module, 977

examples.chemotaxis.test
 module, 977

examples.convection
 module, 977

examples.convection.advection
 module, 978

examples.convection.advection.explicitUpwind
 module, 978

examples.convection.advection.implicitUpwind
 module, 978

examples.convection.advection.vanLeerUpwind
 module, 979

examples.convection.exponential1D
 module, 979

examples.convection.exponential1D.cylindricalMesh1D
 module, 980

examples.convection.exponential1D.cylindricalMesh1DNonUniform
 module, 981

examples.convection.exponential1D.mesh1D
 module, 983

examples.convection.exponential1D.tri2D
 module, 984

examples.convection.exponential1DBack
 module, 985

examples.convection.exponential1DBack.mesh1D
 module, 985

examples.convection.exponential1DSource
 module, 986

examples.convection.exponential1DSource.mesh1D
 module, 987

examples.convection.exponential1DSource.tri2D
 module, 988

examples.convection.exponential2D
 module, 989

examples.convection.exponential2D.cylindricalMesh2D
 module, 989

examples.convection.exponential2D.cylindricalMesh2DNonUniform
 module, 991

examples.convection.exponential2D.mesh2D
 module, 992

examples.convection.exponential2D.tri2D
 module, 993

examples.convection.peclet
 module, 994

examples.convection.powerLaw1D
 module, 996

examples.convection.powerLaw1D.mesh1D
 module, 996

examples.convection.powerLaw1D.tri2D
 module, 997

examples.convection.robin
 module, 998

examples.convection.source
 module, 1000

examples.convection.test
 module, 1001

examples.diffusion
 module, 1001

examples.diffusion.anisotropy
 module, 1001

examples.diffusion.circle
 module, 1003

examples.diffusion.circleQuad
 module, 1008

examples.diffusion.coupled
 module, 1013

examples.diffusion.electrostatics

module, 1015
 examples.diffusion.explicit
 module, 1019
 examples.diffusion.explicit.mesh1D
 module, 1020
 examples.diffusion.explicit.mixedelement
 module, 1021
 examples.diffusion.explicit.test
 module, 1022
 examples.diffusion.explicit.tri2D
 module, 1022
 examples.diffusion.mesh1D
 module, 1023
 examples.diffusion.mesh20x20
 module, 1043
 examples.diffusion.mesh20x20Coupled
 module, 1046
 examples.diffusion.nthOrder
 module, 1049
 examples.diffusion.nthOrder.input4thOrder1D
 module, 1050
 examples.diffusion.nthOrder.input4thOrder_line
 module, 1052
 examples.diffusion.nthOrder.test
 module, 1052
 examples.diffusion.steadyState
 module, 1052
 examples.diffusion.steadyState.mesh1D
 module, 1052
 examples.diffusion.steadyState.mesh1D.inputPeriodic
 module, 1053
 examples.diffusion.steadyState.mesh1D.tri2Dinput
 module, 1054
 examples.diffusion.steadyState.mesh20x20
 module, 1054
 examples.diffusion.steadyState.mesh20x20.gmshinput
 module, 1055
 examples.diffusion.steadyState.mesh20x20.isotropy
 module, 1055
 examples.diffusion.steadyState.mesh20x20.modifiedMeshInput
 module, 1055
 examples.diffusion.steadyState.mesh20x20.orthoerror
 module, 1057
 examples.diffusion.steadyState.mesh20x20.tri2Dinput
 module, 1057
 examples.diffusion.steadyState.mesh50x50
 module, 1057
 examples.diffusion.steadyState.mesh50x50.input
 module, 1057
 examples.diffusion.steadyState.mesh50x50.tri2Dinput
 module, 1058
 examples.diffusion.steadyState.otherMeshes
 module, 1058
 examples.diffusion.steadyState.otherMeshes.cubicalProblem
 module, 1058
 examples.diffusion.steadyState.otherMeshes.grid3Dinput
 module, 1059
 examples.diffusion.steadyState.otherMeshes.prism
 module, 1059
 examples.diffusion.steadyState.test
 module, 1060
 examples.diffusion.test
 module, 1060
 examples.diffusion.variable
 module, 1060
 examples.elphf
 module, 1060
 examples.elphf.diffusion
 module, 1062
 examples.elphf.diffusion.mesh1D
 module, 1062
 examples.elphf.diffusion.mesh1Ddimensional
 module, 1064
 examples.elphf.diffusion.mesh2D
 module, 1067
 examples.elphf.input
 module, 1070
 examples.elphf.phase
 module, 1076
 examples.elphf.phaseDiffusion
 module, 1079
 examples.elphf.poisson
 module, 1087
 examples.elphf.test
 module, 1090
 examples.flow
 module, 1090
 examples.flow.stokesCavity
 module, 1091
 examples.flow.test
 module, 1096
 examples.levelSet
 module, 1096
 examples.levelSet.advection
 module, 1096
 examples.levelSet.advection.circle
 module, 1096
 examples.levelSet.advection.mesh1D
 module, 1098
 examples.levelSet.advection.test
 module, 1099
 examples.levelSet.advection.trench
 module, 1099
 examples.levelSet.distanceFunction
 module, 1101
 examples.levelSet.distanceFunction.circle
 module, 1101
 examples.levelSet.distanceFunction.interior
 module, 1103
 examples.levelSet.distanceFunction.mesh1D
 module, 1103
 examples.levelSet.distanceFunction.square
 module, 1104
 examples.levelSet.distanceFunction.test
 module, 1105
 examples.levelSet.electroChem
 module, 1105
 examples.levelSet.electroChem.adsorbingSurfactantEquation
 module, 1106
 examples.levelSet.electroChem.adsorption
 module, 1106
 examples.levelSet.electroChem.gapFillDistanceVariable
 module, 1107
 examples.levelSet.electroChem.gapFillMesh
 module, 1107
 examples.levelSet.electroChem.gold
 module, 1107
 examples.levelSet.electroChem.howToWriteAScript
 module, 1108
 examples.levelSet.electroChem.leveler
 module, 1114
 examples.levelSet.electroChem.lines
 module, 1118
 examples.levelSet.electroChem.matplotlibSurfactantViewer
 module, 1118
 examples.levelSet.electroChem.mayaviSurfactantViewer

[module](#), 1122
[examples.levelSet.electroChem.metalIonDiffusionEquation](#)
[module](#), 1125
[examples.levelSet.electroChem.simpleTrenchSystem](#)
[module](#), 1125
[examples.levelSet.electroChem.surfactantBulkDiffusionEquation](#)
[module](#), 1128
[examples.levelSet.electroChem.test](#)
[module](#), 1128
[examples.levelSet.electroChem.trenchMesh](#)
[module](#), 1128
[examples.levelSet.surfactant](#)
[module](#), 1128
[examples.levelSet.surfactant.circle](#)
[module](#), 1129
[examples.levelSet.surfactant.expandingCircle](#)
[module](#), 1129
[examples.levelSet.surfactant.square](#)
[module](#), 1130
[examples.levelSet.surfactant.test](#)
[module](#), 1131
[examples.levelSet.test](#)
[module](#), 1131
[examples.meshing](#)
[module](#), 1131
[examples.meshing.gmshRefinement](#)
[module](#), 1131
[examples.meshing.inputGrid2D](#)
[module](#), 1131
[examples.meshing.sphere](#)
[module](#), 1132
[examples.meshing.test](#)
[module](#), 1133
[examples.parallel](#)
[module](#), 1133
[examples.phase](#)
[module](#), 1133
[examples.phase.anisotropy](#)
[module](#), 1134
[examples.phase.anisotropyOLD](#)
[module](#), 1138
[examples.phase.binary](#)
[module](#), 1141
[examples.phase.binaryCoupled](#)
[module](#), 1155
[examples.phase.impingement](#)
[module](#), 1169
[examples.phase.impingement.mesh20x20](#)
[module](#), 1170
[examples.phase.impingement.mesh40x1](#)
[module](#), 1174
[examples.phase.impingement.test](#)
[module](#), 1177
[examples.phase.missOrientation](#)
[module](#), 1177
[examples.phase.missOrientation.circle](#)
[module](#), 1177
[examples.phase.missOrientation.mesh1D](#)
[module](#), 1178
[examples.phase.missOrientation.modCircle](#)
[module](#), 1178
[examples.phase.missOrientation.test](#)
[module](#), 1179
[examples.phase.phaseViewer](#)
[module](#), 1179
[examples.phase.polyxtal](#)
[module](#), 1182
[examples.phase.polyxtalCoupled](#)
[module](#), 1188
[examples.phase.quaternary](#)
[module](#), 1194
[examples.phase.simple](#)
[module](#), 1200
[examples.phase.symmetry](#)
[module](#), 1208
[examples.phase.test](#)
[module](#), 1210
[examples.reactiveWetting](#)
[module](#), 1210
[examples.reactiveWetting.liquidVapor1D](#)
[module](#), 1210
[examples.reactiveWetting.liquidVapor2D](#)
[module](#), 1216
[examples.reactiveWetting.test](#)
[module](#), 1220
[examples.riemann](#)
[module](#), 1220
[examples.riemann.acoustics](#)
[module](#), 1220
[examples.riemann.test](#)
[module](#), 1220
[examples.test](#)
[module](#), 1220
[examples.updating](#)
[module](#), 1220
[examples.updating.update0_1to1_0](#)
[module](#), 1221
[examples.updating.update1_0to2_0](#)
[module](#), 1225
[examples.updating.update2_0to3_0](#)
[module](#), 1229
[execButNoTest\(\)](#) (in module *fipy.tests.doctestPlus*), 617
[exp](#), 972, 980, 982, 984, 987, 990, 992, 1109, 1146, 1160, 1171, 1175
[ExplicitDiffusionTerm](#), 1024, 1140, 1171, 1175
[ExplicitDiffusionTerm](#) (class in *fipy.terms.explicitDiffusionTerm*), 552
[ExplicitUpwindConvectionTerm](#) (class in *fipy.terms.explicitUpwindConvectionTerm*), 555
[ExplicitVariableError](#), 519
[ExponentialConvectionTerm](#), 1222
[ExponentialConvectionTerm](#) (class in *fipy.terms.exponentialConvectionTerm*), 560
[ExponentialNoiseVariable](#) (class in *fipy.variables.exponentialNoiseVariable*), 725
[extendVariable\(\)](#) (*fipy.variables.distanceVariable.DistanceVariable* method), 719
[exteriorFaces](#)
(fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D
property), 176
[exteriorFaces](#)
(fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D
property), 401
[exteriorFaces](#) (*fipy.meshes.uniformGrid1D.UniformGrid1D*
property), 424
[extrude\(\)](#)
(fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D
method), 168
[extrude\(\)](#) (*fipy.meshes.gmshMesh.Gmsh2D* method), 198
[extrude\(\)](#) (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* method), 207
[extrude\(\)](#) (*fipy.meshes.gmshMesh.GmshGrid2D* method), 224
[extrude\(\)](#) (*fipy.meshes.mesh2D.Mesh2D* method), 259
[extrude\(\)](#) (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D*
method), 276
[extrude\(\)](#) (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 303

`extrude()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 311

`extrude()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 320

`extrude()` (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 385

`extrude()` (*fipy.meshes.tri2D.Tri2D* method), 410

F

`faceGrad` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 686

`faceGrad` (*fipy.variables.cellVariable.CellVariable* property), 701

`faceGrad` (*fipy.variables.distanceVariable.DistanceVariable* property), 719

`faceGrad` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 735

`faceGrad` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 761

`faceGrad` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 778

`faceGrad` (*fipy.variables.histogramVariable.HistogramVariable* property), 792

`faceGrad` (*fipy.variables.modularVariable.ModularVariable* property), 818

`faceGrad` (*fipy.variables.noiseVariable.NoiseVariable* property), 833

`faceGrad` (*fipy.variables.surfactantVariable.SurfactantVariable* property), 871

`faceGrad` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 886

`faceGradNoMod` (*fipy.variables.modularVariable.ModularVariable* property), 818

`facesBack` (*fipy.meshes.abstractMesh.AbstractMesh* property), 150

`facesBack` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 160

`facesBack` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 169

`facesBack` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 176

`facesBack` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 183

`facesBack` (*fipy.meshes.gmshMesh.Gmsh2D* property), 199

`facesBack` (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 207

`facesBack` (*fipy.meshes.gmshMesh.Gmsh3D* property), 215

`facesBack` (*fipy.meshes.gmshMesh.GmshGrid2D* property), 224

`facesBack` (*fipy.meshes.gmshMesh.GmshGrid3D* property), 232

`facesBack` (*fipy.meshes.mesh.Mesh* property), 242

`facesBack` (*fipy.meshes.mesh1D.Mesh1D* property), 251

`facesBack` (*fipy.meshes.mesh2D.Mesh2D* property), 260

`facesBack` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 268

`facesBack` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 277

`facesBack` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 285

`facesBack` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 294

`facesBack` (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 303

`facesBack` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 312

`facesBack` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 320

`facesBack` (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 330

`facesBack` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 337

`facesBack` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 345

`facesBack` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 353

`facesBack` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 361

`facesBack` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 368

`facesBack` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 376

`facesBack` (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 385

`facesBack` (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 394

`facesBack` (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 401

`facesBack` (*fipy.meshes.tri2D.Tri2D* property), 410

`facesBack` (*fipy.meshes.uniformGrid.UniformGrid* property), 417

`facesBack` (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 424

`facesBack` (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 431

`facesBack` (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 438

`facesBottom` (*fipy.meshes.abstractMesh.AbstractMesh* property), 150

`facesBottom` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 160

`facesBottom` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 169

`facesBottom` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 176

`facesBottom` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 183

`facesBottom` (*fipy.meshes.gmshMesh.Gmsh2D* property), 199

`facesBottom` (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 207

`facesBottom` (*fipy.meshes.gmshMesh.Gmsh3D* property), 215

`facesBottom` (*fipy.meshes.gmshMesh.GmshGrid2D* property), 224

`facesBottom` (*fipy.meshes.gmshMesh.GmshGrid3D* property), 232

`facesBottom` (*fipy.meshes.mesh.Mesh* property), 242

`facesBottom` (*fipy.meshes.mesh1D.Mesh1D* property), 251

`facesBottom` (*fipy.meshes.mesh2D.Mesh2D* property), 260

`facesBottom` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 268

`facesBottom` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 277

`facesBottom` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 285

`facesBottom` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 294

`facesBottom` (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 304

`facesBottom` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 312

`facesBottom` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 320

<code>facesBottom</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3D</code> property), 330	<code>facesDown</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3D</code> property), 330
<code>facesBottom</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack</code> property), 338	<code>facesDown</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack</code> property), 338
<code>facesBottom</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight</code> property), 345	<code>facesDown</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight</code> property), 346
<code>facesBottom</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack</code> property), 353	<code>facesDown</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack</code> property), 353
<code>facesBottom</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom</code> property), 361	<code>facesDown</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom</code> property), 361
<code>facesBottom</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom</code> property), 369	<code>facesDown</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom</code> property), 369
<code>facesBottom</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack</code> property), 376	<code>facesDown</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack</code> property), 377
<code>facesBottom</code> (<code>fipy.meshes.skewedGrid2D.SkewedGrid2D</code> property), 385	<code>facesDown</code> (<code>fipy.meshes.skewedGrid2D.SkewedGrid2D</code> property), 386
<code>facesBottom</code> (<code>fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D</code> property), 394	<code>facesDown</code> (<code>fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D</code> property), 394
<code>facesBottom</code> (<code>fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D</code> property), 401	<code>facesDown</code> (<code>fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D</code> property), 401
<code>facesBottom</code> (<code>fipy.meshes.tri2D.Tri2D</code> property), 410	<code>facesDown</code> (<code>fipy.meshes.tri2D.Tri2D</code> property), 411
<code>facesBottom</code> (<code>fipy.meshes.uniformGrid.UniformGrid</code> property), 417	<code>facesDown</code> (<code>fipy.meshes.uniformGrid.UniformGrid</code> property), 418
<code>facesBottom</code> (<code>fipy.meshes.uniformGrid1D.UniformGrid1D</code> property), 424	<code>facesDown</code> (<code>fipy.meshes.uniformGrid1D.UniformGrid1D</code> property), 425
<code>facesBottom</code> (<code>fipy.meshes.uniformGrid2D.UniformGrid2D</code> property), 431	<code>facesDown</code> (<code>fipy.meshes.uniformGrid2D.UniformGrid2D</code> property), 432
<code>facesBottom</code> (<code>fipy.meshes.uniformGrid3D.UniformGrid3D</code> property), 439	<code>facesDown</code> (<code>fipy.meshes.uniformGrid3D.UniformGrid3D</code> property), 439
<code>facesDown</code> (<code>fipy.meshes.abstractMesh.AbstractMesh</code> property), 150	<code>facesFront</code> (<code>fipy.meshes.abstractMesh.AbstractMesh</code> property), 151
<code>facesDown</code> (<code>fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D</code> property), 160	<code>facesFront</code> (<code>fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D</code> property), 161
<code>facesDown</code> (<code>fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D</code> property), 169	<code>facesFront</code> (<code>fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D</code> property), 169
<code>facesDown</code> (<code>fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D</code> property), 176	<code>facesFront</code> (<code>fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D</code> property), 177
<code>facesDown</code> (<code>fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D</code> property), 183	<code>facesFront</code> (<code>fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D</code> property), 184
<code>facesDown</code> (<code>fipy.meshes.gmshMesh.Gmsh2D</code> property), 199	<code>facesFront</code> (<code>fipy.meshes.gmshMesh.Gmsh2D</code> property), 199
<code>facesDown</code> (<code>fipy.meshes.gmshMesh.Gmsh2DIn3DSpace</code> property), 208	<code>facesFront</code> (<code>fipy.meshes.gmshMesh.Gmsh2DIn3DSpace</code> property), 208
<code>facesDown</code> (<code>fipy.meshes.gmshMesh.Gmsh3D</code> property), 216	<code>facesFront</code> (<code>fipy.meshes.gmshMesh.Gmsh3D</code> property), 216
<code>facesDown</code> (<code>fipy.meshes.gmshMesh.GmshGrid2D</code> property), 225	<code>facesFront</code> (<code>fipy.meshes.gmshMesh.GmshGrid2D</code> property), 225
<code>facesDown</code> (<code>fipy.meshes.gmshMesh.GmshGrid3D</code> property), 232	<code>facesFront</code> (<code>fipy.meshes.gmshMesh.GmshGrid3D</code> property), 233
<code>facesDown</code> (<code>fipy.meshes.mesh.Mesh</code> property), 243	<code>facesFront</code> (<code>fipy.meshes.mesh.Mesh</code> property), 243
<code>facesDown</code> (<code>fipy.meshes.mesh1D.Mesh1D</code> property), 251	<code>facesFront</code> (<code>fipy.meshes.mesh1D.Mesh1D</code> property), 252
<code>facesDown</code> (<code>fipy.meshes.mesh2D.Mesh2D</code> property), 260	<code>facesFront</code> (<code>fipy.meshes.mesh2D.Mesh2D</code> property), 260
<code>facesDown</code> (<code>fipy.meshes.nonUniformGrid1D.NonUniformGrid1D</code> property), 269	<code>facesFront</code> (<code>fipy.meshes.nonUniformGrid1D.NonUniformGrid1D</code> property), 269
<code>facesDown</code> (<code>fipy.meshes.nonUniformGrid2D.NonUniformGrid2D</code> property), 277	<code>facesFront</code> (<code>fipy.meshes.nonUniformGrid2D.NonUniformGrid2D</code> property), 278
<code>facesDown</code> (<code>fipy.meshes.nonUniformGrid3D.NonUniformGrid3D</code> property), 285	<code>facesFront</code> (<code>fipy.meshes.nonUniformGrid3D.NonUniformGrid3D</code> property), 286
<code>facesDown</code> (<code>fipy.meshes.periodicGrid1D.PeriodicGrid1D</code> property), 294	<code>facesFront</code> (<code>fipy.meshes.periodicGrid1D.PeriodicGrid1D</code> property), 294
<code>facesDown</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2D</code> property), 304	<code>facesFront</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2D</code> property), 304
<code>facesDown</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight</code> property), 312	<code>facesFront</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight</code> property), 313
<code>facesDown</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom</code> property), 321	<code>facesFront</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom</code> property), 321

- `facesFront` (`fipy.meshes.periodicGrid3D.PeriodicGrid3D` property), 330
- `facesFront` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack` property), 338
- `facesFront` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight` property), 346
- `facesFront` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack` property), 354
- `facesFront` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom` property), 361
- `facesFront` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom` property), 369
- `facesFront` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack` property), 377
- `facesFront` (`fipy.meshes.skewedGrid2D.SkewedGrid2D` property), 386
- `facesFront` (`fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D` property), 394
- `facesFront` (`fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D` property), 402
- `facesFront` (`fipy.meshes.tri2D.Tri2D` property), 411
- `facesFront` (`fipy.meshes.uniformGrid.UniformGrid` property), 418
- `facesFront` (`fipy.meshes.uniformGrid1D.UniformGrid1D` property), 425
- `facesFront` (`fipy.meshes.uniformGrid2D.UniformGrid2D` property), 432
- `facesFront` (`fipy.meshes.uniformGrid3D.UniformGrid3D` property), 439
- `facesLeft` (`fipy.meshes.abstractMesh.AbstractMesh` property), 151
- `facesLeft` (`fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D` property), 161
- `facesLeft` (`fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D` property), 170
- `facesLeft` (`fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D` property), 177
- `facesLeft` (`fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D` property), 184
- `facesLeft` (`fipy.meshes.gmshMesh.Gmsh2D` property), 199
- `facesLeft` (`fipy.meshes.gmshMesh.Gmsh2DIn3DSpace` property), 208
- `facesLeft` (`fipy.meshes.gmshMesh.Gmsh3D` property), 216
- `facesLeft` (`fipy.meshes.gmshMesh.GmshGrid2D` property), 225
- `facesLeft` (`fipy.meshes.gmshMesh.GmshGrid3D` property), 233
- `facesLeft` (`fipy.meshes.mesh.Mesh` property), 243
- `facesLeft` (`fipy.meshes.mesh1D.Mesh1D` property), 252
- `facesLeft` (`fipy.meshes.mesh2D.Mesh2D` property), 261
- `facesLeft` (`fipy.meshes.nonUniformGrid1D.NonUniformGrid1D` property), 269
- `facesLeft` (`fipy.meshes.nonUniformGrid2D.NonUniformGrid2D` property), 278
- `facesLeft` (`fipy.meshes.nonUniformGrid3D.NonUniformGrid3D` property), 286
- `facesLeft` (`fipy.meshes.periodicGrid1D.PeriodicGrid1D` property), 295
- `facesLeft` (`fipy.meshes.periodicGrid2D.PeriodicGrid2D` property), 304
- `facesLeft` (`fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight` property), 313
- `facesLeft` (`fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom` property), 321
- `facesLeft` (`fipy.meshes.periodicGrid3D.PeriodicGrid3D` property), 331
- `facesLeft` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack` property), 338
- `facesLeft` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight` property), 346
- `facesLeft` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack` property), 354
- `facesLeft` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom` property), 362
- `facesLeft` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom` property), 369
- `facesLeft` (`fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack` property), 377
- `facesLeft` (`fipy.meshes.skewedGrid2D.SkewedGrid2D` property), 386
- `facesLeft` (`fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D` property), 395
- `facesLeft` (`fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D` property), 402
- `facesLeft` (`fipy.meshes.tri2D.Tri2D` property), 411
- `facesLeft` (`fipy.meshes.uniformGrid.UniformGrid` property), 418
- `facesLeft` (`fipy.meshes.uniformGrid1D.UniformGrid1D` property), 425
- `facesLeft` (`fipy.meshes.uniformGrid2D.UniformGrid2D` property), 432
- `facesLeft` (`fipy.meshes.uniformGrid3D.UniformGrid3D` property), 439
- `facesRight` (`fipy.meshes.abstractMesh.AbstractMesh` property), 151
- `facesRight` (`fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D` property), 161
- `facesRight` (`fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D` property), 170
- `facesRight` (`fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D` property), 177
- `facesRight` (`fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D` property), 184
- `facesRight` (`fipy.meshes.gmshMesh.Gmsh2D` property), 200
- `facesRight` (`fipy.meshes.gmshMesh.Gmsh2DIn3DSpace` property), 208
- `facesRight` (`fipy.meshes.gmshMesh.Gmsh3D` property), 216
- `facesRight` (`fipy.meshes.gmshMesh.GmshGrid2D` property), 225
- `facesRight` (`fipy.meshes.gmshMesh.GmshGrid3D` property), 233
- `facesRight` (`fipy.meshes.mesh.Mesh` property), 243
- `facesRight` (`fipy.meshes.mesh1D.Mesh1D` property), 252
- `facesRight` (`fipy.meshes.mesh2D.Mesh2D` property), 261
- `facesRight` (`fipy.meshes.nonUniformGrid1D.NonUniformGrid1D` property), 269
- `facesRight` (`fipy.meshes.nonUniformGrid2D.NonUniformGrid2D` property), 278
- `facesRight` (`fipy.meshes.nonUniformGrid3D.NonUniformGrid3D` property), 286
- `facesRight` (`fipy.meshes.periodicGrid1D.PeriodicGrid1D` property), 295
- `facesRight` (`fipy.meshes.periodicGrid2D.PeriodicGrid2D` property), 305
- `facesRight` (`fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight` property), 313
- `facesRight` (`fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom` property), 321

<code>facesRight</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3D</code> property), 331	<code>facesTop</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack</code> property), 339
<code>facesRight</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight</code> property), 346	<code>facesTop</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack</code> property), 354
<code>facesRight</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom</code> property), 362	<code>facesTop</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom</code> property), 370
<code>facesRight</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom</code> property), 377	<code>facesTop</code> (<code>fipy.meshes.skewedGrid2D.SkewedGrid2D</code> property), 387
<code>facesRight</code> (<code>fipy.meshes.skewedGrid2D.SkewedGrid2D</code> property), 386	<code>facesTop</code> (<code>fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D</code> property), 395
<code>facesRight</code> (<code>fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D</code> property), 402	<code>facesTop</code> (<code>fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D</code> property), 402
<code>facesRight</code> (<code>fipy.meshes.tri2D.Tri2D</code> property), 411	<code>facesTop</code> (<code>fipy.meshes.tri2D.Tri2D</code> property), 412
<code>facesRight</code> (<code>fipy.meshes.uniformGrid.UniformGrid</code> property), 418	<code>facesTop</code> (<code>fipy.meshes.uniformGrid.UniformGrid</code> property), 419
<code>facesRight</code> (<code>fipy.meshes.uniformGrid1D.UniformGrid1D</code> property), 425	<code>facesTop</code> (<code>fipy.meshes.uniformGrid1D.UniformGrid1D</code> property), 426
<code>facesRight</code> (<code>fipy.meshes.uniformGrid2D.UniformGrid2D</code> property), 432	<code>facesTop</code> (<code>fipy.meshes.uniformGrid2D.UniformGrid2D</code> property), 433
<code>facesRight</code> (<code>fipy.meshes.uniformGrid3D.UniformGrid3D</code> property), 440	<code>facesTop</code> (<code>fipy.meshes.uniformGrid3D.UniformGrid3D</code> property), 440
<code>facesTop</code> (<code>fipy.meshes.abstractMesh.AbstractMesh</code> property), 151	<code>facesUp</code> (<code>fipy.meshes.abstractMesh.AbstractMesh</code> property), 152
<code>facesTop</code> (<code>fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D</code> property), 161	<code>facesUp</code> (<code>fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D</code> property), 162
<code>facesTop</code> (<code>fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D</code> property), 170	<code>facesUp</code> (<code>fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D</code> property), 170
<code>facesTop</code> (<code>fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D</code> property), 177	<code>facesUp</code> (<code>fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D</code> property), 178
<code>facesTop</code> (<code>fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D</code> property), 184	<code>facesUp</code> (<code>fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D</code> property), 185
<code>facesTop</code> (<code>fipy.meshes.gmshMesh.Gmsh2D</code> property), 200	<code>facesUp</code> (<code>fipy.meshes.gmshMesh.Gmsh2D</code> property), 200
<code>facesTop</code> (<code>fipy.meshes.gmshMesh.Gmsh2DIn3DSpace</code> property), 209	<code>facesUp</code> (<code>fipy.meshes.gmshMesh.Gmsh2DIn3DSpace</code> property), 209
<code>facesTop</code> (<code>fipy.meshes.gmshMesh.Gmsh3D</code> property), 217	<code>facesUp</code> (<code>fipy.meshes.gmshMesh.Gmsh3D</code> property), 217
<code>facesTop</code> (<code>fipy.meshes.gmshMesh.GmshGrid2D</code> property), 226	<code>facesUp</code> (<code>fipy.meshes.gmshMesh.GmshGrid2D</code> property), 226
<code>facesTop</code> (<code>fipy.meshes.gmshMesh.GmshGrid3D</code> property), 233	<code>facesUp</code> (<code>fipy.meshes.gmshMesh.GmshGrid3D</code> property), 234
<code>facesTop</code> (<code>fipy.meshes.mesh.Mesh</code> property), 244	<code>facesUp</code> (<code>fipy.meshes.mesh.Mesh</code> property), 244
<code>facesTop</code> (<code>fipy.meshes.mesh1D.Mesh1D</code> property), 252	<code>facesUp</code> (<code>fipy.meshes.mesh1D.Mesh1D</code> property), 253
<code>facesTop</code> (<code>fipy.meshes.mesh2D.Mesh2D</code> property), 261	<code>facesUp</code> (<code>fipy.meshes.mesh2D.Mesh2D</code> property), 261
<code>facesTop</code> (<code>fipy.meshes.nonUniformGrid1D.NonUniformGrid1D</code> property), 270	<code>facesUp</code> (<code>fipy.meshes.nonUniformGrid1D.NonUniformGrid1D</code> property), 270
<code>facesTop</code> (<code>fipy.meshes.nonUniformGrid2D.NonUniformGrid2D</code> property), 278	<code>facesUp</code> (<code>fipy.meshes.nonUniformGrid2D.NonUniformGrid2D</code> property), 279
<code>facesTop</code> (<code>fipy.meshes.nonUniformGrid3D.NonUniformGrid3D</code> property), 286	<code>facesUp</code> (<code>fipy.meshes.nonUniformGrid3D.NonUniformGrid3D</code> property), 287
<code>facesTop</code> (<code>fipy.meshes.periodicGrid1D.PeriodicGrid1D</code> property), 295	<code>facesUp</code> (<code>fipy.meshes.periodicGrid1D.PeriodicGrid1D</code> property), 295
<code>facesTop</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2D</code> property), 305	<code>facesUp</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2D</code> property), 305
<code>facesTop</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight</code> property), 313	<code>facesUp</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight</code> property), 314
<code>facesTop</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom</code> property), 322	<code>facesUp</code> (<code>fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom</code> property), 322
<code>facesTop</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3D</code> property), 331	<code>facesUp</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3D</code> property), 331
	<code>facesUp</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack</code> property), 339
	<code>facesUp</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight</code> property), 347
	<code>facesUp</code> (<code>fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack</code> property), 355

facesUp	(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property), 362	fig	(fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer property), 922
facesUp	(fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property), 370	fig	(fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer property), 925
facesUp	(fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property), 378	fig	(fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer property), 929
facesUp	(fipy.meshes.skewedGrid2D.SkewedGrid2D property), 387	fig	(fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer property), 933
facesUp	(fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property), 395	FiPy, 122	
facesUp	(fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property), 403	fipy	module, 139
facesUp	(fipy.meshes.tri2D.Tri2D property), 412	fipy.boundaryConditions	module, 140
facesUp	(fipy.meshes.uniformGrid.UniformGrid property), 419	fipy.boundaryConditions.boundaryCondition	module, 141
facesUp	(fipy.meshes.uniformGrid1D.UniformGrid1D property), 426	fipy.boundaryConditions.constraint	module, 141
facesUp	(fipy.meshes.uniformGrid2D.UniformGrid2D property), 433	fipy.boundaryConditions.fixedFlux	module, 142
facesUp	(fipy.meshes.uniformGrid3D.UniformGrid3D property), 440	fipy.boundaryConditions.fixedValue	module, 142
FaceTerm	(class in fipy.terms.faceTerm), 566	fipy.boundaryConditions.nthOrderBoundaryCondition	module, 143
faceValue	(fipy.variables.betaNoiseVariable.BetaNoiseVariable property), 686	fipy.boundaryConditions.test	module, 143
faceValue	(fipy.variables.cellVariable.CellVariable property), 701	fipy.matrices	module, 144
faceValue	(fipy.variables.distanceVariable.DistanceVariable property), 719	fipy.matrices.offsetSparseMatrix	module, 144
faceValue	(fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable property), 735	fipy.matrices.petscMatrix	module, 144
faceValue	(fipy.variables.gammaNoiseVariable.GammaNoiseVariable property), 761	fipy.matrices.scipyMatrix	module, 144
faceValue	(fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable property), 778	fipy.matrices.sparseMatrix	module, 144
faceValue	(fipy.variables.histogramVariable.HistogramVariable property), 792	fipy.matrices.test	module, 144
faceValue	(fipy.variables.modularVariable.ModularVariable property), 818	fipy.matrices.trilinosMatrix	module, 144
faceValue	(fipy.variables.noiseVariable.NoiseVariable property), 833	fipy.meshes	module, 144
faceValue	(fipy.variables.surfactantVariable.SurfactantVariable property), 871	fipy.meshes.abstractMesh	module, 146
faceValue	(fipy.variables.uniformNoiseVariable.UniformNoiseVariable property), 886	fipy.meshes.builders	module, 153
FaceVariable, 1030		fipy.meshes.builders.abstractGridBuilder	module, 154
FaceVariable	(class in fipy.variables.faceVariable), 741	fipy.meshes.builders.grid1DBuilder	module, 154
FGMRES, 122		fipy.meshes.builders.grid2DBuilder	module, 154
FGMRESPreconditioner	(class in fipy.solvers.pyamgx.preconditioners), 476	fipy.meshes.builders.grid3DBuilder	module, 154
fig (exam-	ples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer property), 1121	fipy.meshes.builders.periodicGrid1DBuilder	module, 154
fig (examples.phaseViewer.PhaseViewer property), 1180		fipy.meshes.builders.utilityClasses	module, 154
fig	(fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.Matplotlib2DViewer property), 909	fipy.meshes.cylindricalGrid1D	module, 154
fig	(fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.MatplotlibViewer property), 911	fipy.meshes.cylindricalGrid2D	module, 154
fig	(fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer property), 914	fipy.meshes.cylindricalNonUniformGrid1D	module, 154
fig	(fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer property), 916	fipy.meshes.cylindricalNonUniformGrid2D	
fig	(fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer property), 919		

- module, 163
- fipy.meshes.cylindricalUniformGrid1D
 - module, 172
- fipy.meshes.cylindricalUniformGrid2D
 - module, 179
- fipy.meshes.factoryMeshes
 - module, 186
- fipy.meshes.gmshMesh
 - module, 189
- fipy.meshes.grid1D
 - module, 237
- fipy.meshes.grid2D
 - module, 237
- fipy.meshes.grid3D
 - module, 237
- fipy.meshes.mesh
 - module, 237
- fipy.meshes.mesh1D
 - module, 246
- fipy.meshes.mesh2D
 - module, 254
- fipy.meshes.nonUniformGrid1D
 - module, 263
- fipy.meshes.nonUniformGrid2D
 - module, 271
- fipy.meshes.nonUniformGrid3D
 - module, 280
- fipy.meshes.periodicGrid1D
 - module, 288
- fipy.meshes.periodicGrid2D
 - module, 297
- fipy.meshes.periodicGrid3D
 - module, 323
- fipy.meshes.representations
 - module, 379
- fipy.meshes.representations.abstractRepresentation
 - module, 379
- fipy.meshes.representations.gridRepresentation
 - module, 379
- fipy.meshes.representations.meshRepresentation
 - module, 379
- fipy.meshes.skewedGrid2D
 - module, 379
- fipy.meshes.sphericalNonUniformGrid1D
 - module, 388
- fipy.meshes.sphericalUniformGrid1D
 - module, 397
- fipy.meshes.test
 - module, 404
- fipy.meshes.topologies
 - module, 404
- fipy.meshes.topologies.abstractTopology
 - module, 404
- fipy.meshes.topologies.gridTopology
 - module, 404
- fipy.meshes.topologies.meshTopology
 - module, 404
- fipy.meshes.tri2D
 - module, 404
- fipy.meshes.uniformGrid
 - module, 413
- fipy.meshes.uniformGrid1D
 - module, 420
- fipy.meshes.uniformGrid2D
 - module, 427
- fipy.meshes.uniformGrid3D
 - module, 434
- fipy.numerix

- module, 1222
- fipy.solvers
 - module, 441
- fipy.solvers.convergence
 - module, 443
- fipy.solvers.petsc
 - module, 448
- fipy.solvers.petsc.comms
 - module, 449
- fipy.solvers.petsc.comms.parallelPETScCommWrapper
 - module, 449
- fipy.solvers.petsc.comms.petscCommWrapper
 - module, 450
- fipy.solvers.petsc.comms.serialPETScCommWrapper
 - module, 450
- fipy.solvers.petsc.dummySolver
 - module, 451
- fipy.solvers.petsc.linearBicgSolver
 - module, 451
- fipy.solvers.petsc.linearCGSolver
 - module, 453
- fipy.solvers.petsc.linearCGSSolver
 - module, 452
- fipy.solvers.petsc.linearGMRESSolver
 - module, 454
- fipy.solvers.petsc.linearLUSolver
 - module, 455
- fipy.solvers.petsc.petscConvergence
 - module, 456
- fipy.solvers.petsc.petscKrylovSolver
 - module, 461
- fipy.solvers.petsc.petscSolver
 - module, 462
- fipy.solvers.petsc.preconditioners
 - module, 463
- fipy.solvers.petsc.preconditioners.defaultPreconditioner
 - module, 464
- fipy.solvers.petsc.preconditioners.hyprePreconditioner
 - module, 465
- fipy.solvers.petsc.preconditioners.icPreconditioner
 - module, 465
- fipy.solvers.petsc.preconditioners.iluPreconditioner
 - module, 465
- fipy.solvers.petsc.preconditioners.jacobiPreconditioner
 - module, 465
- fipy.solvers.petsc.preconditioners.luPreconditioner
 - module, 466
- fipy.solvers.petsc.preconditioners.petscPreconditioner
 - module, 466
- fipy.solvers.petsc.preconditioners.ssorPreconditioner
 - module, 466
- fipy.solvers.preconditioner
 - module, 466
- fipy.solvers.pyAMG
 - module, 467
- fipy.solvers.pyAMG.preconditioners
 - module, 467
- fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner
 - module, 467
- fipy.solvers.pyamgx
 - module, 468
- fipy.solvers.pyamgx.aggregationAMGSolver
 - module, 468
- fipy.solvers.pyamgx.classicalAMGSolver
 - module, 469
- fipy.solvers.pyamgx.linearBiCGStabSolver
 - module, 470
- fipy.solvers.pyamgx.linearCGSolver

- module, 471
- fiPy.solvers.pyamgx.linearFGMRESSolver
 - module, 472
- fiPy.solvers.pyamgx.linearGMRESSolver
 - module, 473
- fiPy.solvers.pyamgx.linearLUSolver
 - module, 474
- fiPy.solvers.pyamgx.preconditioners
 - module, 475
- fiPy.solvers.pyamgx.pyamgxConvergence
 - module, 478
- fiPy.solvers.pyamgx.pyAMGXSolver
 - module, 477
- fiPy.solvers.pyamgx.smoother
 - module, 478
- fiPy.solvers.scipy
 - module, 479
- fiPy.solvers.scipy.linearBicgstabSolver
 - module, 479
- fiPy.solvers.scipy.linearCGSolver
 - module, 481
- fiPy.solvers.scipy.linearCGSSolver
 - module, 480
- fiPy.solvers.scipy.linearGMRESSolver
 - module, 482
- fiPy.solvers.scipy.linearLUSolver
 - module, 483
- fiPy.solvers.scipy.preconditioners
 - module, 484
- fiPy.solvers.scipy.preconditioners.iluPreconditioner
 - module, 484
- fiPy.solvers.scipy.preconditioners.jacobiPreconditioner
 - module, 484
- fiPy.solvers.scipy.preconditioners.scipyPreconditioner
 - module, 485
- fiPy.solvers.scipy.scipyConvergence
 - module, 485
- fiPy.solvers.scipy.scipyKrylovSolver
 - module, 486
- fiPy.solvers.scipy.scipySolver
 - module, 487
- fiPy.solvers.solver
 - module, 488
- fiPy.solvers.test
 - module, 497
- fiPy.solvers.trilinos
 - module, 497
- fiPy.solvers.trilinos.aztecConvergence
 - module, 497
- fiPy.solvers.trilinos.comms
 - module, 499
- fiPy.solvers.trilinos.comms.epetraCommWrapper
 - module, 499
- fiPy.solvers.trilinos.comms.parallelEpetraCommWrapper
 - module, 500
- fiPy.solvers.trilinos.comms.serialEpetraCommWrapper
 - module, 500
- fiPy.solvers.trilinos.linearBicgstabSolver
 - module, 501
- fiPy.solvers.trilinos.linearCGSolver
 - module, 503
- fiPy.solvers.trilinos.linearCGSSolver
 - module, 502
- fiPy.solvers.trilinos.linearGMRESSolver
 - module, 504
- fiPy.solvers.trilinos.linearLUSolver
 - module, 505
- fiPy.solvers.trilinos.preconditioners
 - module, 505
- fiPy.solvers.trilinos.preconditioners.domDecompPreconditioner
 - module, 507
- fiPy.solvers.trilinos.preconditioners.icPreconditioner
 - module, 508
- fiPy.solvers.trilinos.preconditioners.iluPreconditioner
 - module, 508
- fiPy.solvers.trilinos.preconditioners.jacobiPreconditioner
 - module, 508
- fiPy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner
 - module, 508
- fiPy.solvers.trilinos.preconditioners.multilevelDDPreconditioner
 - module, 509
- fiPy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner
 - module, 509
- fiPy.solvers.trilinos.preconditioners.multilevelPreconditioner
 - module, 509
- fiPy.solvers.trilinos.preconditioners.multilevelSAPreconditioner
 - module, 510
- fiPy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner
 - module, 510
- fiPy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner
 - module, 510
- fiPy.solvers.trilinos.preconditioners.trilinosPreconditioner
 - module, 511
- fiPy.solvers.trilinos.trilinosAztec00Solver
 - module, 511
- fiPy.solvers.trilinos.trilinosMLTest
 - module, 512
- fiPy.solvers.trilinos.trilinosNonlinearSolver
 - module, 513
- fiPy.solvers.trilinos.trilinosSolver
 - module, 515
- fiPy.steppers
 - module, 516
- fiPy.steppers.pidStepper
 - module, 518
- fiPy.steppers.pseudoRKQSS stepper
 - module, 518
- fiPy.steppers.stepper
 - module, 518
- fiPy.terms
 - module, 518
- fiPy.terms.abstractBinaryTerm
 - module, 525
- fiPy.terms.abstractConvectionTerm
 - module, 525
- fiPy.terms.abstractDiffusionTerm
 - module, 525
- fiPy.terms.abstractUpwindConvectionTerm
 - module, 525
- fiPy.terms.advectionTerm
 - module, 525
- fiPy.terms.asymmetricConvectionTerm
 - module, 532
- fiPy.terms.binaryTerm
 - module, 532
- fiPy.terms.cellTerm
 - module, 532
- fiPy.terms.centralDiffConvectionTerm
 - module, 536
- fiPy.terms.coupledBinaryTerm
 - module, 541
- fiPy.terms.diffusionTerm
 - module, 541
- fiPy.terms.diffusionTermCorrection
 - module, 545
- fiPy.terms.diffusionTermNoCorrection

- module, 548
- fiPy.terms.explicitDiffusionTerm
 - module, 552
- fiPy.terms.explicitSourceTerm
 - module, 555
- fiPy.terms.explicitUpwindConvectionTerm
 - module, 555
- fiPy.terms.exponentialConvectionTerm
 - module, 560
- fiPy.terms.faceTerm
 - module, 566
- fiPy.terms.firstOrderAdvectionTerm
 - module, 570
- fiPy.terms.hybridConvectionTerm
 - module, 575
- fiPy.terms.implicitDiffusionTerm
 - module, 580
- fiPy.terms.implicitDiffusionTerm.DiffusionTerm
 - object, 1004, 1009
- fiPy.terms.implicitSourceTerm
 - module, 580
- fiPy.terms.nonDiffusionTerm
 - module, 584
- fiPy.terms.powerLawConvectionTerm
 - module, 584
- fiPy.terms.residualTerm
 - module, 590
- fiPy.terms.sourceTerm
 - module, 594
- fiPy.terms.term
 - module, 598
- fiPy.terms.test
 - module, 601
- fiPy.terms.transientTerm
 - module, 601
- fiPy.terms.transientTerm.TransientTerm
 - object, 1004, 1009
- fiPy.terms.unaryTerm
 - module, 606
- fiPy.terms.upwindConvectionTerm
 - module, 606
- fiPy.terms.vanLeerConvectionTerm
 - module, 611
- fiPy.testFiPy
 - module, 616
- fiPy.tests
 - module, 616
- fiPy.tests.doctestPlus
 - module, 617
- fiPy.tests.lateImportTest
 - module, 619
- fiPy.tests.test
 - module, 619
- fiPy.tests.testProgram
 - module, 619
- fiPy.tools
 - module, 620
- fiPy.tools.comms
 - module, 636
- fiPy.tools.comms.commWrapper
 - module, 637
- fiPy.tools.comms.dummyComm
 - module, 637
- fiPy.tools.copy_script
 - module, 637
- fiPy.tools.debug
 - module, 638
- fiPy.tools.decorators
 - module, 638
- fiPy.tools.dimensions
 - module, 638
- fiPy.tools.dimensions.DictWithDefault
 - module, 639
- fiPy.tools.dimensions.NumberDict
 - module, 639
- fiPy.tools.dimensions.physicalField
 - module, 639
- fiPy.tools.dump
 - module, 663, 1173
- fiPy.tools.inline
 - module, 664
- fiPy.tools.logging
 - module, 664
- fiPy.tools.logging.environment
 - module, 665
- fiPy.tools.numerix
 - module, 666
- fiPy.tools.parser
 - module, 671, 1109, 1170
- fiPy.tools.sharedtempfile
 - module, 672
- fiPy.tools.test
 - module, 673
- fiPy.tools.timer
 - module, 673
- fiPy.tools.vector
 - module, 673
- fiPy.variables
 - module, 674
- fiPy.variables.addOverFacesVariable
 - module, 676
- fiPy.variables.arithmeticCellToFaceVariable
 - module, 676
- fiPy.variables.betaNoiseVariable
 - module, 676
- fiPy.variables.binaryOperatorVariable
 - module, 692
- fiPy.variables.cellToFaceVariable
 - module, 692
- fiPy.variables.cellVariable
 - module, 692
- fiPy.variables.cellVariable.CellVariable
 - object, 1003, 1008
- fiPy.variables.constant
 - module, 707
- fiPy.variables.constraintMask
 - module, 707
- fiPy.variables.coupledCellVariable
 - module, 707
- fiPy.variables.distanceVariable
 - module, 707
- fiPy.variables.exponentialNoiseVariable
 - module, 725
- fiPy.variables.faceGradContributionsVariable
 - module, 741
- fiPy.variables.faceGradVariable
 - module, 741
- fiPy.variables.faceVariable
 - module, 741
- fiPy.variables.gammaNoiseVariable
 - module, 752
- fiPy.variables.gaussCellGradVariable
 - module, 768
- fiPy.variables.gaussianNoiseVariable
 - module, 768
- fiPy.variables.harmonicCellToFaceVariable

module, 784
 fipy.variables.histogramVariable
 module, 784
 fipy.variables.interfaceAreaVariable
 module, 798
 fipy.variables.interfaceFlagVariable
 module, 798
 fipy.variables.leastSquaresCellGradVariable
 module, 798
 fipy.variables.levelSetDiffusionVariable
 module, 798
 fipy.variables.meshVariable
 module, 798
 fipy.variables.minmodCellToFaceVariable
 module, 809
 fipy.variables.modCellGradVariable
 module, 809
 fipy.variables.modCellToFaceVariable
 module, 809
 fipy.variables.modFaceGradVariable
 module, 809
 fipy.variables.modPhysicalField
 module, 809
 fipy.variables.modularVariable
 module, 809
 fipy.variables.noiseVariable
 module, 824
 fipy.variables.operatorVariable
 module, 839
 fipy.variables.scharfetterGummelFaceVariable
 module, 839
 fipy.variables.surfactantConvectionVariable
 module, 850
 fipy.variables.surfactantVariable
 module, 861
 fipy.variables.test
 module, 877
 fipy.variables.unaryOperatorVariable
 module, 877
 fipy.variables.uniformNoiseVariable
 module, 877
 fipy.variables.variable
 module, 892
 fipy.viewers, 1225
 module, 901, 972, 981, 982, 984, 988, 991, 992, 1024, 1044,
 1047, 1051, 1093, 1140, 1147, 1161, 1172, 1176, 1201
 fipy.viewers.matplotlibViewer
 module, 905
 fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer
 module, 908
 fipy.viewers.matplotlibViewer.abstractMatplotlibViewer
 module, 910
 fipy.viewers.matplotlibViewer.matplotlib1DViewer
 module, 913
 fipy.viewers.matplotlibViewer.matplotlib2DContourViewer
 module, 915
 fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer
 module, 918
 fipy.viewers.matplotlibViewer.matplotlib2DGridViewer
 module, 921
 fipy.viewers.matplotlibViewer.matplotlib2DViewer
 module, 924
 fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer
 module, 926
 fipy.viewers.matplotlibViewer.matplotlibStreamViewer
 module, 927
 fipy.viewers.matplotlibViewer.matplotlibVectorViewer
 module, 931

fipy.viewers.matplotlibViewer.test
 module, 934
 fipy.viewers.mayaviViewer
 module, 934
 fipy.viewers.mayaviViewer.mayaviClient
 module, 938
 fipy.viewers.mayaviViewer.mayaviDaemon
 module, 941
 fipy.viewers.mayaviViewer.test
 module, 942
 fipy.viewers.multiViewer
 module, 942
 fipy.viewers.test
 module, 943
 fipy.viewers.testinteractive
 module, 944
 fipy.viewers.tsvViewer
 module, 944
 fipy.viewers.tsvViewer.TSVViewer
 object, 1005, 1010
 fipy.viewers.viewer
 module, 946
 fipy.viewers.vtkViewer
 module, 948
 fipy.viewers.vtkViewer.test
 module, 952
 fipy.viewers.vtkViewer.vtkCellViewer
 module, 952
 fipy.viewers.vtkViewer.vtkFaceViewer
 module, 954
 fipy.viewers.vtkViewer.vtkViewer
 module, 956
 FIPY_INCLUDE_NUMERIX_ALL, 1230
 FIPY_LOG_CONFIG, 16
 FIPY_SOLVERS, 18, 31, 32, 82
 FirstOrderAdvectionTerm (*class in*
 fipy.terms.firstOrderAdvectionTerm), 570
 FixedFlux, 1221
 FixedFlux (*class in fipy.boundaryConditions.fixedFlux*), 142
 FixedValue, 1221, 1223
 FixedValue (*class in fipy.boundaryConditions.fixedValue*), 142
 floor() (*fipy.tools.dimensions.physicalField.PhysicalField method*),
 651
 floor() (*fipy.tools.PhysicalField method*), 630
 fps (*fipy.viewers.mayaviViewer.MayaviClient property*), 936
 fps (*fipy.viewers.mayaviViewer.mayaviClient.MayaviClient property*),
 940

G

GammaNoiseVariable (*class in fipy.variables.gammaNoiseVariable*),
 752
 gaussGrad (*fipy.variables.betaNoiseVariable.BetaNoiseVariable*
 property), 686
 gaussGrad (*fipy.variables.cellVariable.CellVariable property*), 701
 gaussGrad (*fipy.variables.distanceVariable.DistanceVariable*
 property), 720
 gaussGrad
 (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable*
 property), 735
 gaussGrad (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable*
 property), 762
 gaussGrad
 (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable*
 property), 778
 gaussGrad (*fipy.variables.histogramVariable.HistogramVariable*
 property), 793
 gaussGrad (*fipy.variables.modularVariable.ModularVariable*
 property), 819

gaussGrad (*fipy.variables.noiseVariable.NoiseVariable* property), 833
gaussGrad (*fipy.variables.surfactantVariable.SurfactantVariable* property), 871
gaussGrad (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 886
GaussianNoiseVariable (class in *fipy.variables.gaussianNoiseVariable*), 768
GaussSeidelPreconditioner (class in *fipy.solvers.pyamgx.preconditioners*), 476
GeneralSolver (in module *fipy.solvers*), 442
getShape() (in module *fipy.tools.numerix*), 668
getUnit() (in module *fipy.tools.numerix*), 669
Gist1DViewer, 1224
GitHub Actions, 122
globalValue (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 686
globalValue (*fipy.variables.cellVariable.CellVariable* property), 701
globalValue (*fipy.variables.distanceVariable.DistanceVariable* property), 720
globalValue (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 735
globalValue (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 762
globalValue (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 778
globalValue (*fipy.variables.histogramVariable.HistogramVariable* property), 793
globalValue (*fipy.variables.modularVariable.ModularVariable* property), 819
globalValue (*fipy.variables.noiseVariable.NoiseVariable* property), 833
globalValue (*fipy.variables.surfactantVariable.SurfactantVariable* property), 871
globalValue (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 886
GMRES, 122
Gmsh, 122
gmsh module, 1107, 1115, 1126
Gmsh2D (class in *fipy.meshes.gmshMesh*), 189
Gmsh2DIn3DSpace (class in *fipy.meshes.gmshMesh*), 201
Gmsh3D (class in *fipy.meshes.gmshMesh*), 210
GmshException, 218
GmshFile (class in *fipy.meshes.gmshMesh*), 218
GmshGrid2D (class in *fipy.meshes.gmshMesh*), 219
GmshGrid3D (class in *fipy.meshes.gmshMesh*), 227
gmshVersion() (in module *fipy.meshes.gmshMesh*), 236
grad (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 686
grad (*fipy.variables.cellVariable.CellVariable* property), 702
grad (*fipy.variables.distanceVariable.DistanceVariable* property), 720
grad (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 736
grad (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 762
grad (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 779
grad (*fipy.variables.histogramVariable.HistogramVariable* property), 793
grad (*fipy.variables.modularVariable.ModularVariable* property), 819
grad (*fipy.variables.noiseVariable.NoiseVariable* property), 834

grad (*fipy.variables.surfactantVariable.SurfactantVariable* property), 871
grad (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 887
Grid1D, 980, 983, 987, 990, 1050, 1098, 1141, 1155, 1174, 1194, 1200, 1223
Grid1D() (in module *fipy.meshes.factoryMeshes*), 187
Grid2D, 970, 1043, 1046, 1091, 1101, 1103, 1110, 1138, 1170, 1221
Grid2D() (in module *fipy.meshes.factoryMeshes*), 187
Grid2DViewer, 1223
Grid3D() (in module *fipy.meshes.factoryMeshes*), 188

H

HappyBreakdownConvergence (class in *fipy.solvers.convergence*), 446
harmonicFaceValue (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 686
harmonicFaceValue (*fipy.variables.cellVariable.CellVariable* property), 702
harmonicFaceValue (*fipy.variables.distanceVariable.DistanceVariable* property), 720
harmonicFaceValue (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 736
harmonicFaceValue (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 762
harmonicFaceValue (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 779
harmonicFaceValue (*fipy.variables.histogramVariable.HistogramVariable* property), 793
harmonicFaceValue (*fipy.variables.modularVariable.ModularVariable* property), 819
harmonicFaceValue (*fipy.variables.noiseVariable.NoiseVariable* property), 834
harmonicFaceValue (*fipy.variables.surfactantVariable.SurfactantVariable* property), 871
harmonicFaceValue (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 887
HistogramVariable (class in *fipy.variables.histogramVariable*), 784
HybridConvectionTerm (class in *fipy.terms.hybridConvectionTerm*), 575
HYPREPreconditioner, 1162
HYPREPreconditioner (class in *fipy.solvers.petsc.preconditioners*), 463
HYPREPreconditioner (class in *fipy.solvers.petsc.preconditioners.hyprePreconditioner*), 465
ICPreconditioner (class in *fipy.solvers.petsc.preconditioners*), 463
ICPreconditioner (class in *fipy.solvers.petsc.preconditioners.icPreconditioner*), 465
ICPreconditioner (class in *fipy.solvers.trilinos.preconditioners*), 505
ICPreconditioner (class in *fipy.solvers.trilinos.preconditioners.icPreconditioner*), 508
id (examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer property), 1121
id (examples.phase.phaseViewer.PhaseViewer property), 1180

id	(fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer property), 909	inBaseUnits() (fipy.variables.histogramVariable.HistogramVariable method), 806
id	(fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer property), 911	inBaseUnits() (fipy.variables.meshVariable.MeshVariable method), 806
id	(fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer property), 914	inBaseUnits() (fipy.variables.modularVariable.ModularVariable method), 819
id	(fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer property), 916	inBaseUnits() (fipy.variables.noiseVariable.NoiseVariable method), 834
id	(fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer property), 919	inBaseUnits() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 847
id	(fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer property), 922	inBaseUnits() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 858
id	(fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer property), 925	inBaseUnits() (fipy.variables.surfactantVariable.SurfactantVariable method), 872
id	(fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer property), 929	inBaseUnits() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 887
id	(fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer property), 933	inBaseUnits() (fipy.variables.variable.Variable method), 899
IllConditionedDivergence (class in fipy.solvers.convergence), 446		IncorrectSolutionVariable, 520
IllConditionedPreconditionerDivergence (class in fipy.solvers.convergence), 446		inDimensionless() (fipy.tools.dimensions.physicalField.PhysicalField method), 651
IllConditionedPreconditionerWarning, 489		inDimensionless() (fipy.tools.PhysicalField method), 630
IllegalInputOrBreakdownWarning, 490		inRadians() (fipy.tools.dimensions.physicalField.PhysicalField method), 652
ILUPreconditioner (class in fipy.solvers.petsc.preconditioners), 463		inRadians() (fipy.tools.PhysicalField method), 630
ILUPreconditioner (class in fipy.solvers.petsc.preconditioners.iluPreconditioner), 465		inSIUnits() (fipy.tools.dimensions.physicalField.PhysicalField method), 652
ILUPreconditioner (class in fipy.solvers.pyamgx.preconditioners), 476		inSIUnits() (fipy.tools.PhysicalField method), 630
ILUPreconditioner (class in fipy.solvers.scipy.preconditioners), 484		interfaceVar (fipy.variables.surfactantVariable.SurfactantVariable property), 873
ILUPreconditioner (class in fipy.solvers.scipy.preconditioners.iluPreconditioner), 484		inUnitsOf() (fipy.tools.dimensions.physicalField.PhysicalField method), 652
ILUPreconditioner (class in fipy.solvers.trilinos.preconditioners), 505		inUnitsOf() (fipy.tools.PhysicalField method), 631
ILUPreconditioner (class in fipy.solvers.trilinos.preconditioners.iluPreconditioner), 508		inUnitsOf() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 687
ImplicitDiffusionTerm (in module fipy.terms.implicitDiffusionTerm), 580		inUnitsOf() (fipy.variables.cellVariable.CellVariable method), 702
ImplicitSourceTerm, 1140, 1171, 1175, 1204		inUnitsOf() (fipy.variables.distanceVariable.DistanceVariable method), 721
ImplicitSourceTerm (class in fipy.terms.implicitSourceTerm), 580		inUnitsOf() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 736
inBaseUnits() (fipy.tools.dimensions.physicalField.PhysicalField method), 651		inUnitsOf() (fipy.variables.faceVariable.FaceVariable method), 749
inBaseUnits() (fipy.tools.PhysicalField method), 630		inUnitsOf() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 763
inBaseUnits() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 687		inUnitsOf() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 779
inBaseUnits() (fipy.variables.cellVariable.CellVariable method), 702		inUnitsOf() (fipy.variables.histogramVariable.HistogramVariable method), 794
inBaseUnits() (fipy.variables.distanceVariable.DistanceVariable method), 721		inUnitsOf() (fipy.variables.meshVariable.MeshVariable method), 806
inBaseUnits() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 736		inUnitsOf() (fipy.variables.modularVariable.ModularVariable method), 820
inBaseUnits() (fipy.variables.faceVariable.FaceVariable method), 749		inUnitsOf() (fipy.variables.noiseVariable.NoiseVariable method), 834
inBaseUnits() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 763		inUnitsOf() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 847
inBaseUnits() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 779		inUnitsOf() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 858
		inUnitsOf() (fipy.variables.surfactantVariable.SurfactantVariable method), 872
		inUnitsOf()

- `(fipy.variables.uniformNoiseVariable.UniformNoiseVariable method)`, 887
 - `inUnitsOf()` (*fipy.variables.variable.Variable method*), 899
 - IPython, 122
 - `isAngle()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 662
 - `isclose()` (*in module fipy.tools.numerix*), 669
 - `isCompatible()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 662
 - `isDimensionless()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 662
 - `isDimensionlessOrAngle()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 662
 - `isInverseAngle()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 663
 - IteratingConvergence (*class in fipy.solvers.convergence*), 446
 - IterationConvergence (*class in fipy.solvers.convergence*), 446
 - IterationDivergence (*class in fipy.solvers.convergence*), 447
 - iterations (*fipy.solvers.convergence.ConvergenceBase attribute*), 444
 - Iterator, 1222
- ## J
- JacobiPreconditioner (*class in fipy.solvers.petsc.preconditioners*), 463
 - JacobiPreconditioner (*class in fipy.solvers.petsc.preconditioners.jacobiPreconditioner*), 465
 - JacobiPreconditioner (*class in fipy.solvers.pyamgx.preconditioners*), 476
 - JacobiPreconditioner (*class in fipy.solvers.scipy.preconditioners*), 484
 - JacobiPreconditioner (*class in fipy.solvers.scipy.preconditioners.jacobiPreconditioner*), 484
 - JacobiPreconditioner (*class in fipy.solvers.trilinos.preconditioners*), 506
 - JacobiPreconditioner (*class in fipy.solvers.trilinos.preconditioners.jacobiPreconditioner*), 508
 - JOR, 122
 - JSON, 122
 - `justErrorVector()` (*fipy.terms.advectionTerm.AdvectionTerm method*), 529
 - `justErrorVector()` (*fipy.terms.cellTerm.CellTerm method*), 533
 - `justErrorVector()` (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm method*), 538
 - `justErrorVector()` (*fipy.terms.diffusionTerm.DiffusionTerm method*), 542
 - `justErrorVector()` (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection method*), 545
 - `justErrorVector()` (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection method*), 549
 - `justErrorVector()` (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm method*), 552
 - `justErrorVector()` (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm method*), 558
 - `justErrorVector()` (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm method*), 563
 - `justErrorVector()` (*fipy.terms.faceTerm.FaceTerm method*), 567
 - `justErrorVector()` (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm method*), 572
 - `justErrorVector()` (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm method*), 577
 - `justErrorVector()` (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm method*), 582
 - `justErrorVector()` (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm method*), 587
 - `justErrorVector()` (*fipy.terms.residualTerm.ResidualTerm method*), 591
 - `justErrorVector()` (*fipy.terms.sourceTerm.SourceTerm method*), 595
 - `justErrorVector()` (*fipy.terms.term.Term method*), 599
 - `justErrorVector()` (*fipy.terms.transientTerm.TransientTerm method*), 603
 - `justErrorVector()` (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm method*), 609
 - `justErrorVector()` (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm method*), 613
 - `justResidualVector()` (*fipy.terms.advectionTerm.AdvectionTerm method*), 529
 - `justResidualVector()` (*fipy.terms.cellTerm.CellTerm method*), 534
 - `justResidualVector()` (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm method*), 539
 - `justResidualVector()` (*fipy.terms.diffusionTerm.DiffusionTerm method*), 543
 - `justResidualVector()` (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection method*), 546
 - `justResidualVector()` (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection method*), 549
 - `justResidualVector()` (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm method*), 553
 - `justResidualVector()` (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm method*), 558
 - `justResidualVector()` (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm method*), 563
 - `justResidualVector()` (*fipy.terms.faceTerm.FaceTerm method*), 568
 - `justResidualVector()` (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm method*), 573
 - `justResidualVector()` (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm method*), 578
 - `justResidualVector()` (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm method*), 582
 - `justResidualVector()` (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm method*), 587
 - `justResidualVector()` (*fipy.terms.residualTerm.ResidualTerm method*), 591
 - `justResidualVector()` (*fipy.terms.sourceTerm.SourceTerm method*), 596
 - `justResidualVector()` (*fipy.terms.term.Term method*), 599
 - `justResidualVector()` (*fipy.terms.transientTerm.TransientTerm method*), 603

method), 604
 justResidualVector()
 (fipy.terms.upwindConvectionTerm.UpwindConvectionTerm
 method), 609
 justResidualVector()
 (fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm
 method), 614

K

KSP_AbsoluteToleranceConvergence (class in
 fipy.solvers.petsc.petscConvergence), 457
 KSP_BreakdownBICGDivergence (class in
 fipy.solvers.petsc.petscConvergence), 457
 KSP_BreakdownDivergence (class in
 fipy.solvers.petsc.petscConvergence), 458
 KSP_HappyBreakdownConvergence (class in
 fipy.solvers.petsc.petscConvergence), 458
 KSP_IndefiniteMatrixDivergence (class in
 fipy.solvers.petsc.petscConvergence), 458
 KSP_IndefinitePreconditionerDivergence (class in
 fipy.solvers.petsc.petscConvergence), 458
 KSP_IteratingConvergence (class in
 fipy.solvers.petsc.petscConvergence), 458
 KSP_IterationConvergence (class in
 fipy.solvers.petsc.petscConvergence), 459
 KSP_IterationDivergence (class in
 fipy.solvers.petsc.petscConvergence), 459
 KSP_NaNOrInfDivergence (class in
 fipy.solvers.petsc.petscConvergence), 459
 KSP_NonSymmetricDivergence (class in
 fipy.solvers.petsc.petscConvergence), 459
 KSP_NormalAbsoluteToleranceConvergence (class in
 fipy.solvers.petsc.petscConvergence), 459
 KSP_NormalRelativeToleranceConvergence (class in
 fipy.solvers.petsc.petscConvergence), 459
 KSP_NullDivergence (class in fipy.solvers.petsc.petscConvergence),
 460
 KSP_PreconditioningDivergence (class in
 fipy.solvers.petsc.petscConvergence), 460
 KSP_RelativeToleranceConvergence (class in
 fipy.solvers.petsc.petscConvergence), 460
 KSP_ToleranceDivergence (class in
 fipy.solvers.petsc.petscConvergence), 460
 kwargs
 (fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer.
 property), 929

L

L1error() (in module fipy.steps), 516
 L1norm() (in module fipy.tools.numerix), 667
 L2error() (in module fipy.steps), 516
 L2norm() (in module fipy.tools.numerix), 667
 leastSquaresGrad
 (fipy.variables.betaNoiseVariable.BetaNoiseVariable
 property), 688
 leastSquaresGrad (fipy.variables.cellVariable.CellVariable
 property), 703
 leastSquaresGrad (fipy.variables.distanceVariable.DistanceVariable
 property), 721
 leastSquaresGrad
 (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable
 property), 737
 leastSquaresGrad
 (fipy.variables.gammaNoiseVariable.GammaNoiseVariable
 property), 763

leastSquaresGrad
 (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable
 property), 780
 leastSquaresGrad
 (fipy.variables.histogramVariable.HistogramVariable
 property), 794
 leastSquaresGrad (fipy.variables.modularVariable.ModularVariable
 property), 820
 leastSquaresGrad (fipy.variables.noiseVariable.NoiseVariable
 property), 835
 leastSquaresGrad
 (fipy.variables.surfactantVariable.SurfactantVariable
 property), 873
 leastSquaresGrad
 (fipy.variables.uniformNoiseVariable.UniformNoiseVariable
 property), 888
 LEGACY_TOLERANCE (fipy.solvers.petsc.dummySolver.DummySolver
 attribute), 451
 LEGACY_TOLERANCE
 (fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver
 attribute), 452
 LEGACY_TOLERANCE
 (fipy.solvers.petsc.linearCGSolver.LinearCGSolver
 attribute), 454
 LEGACY_TOLERANCE
 (fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver
 attribute), 453
 LEGACY_TOLERANCE
 (fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver
 attribute), 455
 LEGACY_TOLERANCE
 (fipy.solvers.petsc.linearLUSolver.LinearLUSolver
 attribute), 456
 LEGACY_TOLERANCE
 (fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver
 attribute), 462
 LEGACY_TOLERANCE (fipy.solvers.petsc.petscSolver.PETScSolver
 attribute), 463
 LEGACY_TOLERANCE
 (fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver
 attribute), 469
 LEGACY_TOLERANCE
 (fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver
 attribute), 470
 LEGACY_TOLERANCE
 (fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver
 attribute), 471
 LEGACY_TOLERANCE
 (fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver
 attribute), 472
 LEGACY_TOLERANCE
 (fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver
 attribute), 473
 LEGACY_TOLERANCE
 (fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver
 attribute), 474
 LEGACY_TOLERANCE
 (fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver
 attribute), 474
 LEGACY_TOLERANCE
 (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver
 attribute), 477
 LEGACY_TOLERANCE
 (fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver
 attribute), 480
 LEGACY_TOLERANCE
 (fipy.solvers.scipy.linearCGSolver.LinearCGSolver
 attribute), 481

- LEGACY_TOLERANCE (fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver attribute), 481
- LEGACY_TOLERANCE (fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver attribute), 482
- LEGACY_TOLERANCE (fipy.solvers.scipy.linearLUSolver.LinearLUSolver attribute), 483
- LEGACY_TOLERANCE (fipy.solvers.scipy.scipyKrylovSolver.ScipyKrylovSolver attribute), 486
- LEGACY_TOLERANCE (fipy.solvers.scipy.scipySolver.ScipySolver attribute), 487
- LEGACY_TOLERANCE (fipy.solvers.solver.Solver attribute), 495
- LEGACY_TOLERANCE (fipy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver attribute), 501
- LEGACY_TOLERANCE (fipy.solvers.trilinos.linearCGSolver.LinearCGSolver attribute), 503
- LEGACY_TOLERANCE (fipy.solvers.trilinos.linearCGSSolver.LinearCGSSolver attribute), 502
- LEGACY_TOLERANCE (fipy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver attribute), 504
- LEGACY_TOLERANCE (fipy.solvers.trilinos.linearLUSolver.LinearLUSolver attribute), 505
- LEGACY_TOLERANCE (fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver attribute), 512
- LEGACY_TOLERANCE (fipy.solvers.trilinos.trilinosMLTest.TrilinosMLTest attribute), 513
- LEGACY_TOLERANCE (fipy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver attribute), 514
- LEGACY_TOLERANCE (fipy.solvers.trilinos.trilinosSolver.TrilinosSolver attribute), 515
- levels (fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer property), 917
- levels (fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer property), 920
- LinearBicgSolver (class in fipy.solvers.petsc.linearBicgSolver), 451
- LinearBicgstabSolver, 1162
- LinearBiCGStabSolver (class in fipy.solvers.pyamgx.linearBiCGStabSolver), 470
- LinearBicgstabSolver (class in fipy.solvers.scipy.linearBicgstabSolver), 479
- LinearBicgstabSolver (class in fipy.solvers.trilinos.linearBicgstabSolver), 501
- LinearCGSolver (class in fipy.solvers.petsc.linearCGSolver), 453
- LinearCGSolver (class in fipy.solvers.pyamgx.linearCGSolver), 471
- LinearCGSolver (class in fipy.solvers.scipy.linearCGSolver), 481
- LinearCGSolver (class in fipy.solvers.trilinos.linearCGSolver), 503
- LinearCGSSolver (class in fipy.solvers.petsc.linearCGSSolver), 452
- LinearCGSSolver (class in fipy.solvers.scipy.linearCGSSolver), 480
- LinearCGSSolver (class in fipy.solvers.trilinos.linearCGSSolver), 502
- LinearFGMRESSolver (class in fipy.solvers.pyamgx.linearFGMRESSolver), 472
- LinearGMRESSolver (class in fipy.solvers.petsc.linearGMRESSolver), 454
- LinearGMRESSolver (class in fipy.solvers.pyamgx.linearGMRESSolver), 473
- LinearGMRESSolver (class in fipy.solvers.scipy.linearGMRESSolver), 482
- LinearGMRESSolver (class in fipy.solvers.trilinos.linearGMRESSolver), 504
- LinearLUSolver, 971, 1222
- LinearLUSolver (class in fipy.solvers.petsc.linearLUSolver), 455
- LinearLUSolver (class in fipy.solvers.pyamgx.linearLUSolver), 474
- LinearLUSolver (class in fipy.solvers.scipy.linearLUSolver), 483
- LinearLUSolver (class in fipy.solvers.trilinos.linearLUSolver), 505
- LinearPCGSolver (in module fipy.solvers.petsc.linearCGSolver), 454
- LinearPCGSolver (in module fipy.solvers.pyamgx.linearCGSolver), 472
- LinearPCGSolver (in module fipy.solvers.scipy.linearCGSolver), 482
- LinearPCGSolver (in module fipy.solvers.trilinos.linearCGSolver), 503
- lines (examples.phase.phaseViewer.PhaseViewer property), 1180
- lines (fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer property), 914
- LINError() (in module fipy.steps), 516
- LINFnorm() (in module fipy.tools.numerix), 667
- linux, 122
- loadtxt, 1114, 1141, 1172, 1176
- log, 1146, 1160, 1196
- log (examples.phase.phaseViewer.PhaseViewer property), 1180
- log (fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer property), 909
- log (fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer property), 911
- log (fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer property), 914
- log (fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer property), 917
- log (fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer property), 920
- log (fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer property), 922
- log (fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer property), 925
- log (fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer property), 930
- log (fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer property), 933
- log() (fipy.tools.dimensions.physicalField.PhysicalField method), 653
- log() (fipy.tools.physicalField.PhysicalField method), 631
- log10() (fipy.tools.dimensions.physicalField.PhysicalField method), 653
- log10() (fipy.tools.physicalField.PhysicalField method), 631
- LossOfAccuracyConvergence (class in fipy.solvers.convergence), 447
- LossOfPrecisionWarning, 490
- LU, 122
- LUPreconditioner (class in fipy.solvers.petsc.preconditioners), 463
- LUPreconditioner (class in fipy.solvers.petsc.preconditioners.luPreconditioner), 466

M

- macOS, 122
- mag (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 688
- mag (*fipy.variables.cellVariable.CellVariable* property), 703
- mag (*fipy.variables.distanceVariable.DistanceVariable* property), 722
- mag (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 737
- mag (*fipy.variables.faceVariable.FaceVariable* property), 749
- mag (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 764
- mag (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 780
- mag (*fipy.variables.histogramVariable.HistogramVariable* property), 795
- mag (*fipy.variables.meshVariable.MeshVariable* property), 806
- mag (*fipy.variables.modularVariable.ModularVariable* property), 820
- mag (*fipy.variables.noiseVariable.NoiseVariable* property), 835
- mag (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 847
- mag (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 859
- mag (*fipy.variables.surfactantVariable.SurfactantVariable* property), 873
- mag (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 888
- mag (*fipy.variables.variable.Variable* property), 899
- main (in module *fipy.tests.testProgram*), 619
- main() (in module *fipy.viewers.mayaViViewer.mayaViDaemon*), 942
- makeMapVariables() (*fipy.meshes.gmshMesh.MSHFile* method), 235
- Matplotlib, 122
- Matplotlib1DViewer (class in *fipy.viewers.matplotlibViewer.matplotlib1DViewer*), 913
- Matplotlib2DContourViewer (class in *fipy.viewers.matplotlibViewer.matplotlib2DContourViewer*), 915
- Matplotlib2DGridContourViewer (class in *fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer*), 918
- Matplotlib2DGridViewer (class in *fipy.viewers.matplotlibViewer.matplotlib2DGridViewer*), 921
- Matplotlib2DViewer (class in *fipy.viewers.matplotlibViewer.matplotlib2DViewer*), 924
- MatplotlibSparseMatrixViewer (class in *fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer*), 926
- MatplotlibStreamViewer (class in *fipy.viewers.matplotlibViewer.matplotlibStreamViewer*), 927
- MatplotlibSurfactantViewer (class in *examples.levelSet.electroChem.matplotlibSurfactantViewer*), 1118
- MatplotlibVectorViewer (class in *fipy.viewers.matplotlibViewer.matplotlibVectorViewer*), 931
- MatplotlibViewer() (in module *fipy.viewers.matplotlibViewer*), 905
- matrix, 1094
- matrix (*fipy.terms.advectionTerm.AdvectionTerm* property), 530
- matrix (*fipy.terms.cellTerm.CellTerm* property), 535
- matrix (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* property), 540
- matrix (*fipy.terms.diffusionTerm.DiffusionTerm* property), 543
- matrix (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* property), 547
- matrix (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* property), 550
- matrix (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* property), 554
- matrix (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* property), 559
- matrix (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* property), 564
- matrix (*fipy.terms.faceTerm.FaceTerm* property), 568
- matrix (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* property), 574
- matrix (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* property), 579
- matrix (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* property), 583
- matrix (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* property), 588
- matrix (*fipy.terms.residualTerm.ResidualTerm* property), 592
- matrix (*fipy.terms.sourceTerm.SourceTerm* property), 596
- matrix (*fipy.terms.term.Term* property), 600
- matrix (*fipy.terms.transientTerm.TransientTerm* property), 605
- matrix (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* property), 610
- matrix (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* property), 615
- MatrixIllConditionedWarning, 491
- MatrixModifyingPreconditioner (class in *fipy.solvers.preconditioner*), 466
- max() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 688
- max() (*fipy.variables.cellVariable.CellVariable* method), 703
- max() (*fipy.variables.distanceVariable.DistanceVariable* method), 722
- max() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 737
- max() (*fipy.variables.faceVariable.FaceVariable* method), 749
- max() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 764
- max() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 780
- max() (*fipy.variables.histogramVariable.HistogramVariable* method), 795
- max() (*fipy.variables.meshVariable.MeshVariable* method), 807
- max() (*fipy.variables.modularVariable.ModularVariable* method), 821
- max() (*fipy.variables.noiseVariable.NoiseVariable* method), 835
- max() (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 847
- max() (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 859
- max() (*fipy.variables.surfactantVariable.SurfactantVariable* method), 873
- max() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 888
- max() (*fipy.variables.variable.Variable* method), 899
- MaxAll() (*fipy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper* method), 500
- MaximumIterationWarning, 491
- MayaVi, 122
- Mayavi, 122
- MayaviClient (class in *fipy.viewers.mayaViViewer*), 934
- MayaviClient (class in *fipy.viewers.mayaViViewer.mayaViClient*), 938

- MayaviDaemon (class in fipy.viewers.mayaviViewer.mayaviDaemon), 941
- MayaviSurfactantViewer, 1113
- MayaviSurfactantViewer (class in examples.levelSet.electroChem.mayaviSurfactantViewer), 1122
- Mesh (class in fipy.meshes.mesh), 237
- Mesh1D (class in fipy.meshes.mesh1D), 246
- Mesh2D (class in fipy.meshes.mesh2D), 254
- MeshAdditionError, 153, 245
- MeshDimensionError, 903
- MeshExportError, 235
- MeshVariable (class in fipy.variables.meshVariable), 798
- method1() (package.subpackage.base.Base method), 137
- method1() (package.subpackage.object.Object method), 138
- method2() (package.subpackage.base.Base method), 137
- method2() (package.subpackage.object.Object method), 138
- min() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 688
- min() (fipy.variables.cellVariable.CellVariable method), 703
- min() (fipy.variables.distanceVariable.DistanceVariable method), 722
- min() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 737
- min() (fipy.variables.faceVariable.FaceVariable method), 749
- min() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 764
- min() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 780
- min() (fipy.variables.histogramVariable.HistogramVariable method), 795
- min() (fipy.variables.meshVariable.MeshVariable method), 807
- min() (fipy.variables.modularVariable.ModularVariable method), 821
- min() (fipy.variables.noiseVariable.NoiseVariable method), 835
- min() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 847
- min() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 859
- min() (fipy.variables.surfactantVariable.SurfactantVariable method), 873
- min() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 888
- min() (fipy.variables.variable.Variable method), 900
- MinAll() (fipy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper method), 500
- minmodFaceValue (fipy.variables.betaNoiseVariable.BetaNoiseVariable property), 688
- minmodFaceValue (fipy.variables.cellVariable.CellVariable property), 703
- minmodFaceValue (fipy.variables.distanceVariable.DistanceVariable property), 722
- minmodFaceValue (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable property), 737
- minmodFaceValue (fipy.variables.gammaNoiseVariable.GammaNoiseVariable property), 764
- minmodFaceValue (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable property), 780
- minmodFaceValue (fipy.variables.histogramVariable.HistogramVariable property), 795
- minmodFaceValue (fipy.variables.modularVariable.ModularVariable property), 821
- minmodFaceValue (fipy.variables.noiseVariable.NoiseVariable property), 835
- minmodFaceValue (fipy.variables.surfactantVariable.SurfactantVariable property), 873
- minmodFaceValue (fipy.variables.uniformNoiseVariable.UniformNoiseVariable property), 888
- ModularVariable, 1171
- ModularVariable (class in fipy.variables.modularVariable), 809
- module
 - examples, 959
 - examples.benchmarking, 960
 - examples.benchmarking.benchmark, 960
 - examples.benchmarking.size, 960
 - examples.benchmarking.steps, 960
 - examples.benchmarking.utils, 960
 - examples.benchmarking.versions, 960
 - examples.cahnHilliard, 960
 - examples.cahnHilliard.mesh2D, 961
 - examples.cahnHilliard.mesh2DCoupled, 963
 - examples.cahnHilliard.mesh3D, 966
 - examples.cahnHilliard.sphere, 968
 - examples.cahnHilliard.sphereDaemon, 970
 - examples.cahnHilliard.tanh1D, 970
 - examples.cahnHilliard.test, 972
 - examples.chemotaxis, 972
 - examples.chemotaxis.input, 973
 - examples.chemotaxis.input2D, 975
 - examples.chemotaxis.parameters, 977
 - examples.chemotaxis.test, 977
 - examples.convection, 977
 - examples.convection.advection, 978
 - examples.convection.advection.explicitUpwind, 978
 - examples.convection.advection.implicitUpwind, 978
 - examples.convection.advection.vanLeerUpwind, 979
 - examples.convection.exponential1D, 979
 - examples.convection.exponential1D.cylindricalMesh1D, 980
 - examples.convection.exponential1D.cylindricalMesh1DNonUniform, 981
 - examples.convection.exponential1D.mesh1D, 983
 - examples.convection.exponential1D.tri2D, 984
 - examples.convection.exponential1DBack, 985
 - examples.convection.exponential1DBack.mesh1D, 985
 - examples.convection.exponential1DSource, 986
 - examples.convection.exponential1DSource.mesh1D, 987
 - examples.convection.exponential1DSource.tri2D, 988
 - examples.convection.exponential2D, 989
 - examples.convection.exponential2D.cylindricalMesh2D, 989
 - examples.convection.exponential2D.cylindricalMesh2DNonUniform, 991
 - examples.convection.exponential2D.mesh2D, 992
 - examples.convection.exponential2D.tri2D, 993
 - examples.convection.peclet, 994
 - examples.convection.powerLaw1D, 996
 - examples.convection.powerLaw1D.mesh1D, 996
 - examples.convection.powerLaw1D.tri2D, 997
 - examples.convection.robin, 998
 - examples.convection.source, 1000
 - examples.convection.test, 1001
 - examples.diffusion, 1001
 - examples.diffusion.anisotropy, 1001
 - examples.diffusion.circle, 1003
 - examples.diffusion.circleQuad, 1008

examples.diffusion.coupled, 1013
 examples.diffusion.electrostatics, 1015
 examples.diffusion.explicit, 1019
 examples.diffusion.explicit.mesh1D, 1020
 examples.diffusion.explicit.mixedelement, 1021
 examples.diffusion.explicit.test, 1022
 examples.diffusion.explicit.tri2D, 1022
 examples.diffusion.mesh1D, 1023
 examples.diffusion.mesh20x20, 1043
 examples.diffusion.mesh20x20Coupled, 1046
 examples.diffusion.nthOrder, 1049
 examples.diffusion.nthOrder.input4thOrder1D, 1050
 examples.diffusion.nthOrder.input4thOrder_line, 1052
 examples.diffusion.nthOrder.test, 1052
 examples.diffusion.steadyState, 1052
 examples.diffusion.steadyState.mesh1D, 1052
 examples.diffusion.steadyState.mesh1D.inputPeriodic, 1053
 examples.diffusion.steadyState.mesh1D.tri2Dinput, 1054
 examples.diffusion.steadyState.mesh20x20, 1054
 examples.diffusion.steadyState.mesh20x20.gmshinput, 1055
 examples.diffusion.steadyState.mesh20x20.isotropy, 1055
 examples.diffusion.steadyState.mesh20x20.modifiedMeshInput, 1055
 examples.diffusion.steadyState.mesh20x20.orthoerror, 1057
 examples.diffusion.steadyState.mesh20x20.tri2Dinput, 1057
 examples.diffusion.steadyState.mesh50x50, 1057
 examples.diffusion.steadyState.mesh50x50.input, 1057
 examples.diffusion.steadyState.mesh50x50.tri2Dinput, 1058
 examples.diffusion.steadyState.otherMeshes, 1058
 examples.diffusion.steadyState.otherMeshes.cubicalProblem, 1058
 examples.diffusion.steadyState.otherMeshes.grid3Dinput, 1059
 examples.diffusion.steadyState.otherMeshes.prism, 1059
 examples.diffusion.steadyState.test, 1060
 examples.diffusion.test, 1060
 examples.diffusion.variable, 1060
 examples.elphf, 1060
 examples.elphf.diffusion, 1062
 examples.elphf.diffusion.mesh1D, 1062
 examples.elphf.diffusion.mesh1Ddimensional, 1064
 examples.elphf.diffusion.mesh2D, 1067
 examples.elphf.input, 1070
 examples.elphf.phase, 1076
 examples.elphf.phaseDiffusion, 1079
 examples.elphf.poisson, 1087
 examples.elphf.test, 1090
 examples.flow, 1090
 examples.flow.stokesCavity, 1091
 examples.flow.test, 1096
 examples.levelSet, 1096
 examples.levelSet.advection, 1096
 examples.levelSet.advection.circle, 1096
 examples.levelSet.advection.mesh1D, 1098
 examples.levelSet.advection.test, 1099
 examples.levelSet.advection.trench, 1099
 examples.levelSet.distanceFunction, 1101
 examples.levelSet.distanceFunction.circle, 1101
 examples.levelSet.distanceFunction.interior, 1103
 examples.levelSet.distanceFunction.mesh1D, 1103
 examples.levelSet.distanceFunction.square, 1104
 examples.levelSet.distanceFunction.test, 1105
 examples.levelSet.electroChem, 1105
 examples.levelSet.electroChem.adsorbingSurfactantEquation, 1106
 examples.levelSet.electroChem.adsorption, 1106
 examples.levelSet.electroChem.gapFillDistanceVariable, 1107
 examples.levelSet.electroChem.gapFillMesh, 1107
 examples.levelSet.electroChem.gold, 1107
 examples.levelSet.electroChem.howToWriteAScript, 1108
 examples.levelSet.electroChem.leveler, 1114
 examples.levelSet.electroChem.lines, 1118
 examples.levelSet.electroChem.matplotlibSurfactantViewer, 1118
 examples.levelSet.electroChem.mayaviSurfactantViewer, 1122
 examples.levelSet.electroChem.metalIonDiffusionEquation, 1125
 examples.levelSet.electroChem.simpleTrenchSystem, 1125
 examples.levelSet.electroChem.surfactantBulkDiffusionEquation, 1128
 examples.levelSet.electroChem.test, 1128
 examples.levelSet.electroChem.trenchMesh, 1128
 examples.levelSet.surfactant, 1128
 examples.levelSet.surfactant.circle, 1129
 examples.levelSet.surfactant.expandingCircle, 1129
 examples.levelSet.surfactant.square, 1130
 examples.levelSet.surfactant.test, 1131
 examples.levelSet.test, 1131
 examples.meshing, 1131
 examples.meshing.gmshRefinement, 1131
 examples.meshing.inputGrid2D, 1131
 examples.meshing.sphere, 1132
 examples.meshing.test, 1133
 examples.parallel, 1133
 examples.phase, 1133
 examples.phase.anisotropy, 1134
 examples.phase.anisotropyOLD, 1138
 examples.phase.binary, 1141
 examples.phase.binaryCoupled, 1155
 examples.phase.impingement, 1169
 examples.phase.impingement.mesh20x20, 1170
 examples.phase.impingement.mesh40x1, 1174
 examples.phase.impingement.test, 1177
 examples.phase.missOrientation, 1177
 examples.phase.missOrientation.circle, 1177
 examples.phase.missOrientation.mesh1D, 1178
 examples.phase.missOrientation.modCircle, 1178
 examples.phase.missOrientation.test, 1179
 examples.phase.phaseViewer, 1179
 examples.phase.polyxtal, 1182
 examples.phase.polyxtalCoupled, 1188
 examples.phase.quaternary, 1194
 examples.phase.simple, 1200
 examples.phase.symmetry, 1208
 examples.phase.test, 1210
 examples.reactiveWetting, 1210
 examples.reactiveWetting.liquidVapor1D, 1210
 examples.reactiveWetting.liquidVapor2D, 1216
 examples.reactiveWetting.test, 1220
 examples.riemann, 1220
 examples.riemann.acoustics, 1220
 examples.riemann.test, 1220

- examples.test, 1220
- examples.updating, 1220
- examples.updating.update0_1to1_0, 1221
- examples.updating.update1_0to2_0, 1225
- examples.updating.update2_0to3_0, 1229
- fiPy, 139
- fiPy.boundaryConditions, 140
- fiPy.boundaryConditions.boundaryCondition, 141
- fiPy.boundaryConditions.constraint, 141
- fiPy.boundaryConditions.fixedFlux, 142
- fiPy.boundaryConditions.fixedValue, 142
- fiPy.boundaryConditions.nthOrderBoundaryCondition, 143
- fiPy.boundaryConditions.test, 143
- fiPy.matrices, 144
- fiPy.matrices.offsetSparseMatrix, 144
- fiPy.matrices.petscMatrix, 144
- fiPy.matrices.scipyMatrix, 144
- fiPy.matrices.sparseMatrix, 144
- fiPy.matrices.test, 144
- fiPy.matrices.trilinosMatrix, 144
- fiPy.meshes, 144
- fiPy.meshes.abstractMesh, 146
- fiPy.meshes.builders, 153
- fiPy.meshes.builders.abstractGridBuilder, 154
- fiPy.meshes.builders.grid1DBuilder, 154
- fiPy.meshes.builders.grid2DBuilder, 154
- fiPy.meshes.builders.grid3DBuilder, 154
- fiPy.meshes.builders.periodicGrid1DBuilder, 154
- fiPy.meshes.builders.utilityClasses, 154
- fiPy.meshes.cylindricalGrid1D, 154
- fiPy.meshes.cylindricalGrid2D, 154
- fiPy.meshes.cylindricalNonUniformGrid1D, 154
- fiPy.meshes.cylindricalNonUniformGrid2D, 163
- fiPy.meshes.cylindricalUniformGrid1D, 172
- fiPy.meshes.cylindricalUniformGrid2D, 179
- fiPy.meshes.factoryMeshes, 186
- fiPy.meshes.gmshMesh, 189
- fiPy.meshes.grid1D, 237
- fiPy.meshes.grid2D, 237
- fiPy.meshes.grid3D, 237
- fiPy.meshes.mesh, 237
- fiPy.meshes.mesh1D, 246
- fiPy.meshes.mesh2D, 254
- fiPy.meshes.nonUniformGrid1D, 263
- fiPy.meshes.nonUniformGrid2D, 271
- fiPy.meshes.nonUniformGrid3D, 280
- fiPy.meshes.periodicGrid1D, 288
- fiPy.meshes.periodicGrid2D, 297
- fiPy.meshes.periodicGrid3D, 323
- fiPy.meshes.representations, 379
- fiPy.meshes.representations.abstractRepresentation, 379
- fiPy.meshes.representations.gridRepresentation, 379
- fiPy.meshes.representations.meshRepresentation, 379
- fiPy.meshes.skewedGrid2D, 379
- fiPy.meshes.sphericalNonUniformGrid1D, 388
- fiPy.meshes.sphericalUniformGrid1D, 397
- fiPy.meshes.test, 404
- fiPy.meshes.topologies, 404
- fiPy.meshes.topologies.abstractTopology, 404
- fiPy.meshes.topologies.gridTopology, 404
- fiPy.meshes.topologies.meshTopology, 404
- fiPy.meshes.tri2D, 404
- fiPy.meshes.uniformGrid, 413
- fiPy.meshes.uniformGrid1D, 420
- fiPy.meshes.uniformGrid2D, 427
- fiPy.meshes.uniformGrid3D, 434

- fiPy.numerix, 1222
- fiPy.solvers, 441
- fiPy.solvers.convergence, 443
- fiPy.solvers.petsc, 448
- fiPy.solvers.petsc.comms, 449
- fiPy.solvers.petsc.comms.parallelPETScCommWrapper, 449
- fiPy.solvers.petsc.comms.petscCommWrapper, 450
- fiPy.solvers.petsc.comms.serialPETScCommWrapper, 450
- fiPy.solvers.petsc.dummySolver, 451
- fiPy.solvers.petsc.linearBicgSolver, 451
- fiPy.solvers.petsc.linearCGSolver, 453
- fiPy.solvers.petsc.linearCGSSolver, 452
- fiPy.solvers.petsc.linearGMRESSolver, 454
- fiPy.solvers.petsc.linearLUSolver, 455
- fiPy.solvers.petsc.petscConvergence, 456
- fiPy.solvers.petsc.petscKrylovSolver, 461
- fiPy.solvers.petsc.petscSolver, 462
- fiPy.solvers.petsc.preconditioners, 463
- fiPy.solvers.petsc.preconditioners.defaultPreconditioner, 464
- fiPy.solvers.petsc.preconditioners.hyprePreconditioner, 465
- fiPy.solvers.petsc.preconditioners.icPreconditioner, 465
- fiPy.solvers.petsc.preconditioners.iluPreconditioner, 465
- fiPy.solvers.petsc.preconditioners.jacobiPreconditioner, 465
- fiPy.solvers.petsc.preconditioners.luPreconditioner, 466
- fiPy.solvers.petsc.preconditioners.petscPreconditioner, 466
- fiPy.solvers.petsc.preconditioners.ssorPreconditioner, 466
- fiPy.solvers.preconditioner, 466
- fiPy.solvers.pyAMG, 467
- fiPy.solvers.pyAMG.preconditioners, 467
- fiPy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner, 467
- fiPy.solvers.pyamgx, 468
- fiPy.solvers.pyamgx.aggregationAMGSolver, 468
- fiPy.solvers.pyamgx.classicalAMGSolver, 469
- fiPy.solvers.pyamgx.linearBiCGStabSolver, 470
- fiPy.solvers.pyamgx.linearCGSolver, 471
- fiPy.solvers.pyamgx.linearFGMRESSolver, 472
- fiPy.solvers.pyamgx.linearGMRESSolver, 473
- fiPy.solvers.pyamgx.linearLUSolver, 474
- fiPy.solvers.pyamgx.preconditioners, 475
- fiPy.solvers.pyamgx.pyamgxConvergence, 478
- fiPy.solvers.pyamgx.pyAMGXSolver, 477
- fiPy.solvers.pyamgx.smoothers, 478
- fiPy.solvers.scipy, 479
- fiPy.solvers.scipy.linearBicgstabSolver, 479
- fiPy.solvers.scipy.linearCGSolver, 481
- fiPy.solvers.scipy.linearCGSSolver, 480
- fiPy.solvers.scipy.linearGMRESSolver, 482
- fiPy.solvers.scipy.linearLUSolver, 483
- fiPy.solvers.scipy.preconditioners, 484
- fiPy.solvers.scipy.preconditioners.iluPreconditioner, 484
- fiPy.solvers.scipy.preconditioners.jacobiPreconditioner, 484
- fiPy.solvers.scipy.preconditioners.scipyPreconditioner, 485
- fiPy.solvers.scipy.scipyConvergence, 485
- fiPy.solvers.scipy.scipyKrylovSolver, 486

- fiPy.solvers.scipy.scipySolver, 487
- fiPy.solvers.solver, 488
- fiPy.solvers.test, 497
- fiPy.solvers.trilinos, 497
- fiPy.solvers.trilinos.aztecConvergence, 497
- fiPy.solvers.trilinos.comms, 499
- fiPy.solvers.trilinos.comms.epetraCommWrapper, 499
- fiPy.solvers.trilinos.comms.parallelEpetraCommWrapper, 500
- fiPy.solvers.trilinos.comms.serialEpetraCommWrapper, 500
- fiPy.solvers.trilinos.linearBicgstabSolver, 501
- fiPy.solvers.trilinos.linearCGSolver, 503
- fiPy.solvers.trilinos.linearCGSSolver, 502
- fiPy.solvers.trilinos.linearGMRESSolver, 504
- fiPy.solvers.trilinos.linearLUSolver, 505
- fiPy.solvers.trilinos.preconditioners, 505
- fiPy.solvers.trilinos.preconditioners.domDecompPreconditioner, 507
- fiPy.solvers.trilinos.preconditioners.icPreconditioner, 508
- fiPy.solvers.trilinos.preconditioners.iluPreconditioner, 508
- fiPy.solvers.trilinos.preconditioners.jacobiPreconditioner, 508
- fiPy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner, 508
- fiPy.solvers.trilinos.preconditioners.multilevelDDPreconditioner, 509
- fiPy.solvers.trilinos.preconditioners.multilevelNSSAPPreconditioner, 509
- fiPy.solvers.trilinos.preconditioners.multilevelPreconditioner, 509
- fiPy.solvers.trilinos.preconditioners.multilevelSAPreconditioner, 510
- fiPy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner, 510
- fiPy.solvers.trilinos.preconditioners.multilevelSolverSmoother, 510
- fiPy.solvers.trilinos.preconditioners.trilinosPreconditioner, 511
- fiPy.solvers.trilinos.trilinosAztec00Solver, 511
- fiPy.solvers.trilinos.trilinosMLTest, 512
- fiPy.solvers.trilinos.trilinosNonlinearSolver, 513
- fiPy.solvers.trilinos.trilinosSolver, 515
- fiPy.steps, 516
- fiPy.steps.pidStepper, 518
- fiPy.steps.pseudoRKQSStepper, 518
- fiPy.steps.stepper, 518
- fiPy.terms, 518
- fiPy.terms.abstractBinaryTerm, 525
- fiPy.terms.abstractConvectionTerm, 525
- fiPy.terms.abstractDiffusionTerm, 525
- fiPy.terms.abstractUpwindConvectionTerm, 525
- fiPy.terms.advectionTerm, 525
- fiPy.terms.asymmetricConvectionTerm, 532
- fiPy.terms.binaryTerm, 532
- fiPy.terms.cellTerm, 532
- fiPy.terms.centralDiffConvectionTerm, 536
- fiPy.terms.coupledBinaryTerm, 541
- fiPy.terms.diffusionTerm, 541
- fiPy.terms.diffusionTermCorrection, 545
- fiPy.terms.diffusionTermNoCorrection, 548
- fiPy.terms.explicitDiffusionTerm, 552
- fiPy.terms.explicitSourceTerm, 555
- fiPy.terms.explicitUpwindConvectionTerm, 555
- fiPy.terms.exponentialConvectionTerm, 560
- fiPy.terms.faceTerm, 566
- fiPy.terms.firstOrderAdvectionTerm, 570
- fiPy.terms.hybridConvectionTerm, 575
- fiPy.terms.implicitDiffusionTerm, 580
- fiPy.terms.implicitSourceTerm, 580
- fiPy.terms.nonDiffusionTerm, 584
- fiPy.terms.powerLawConvectionTerm, 584
- fiPy.terms.residualTerm, 590
- fiPy.terms.sourceTerm, 594
- fiPy.terms.term, 598
- fiPy.terms.test, 601
- fiPy.terms.transientTerm, 601
- fiPy.terms.unaryTerm, 606
- fiPy.terms.upwindConvectionTerm, 606
- fiPy.terms.vanLeerConvectionTerm, 611
- fiPy.testFiPy, 616
- fiPy.tests, 616
- fiPy.tests.doctestPlus, 617
- fiPy.tests.lateImportTest, 619
- fiPy.tests.test, 619
- fiPy.tests.testProgram, 619
- fiPy.tools, 620
- fiPy.tools.comms, 636
- fiPy.tools.comms.commWrapper, 637
- fiPy.tools.comms.dummyComm, 637
- fiPy.tools.copy_script, 637
- fiPy.tools.debug, 638
- fiPy.tools.decorators, 638
- fiPy.tools.dimensions, 638
- fiPy.tools.dimensions.DictWithDefault, 639
- fiPy.tools.dimensions.NumberDict, 639
- fiPy.tools.dimensions.physicalField, 639
- fiPy.tools.dump, 663, 1173
- fiPy.tools.inline, 664
- fiPy.tools.logging, 664
- fiPy.tools.logging.environment, 665
- fiPy.tools.numerix, 666
- fiPy.tools.parser, 671, 1109, 1170
- fiPy.tools.readSharedTempFile, 672
- fiPy.tools.test, 673
- fiPy.tools.timer, 673
- fiPy.tools.vector, 673
- fiPy.variables, 674
- fiPy.variables.addOverFacesVariable, 676
- fiPy.variables.arithmeticCellToFaceVariable, 676
- fiPy.variables.betaNoiseVariable, 676
- fiPy.variables.binaryOperatorVariable, 692
- fiPy.variables.cellToFaceVariable, 692
- fiPy.variables.cellVariable, 692
- fiPy.variables.constant, 707
- fiPy.variables.constraintMask, 707
- fiPy.variables.coupledCellVariable, 707
- fiPy.variables.distanceVariable, 707
- fiPy.variables.exponentialNoiseVariable, 725
- fiPy.variables.faceGradContributionsVariable, 741
- fiPy.variables.faceGradVariable, 741
- fiPy.variables.faceVariable, 741
- fiPy.variables.gammaNoiseVariable, 752
- fiPy.variables.gaussCellGradVariable, 768
- fiPy.variables.gaussianNoiseVariable, 768
- fiPy.variables.harmonicCellToFaceVariable, 784
- fiPy.variables.histogramVariable, 784
- fiPy.variables.interfaceAreaVariable, 798
- fiPy.variables.interfaceFlagVariable, 798
- fiPy.variables.leastSquaresCellGradVariable, 798
- fiPy.variables.levelSetDiffusionVariable, 798
- fiPy.variables.meshVariable, 798
- fiPy.variables.minmodCellToFaceVariable, 809
- fiPy.variables.modCellGradVariable, 809

- fipy.variables.modCellToFaceVariable, 809
 - fipy.variables.modFaceGradVariable, 809
 - fipy.variables.modPhysicalField, 809
 - fipy.variables.modularVariable, 809
 - fipy.variables.noiseVariable, 824
 - fipy.variables.operatorVariable, 839
 - fipy.variables.scharfetterGummelFaceVariable, 839
 - fipy.variables.surfactantConvectionVariable, 850
 - fipy.variables.surfactantVariable, 861
 - fipy.variables.test, 877
 - fipy.variables.unaryOperatorVariable, 877
 - fipy.variables.uniformNoiseVariable, 877
 - fipy.variables.variable, 892
 - fipy.viewers, 901, 972, 981, 982, 984, 988, 991, 992, 1024, 1044, 1047, 1051, 1093, 1140, 1147, 1161, 1172, 1176, 1201
 - fipy.viewers.matplotlibViewer, 905
 - fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer, 908
 - fipy.viewers.matplotlibViewer.abstractMatplotlibViewer, 910
 - fipy.viewers.matplotlibViewer.matplotlib1DViewer, 913
 - fipy.viewers.matplotlibViewer.matplotlib2DContourViewer, 915
 - fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer, 918
 - fipy.viewers.matplotlibViewer.matplotlib2DGridViewer, 921
 - fipy.viewers.matplotlibViewer.matplotlib2DViewer, 924
 - fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer, 926
 - fipy.viewers.matplotlibViewer.matplotlibStreamViewer, 927
 - fipy.viewers.matplotlibViewer.matplotlibVectorViewer, 931
 - fipy.viewers.matplotlibViewer.test, 934
 - fipy.viewers.mayaviViewer, 934
 - fipy.viewers.mayaviViewer.mayaviClient, 938
 - fipy.viewers.mayaviViewer.mayaviDaemon, 941
 - fipy.viewers.mayaviViewer.test, 942
 - fipy.viewers.multiViewer, 942
 - fipy.viewers.test, 943
 - fipy.viewers.testinteractive, 944
 - fipy.viewers.tsvViewer, 944
 - fipy.viewers.viewer, 946
 - fipy.viewers.vtkViewer, 948
 - fipy.viewers.vtkViewer.test, 952
 - fipy.viewers.vtkViewer.vtkCellViewer, 952
 - fipy.viewers.vtkViewer.vtkFaceViewer, 954
 - fipy.viewers.vtkViewer.vtkViewer, 956
 - gmsh, 1107, 1115, 1126
 - package, 135
 - package.subpackage, 135
 - package.subpackage.base, 136
 - package.subpackage.object, 137
 - scipy, 1006, 1011, 1161, 1207
 - viewers, 1198
- MPI, 122
- mpi4py, 122
- MSHFile (class in fipy.meshes.gmshMesh), 235
- MultilevelDDMLPreconditioner (class in fipy.solvers.trilinos.preconditioners), 506
- MultilevelDDMLPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner), 508
- MultilevelDDPreconditioner (class in fipy.solvers.trilinos.preconditioners), 506
- MultilevelDDPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner), 509
- MultilevelNSSAPreconditioner (class in fipy.solvers.trilinos.preconditioners), 506
- MultilevelNSSAPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner), 509
- MultilevelPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelPreconditioner), 509
- MultilevelSAPreconditioner (class in fipy.solvers.trilinos.preconditioners), 506
- MultilevelSAPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner), 510
- MultilevelSGSPreconditioner (class in fipy.solvers.trilinos.preconditioners), 506
- MultilevelSGSPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner), 510
- MultilevelSolverSmootherPreconditioner (class in fipy.solvers.trilinos.preconditioners), 506
- MultilevelSolverSmootherPreconditioner (class in fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner), 510
- multiply() (fipy.tools.dimensions.physicalField.PhysicalField method), 653
- multiply() (fipy.tools.PhysicalField method), 631
- MultiViewer (class in fipy.viewers.multiViewer), 942
- N**
- name() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 663
- nearest() (in module fipy.tools.numerix), 669
- nix_info() (in module fipy.tools.logging.environment), 665
- NoiseVariable (class in fipy.variables.noiseVariable), 824
- NonUniformGrid1D (class in fipy.meshes.nonUniformGrid1D), 263
- NonUniformGrid2D (class in fipy.meshes.nonUniformGrid2D), 271
- NonUniformGrid3D (class in fipy.meshes.nonUniformGrid3D), 280
- NormalConvergence, 492
- NthOrderBoundaryCondition, 971, 1050
- NthOrderBoundaryCondition (class in fipy.boundaryConditions.nthOrderBoundaryCondition), 143
- NullDivergence (class in fipy.solvers.convergence), 447
- numarray, 123
- Numeric, 123
- numericValue (fipy.tools.dimensions.physicalField.PhysicalField property), 653
- numericValue (fipy.tools.PhysicalField property), 632
- NumPy, 123
- O**
- object
 - fipy.terms.implicitDiffusionTerm.DiffusionTerm, 1004, 1009
 - fipy.terms.transientTerm.TransientTerm, 1004, 1009
 - fipy.variables.cellVariable.CellVariable, 1003, 1008
 - fipy.viewers.tsvViewer.TSVViewer, 1005, 1010
- Object (class in package.subpackage.object), 137
- OffsetSparseMatrix() (in module fipy.matrices.offsetSparseMatrix), 144
- old (fipy.variables.betaNoiseVariable.BetaNoiseVariable property), 689
- old (fipy.variables.cellVariable.CellVariable property), 704
- old (fipy.variables.distanceVariable.DistanceVariable property), 722

old (*fiPy*.variables.exponentialNoiseVariable.ExponentialNoiseVariable property), 738
old (*fiPy*.variables.gammaNoiseVariable.GammaNoiseVariable property), 765
old (*fiPy*.variables.gaussianNoiseVariable.GaussianNoiseVariable property), 781
old (*fiPy*.variables.histogramVariable.HistogramVariable property), 795
old (*fiPy*.variables.modularVariable.ModularVariable property), 821
old (*fiPy*.variables.noiseVariable.NoiseVariable property), 836
old (*fiPy*.variables.surfactantVariable.SurfactantVariable property), 874
old (*fiPy*.variables.uniformNoiseVariable.UniformNoiseVariable property), 889
OpenMP, 123
openMShFile() (in module *fiPy*.meshes.gmshMesh), 236
openPOsFile() (in module *fiPy*.meshes.gmshMesh), 237
OutOfRangeDivergence (class in *fiPy*.solvers.convergence), 447

P

package
 module, 135
package.subpackage
 module, 135
package.subpackage.base
 module, 136
package.subpackage.object
 module, 137
package_info() (in module *fiPy*.tools.logging.environment), 665
pandas, 123
parallelComm (in module *fiPy*.tools), 620
ParallelEpetraCommWrapper (class in *fiPy*.solvers.trilinos.comms.parallelEpetraCommWrapper), 500
ParallelPETScCommWrapper (class in *fiPy*.solvers.petsc.comms.parallelPETScCommWrapper), 449
ParameterWarning, 492
parse() (in module *fiPy*.tools.parser), 671
parse_command_line() (*fiPy*.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon method), 941
PCG, 123
PeriodicGrid1D (class in *fiPy*.meshes.periodicGrid1D), 288
PeriodicGrid2D (class in *fiPy*.meshes.periodicGrid2D), 297
PeriodicGrid2DLeftRight (class in *fiPy*.meshes.periodicGrid2D), 306
PeriodicGrid2DTopBottom (class in *fiPy*.meshes.periodicGrid2D), 315
PeriodicGrid3D (class in *fiPy*.meshes.periodicGrid3D), 323
PeriodicGrid3DFrontBack (class in *fiPy*.meshes.periodicGrid3D), 332
PeriodicGrid3DLeftRight (class in *fiPy*.meshes.periodicGrid3D), 340
PeriodicGrid3DLeftRightFrontBack (class in *fiPy*.meshes.periodicGrid3D), 348
PeriodicGrid3DLeftRightTopBottom (class in *fiPy*.meshes.periodicGrid3D), 356
PeriodicGrid3DTopBottom (class in *fiPy*.meshes.periodicGrid3D), 363
PeriodicGrid3DTopBottomFrontBack (class in *fiPy*.meshes.periodicGrid3D), 371
petsc4py, 123
PETScCommWrapper (class in *fiPy*.solvers.petsc.comms.petscCommWrapper), 450
PETScKrylovSolver (class in *fiPy*.solvers.petsc.petscKrylovSolver), 461

PETScPreconditioner (class in *fiPy*.solvers.petsc.preconditioners.petscPreconditioner), 466
PETScSolver (class in *fiPy*.solvers.petsc.petscSolver), 462
PhaseViewer (class in *examples.phase.phaseViewer*), 1179
PhysicalField (class in *fiPy*.tools), 620
PhysicalField (class in *fiPy*.tools.dimensions.physicalField), 641
physicalShape (*fiPy*.meshes.skewedGrid2D.SkewedGrid2D property), 387
physicalShape (*fiPy*.meshes.tri2D.Tri2D property), 412
PhysicalUnit (class in *fiPy*.tools.dimensions.physicalField), 656
pi, 1135, 1140, 1171, 1176
PIDStepper() (in module *fiPy*.steppers.pidStepper), 518
pip, 123
pip_info() (in module *fiPy*.tools.logging.environment), 665
platform_info() (in module *fiPy*.tools.logging.environment), 665
plot() (exam-
 ples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer method), 1121
plot() (*examples.phase.phaseViewer.PhaseViewer* method), 1180
plot() (*fiPy*.viewers.DummyViewer method), 902
plot() (*fiPy*.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer method), 909
plot() (*fiPy*.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer method), 911
plot() (*fiPy*.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer method), 914
plot() (*fiPy*.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer method), 917
plot() (*fiPy*.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer method), 920
plot() (*fiPy*.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer method), 922
plot() (*fiPy*.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer method), 925
plot() (*fiPy*.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer method), 930
plot() (*fiPy*.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer method), 933
plot() (*fiPy*.viewers.mayaviViewer.MayaviClient method), 936
plot() (*fiPy*.viewers.mayaviViewer.mayaviClient.MayaviClient method), 940
plot() (*fiPy*.viewers.multiViewer.MultiViewer method), 942
plot() (*fiPy*.viewers.tsvViewer.TSVViewer method), 945
plot() (*fiPy*.viewers.viewer.AbstractViewer method), 947
plot() (*fiPy*.viewers.vtkViewer.VTKCellViewer method), 948
plot() (*fiPy*.viewers.vtkViewer.vtkCellViewer.VTKCellViewer method), 953
plot() (*fiPy*.viewers.vtkViewer.VTKFaceViewer method), 950
plot() (*fiPy*.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer method), 954
plot() (*fiPy*.viewers.vtkViewer.vtkViewer.VTKViewer method), 956
plotMesh() (exam-
 ples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer method), 1121
plotMesh() (exam-
 ples.levelSet.electroChem.mayaviSurfactantViewer.MayaviSurfactantViewer method), 1124

- `plotMesh()` (*examples.phase.phaseViewer.PhaseViewer method*), 1180
- `plotMesh()` (*fiPy.viewers.DummyViewer method*), 902
- `plotMesh()` (*fiPy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer method*), 909
- `plotMesh()` (*fiPy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer method*), 911
- `plotMesh()` (*fiPy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer method*), 914
- `plotMesh()` (*fiPy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer method*), 917
- `plotMesh()` (*fiPy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer method*), 920
- `plotMesh()` (*fiPy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer method*), 922
- `plotMesh()` (*fiPy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer method*), 925
- `plotMesh()` (*fiPy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer method*), 930
- `plotMesh()` (*fiPy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer method*), 933
- `plotMesh()` (*fiPy.viewers.mayaviViewer.MayaviClient method*), 936
- `plotMesh()` (*fiPy.viewers.mayaviViewer.mayaviClient.MayaviClient method*), 940
- `plotMesh()` (*fiPy.viewers.multiViewer.MultiViewer method*), 942
- `plotMesh()` (*fiPy.viewers.tsvViewer.TSVViewer method*), 945
- `plotMesh()` (*fiPy.viewers.viewer.AbstractViewer method*), 947
- `plotMesh()` (*fiPy.viewers.vtkViewer.VTKCellViewer method*), 949
- `plotMesh()` (*fiPy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer method*), 953
- `plotMesh()` (*fiPy.viewers.vtkViewer.VTKFaceViewer method*), 950
- `plotMesh()` (*fiPy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer method*), 954
- `plotMesh()` (*fiPy.viewers.vtkViewer.vtkViewer.VTKViewer method*), 956
- `PolynomialPreconditioner` (class in *fiPy.solvers.pyamgx.preconditioners*), 476
- `POSFile` (class in *fiPy.meshes.gmshMesh*), 236
- `PowerLawConvectionTerm`, 1146, 1197
- `PowerLawConvectionTerm` (class in *fiPy.terms.powerLawConvectionTerm*), 584
- `Preconditioner` (class in *fiPy.solvers.preconditioner*), 467
- `PreconditionerNotPositiveDefiniteWarning`, 493
- `PreconditionerWarning`, 494
- `PreconditioningDivergence` (class in *fiPy.solvers.convergence*), 447
- `PRINT()` (in module *fiPy.tools.debug*), 638
- `prune()` (in module *fiPy.tools.vector*), 674
- `PseudoRKQSS stepper` (in module *fiPy.steps.pseudoRKQSSStepper*), 518
- `put()` (*fiPy.tools.dimensions.physicalField.PhysicalField method*), 653
- `put()` (*fiPy.tools.PhysicalField method*), 632
- `put()` (in module *fiPy.tools.numerix*), 669
- `putAdd()` (in module *fiPy.tools.vector*), 674
- `pyamgx_BreakdownDivergence` (class in *fiPy.solvers.pyamgx.pyamgxConvergence*), 478
- `pyamgx_Convergence` (class in *fiPy.solvers.pyamgx.pyamgxConvergence*), 478
- `pyamgx_IterationDivergence` (class in *fiPy.solvers.pyamgx.pyamgxConvergence*), 478
- `PyAMGXPreconditioner` (class in *fiPy.solvers.pyamgx.preconditioners*), 476
- `PyAMGXSolver` (class in *fiPy.solvers.pyamgx.pyAMGXSolver*), 477
- `PyPI`, 123
- `Pyrex`, 123
- `PySphinx`, 123
- `Python`, 123
- `Python 3`, 123
- `Python Enhancement Proposals`, 123
- `PEP 3000`, 123
- `PyTrilinos`, 123
- `PyVTK`, 123
- R**
- `rank()` (in module *fiPy.tools.numerix*), 670
- `rdot()` (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable method*), 689
- `rdot()` (*fiPy.variables.cellVariable.CellVariable method*), 704
- `rdot()` (*fiPy.variables.distanceVariable.DistanceVariable method*), 723
- `rdot()` (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 738
- `rdot()` (*fiPy.variables.faceVariable.FaceVariable method*), 749
- `rdot()` (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 765
- `rdot()` (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 781
- `rdot()` (*fiPy.variables.histogramVariable.HistogramVariable method*), 796
- `rdot()` (*fiPy.variables.meshVariable.MeshVariable method*), 807
- `rdot()` (*fiPy.variables.modularVariable.ModularVariable method*), 822
- `rdot()` (*fiPy.variables.noiseVariable.NoiseVariable method*), 836
- `rdot()` (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 847
- `rdot()` (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 859
- `rdot()` (*fiPy.variables.surfactantVariable.SurfactantVariable method*), 874
- `rdot()` (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 889
- `read()` (*fiPy.meshes.gmshMesh.MSHFile method*), 235
- `read()` (in module *fiPy.tools.dump*), 663
- `register_skipper()` (in module *fiPy.tests.doctestPlus*), 617
- `RelativeToleranceConvergence` (class in *fiPy.solvers.convergence*), 448
- `release()` (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable method*), 689
- `release()` (*fiPy.variables.cellVariable.CellVariable method*), 704
- `release()` (*fiPy.variables.distanceVariable.DistanceVariable method*), 723
- `release()` (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 738
- `release()` (*fiPy.variables.faceVariable.FaceVariable method*), 750
- `release()` (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 765
- `release()` (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 781
- `release()` (*fiPy.variables.histogramVariable.HistogramVariable method*), 796
- `release()` (*fiPy.variables.meshVariable.MeshVariable method*), 807

- [release\(\)](#) (*fipy.variables.modularVariable.ModularVariable* method), 822
[release\(\)](#) (*fipy.variables.noiseVariable.NoiseVariable* method), 836
[release\(\)](#) (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 848
[release\(\)](#) (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 859
[release\(\)](#) (*fipy.variables.surfactantVariable.SurfactantVariable* method), 874
[release\(\)](#) (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 889
[release\(\)](#) (*fipy.variables.variable.Variable* method), 900
[report_skips\(\)](#) (in module *fipy.tests.doctestPlus*), 617
[reshape\(\)](#) (*fipy.tools.dimensions.physicalField.PhysicalField* method), 654
[reshape\(\)](#) (*fipy.tools.physicalField.PhysicalField* method), 632
[reshape\(\)](#) (in module *fipy.tools.numerix*), 670
[residual](#) (*fipy.solvers.convergence.ConvergenceBase* attribute), 445
[residual\(\)](#) (in module *fipy.steps*), 517
[ResidualTerm](#) (class in *fipy.terms.residualTerm*), 590
[residualVectorAndNorm\(\)](#) (*fipy.terms.advectionTerm.AdvectionTerm* method), 530
[residualVectorAndNorm\(\)](#) (*fipy.terms.cellTerm.CellTerm* method), 535
[residualVectorAndNorm\(\)](#) (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 540
[residualVectorAndNorm\(\)](#) (*fipy.terms.diffusionTerm.DiffusionTerm* method), 544
[residualVectorAndNorm\(\)](#) (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* method), 547
[residualVectorAndNorm\(\)](#) (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* method), 550
[residualVectorAndNorm\(\)](#) (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* method), 554
[residualVectorAndNorm\(\)](#) (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 559
[residualVectorAndNorm\(\)](#) (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 564
[residualVectorAndNorm\(\)](#) (*fipy.terms.faceTerm.FaceTerm* method), 569
[residualVectorAndNorm\(\)](#) (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 574
[residualVectorAndNorm\(\)](#) (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 579
[residualVectorAndNorm\(\)](#) (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 583
[residualVectorAndNorm\(\)](#) (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 588
[residualVectorAndNorm\(\)](#) (*fipy.terms.residualTerm.ResidualTerm* method), 592
[residualVectorAndNorm\(\)](#) (*fipy.terms.sourceTerm.SourceTerm* method), 597
[residualVectorAndNorm\(\)](#) (*fipy.terms.term.Term* method), 600
[residualVectorAndNorm\(\)](#) (*fipy.terms.transientTerm.TransientTerm* method), 605
[residualVectorAndNorm\(\)](#) (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 610
[residualVectorAndNorm\(\)](#) (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 615
[RHSvector](#), 1094
[RHSvector](#) (*fipy.terms.advectionTerm.AdvectionTerm* property), 528
[RHSvector](#) (*fipy.terms.cellTerm.CellTerm* property), 532
[RHSvector](#) (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* property), 537
[RHSvector](#) (*fipy.terms.diffusionTerm.DiffusionTerm* property), 542
[RHSvector](#) (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* property), 545
[RHSvector](#) (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* property), 548
[RHSvector](#) (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* property), 552
[RHSvector](#) (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* property), 557
[RHSvector](#) (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* property), 562
[RHSvector](#) (*fipy.terms.faceTerm.FaceTerm* property), 566
[RHSvector](#) (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* property), 571
[RHSvector](#) (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* property), 576
[RHSvector](#) (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* property), 581
[RHSvector](#) (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* property), 586
[RHSvector](#) (*fipy.terms.residualTerm.ResidualTerm* property), 590
[RHSvector](#) (*fipy.terms.sourceTerm.SourceTerm* property), 594
[RHSvector](#) (*fipy.terms.term.Term* property), 598
[RHSvector](#) (*fipy.terms.transientTerm.TransientTerm* property), 602
[RHSvector](#) (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* property), 608
[RHSvector](#) (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* property), 612
[RHSZeroConvergence](#) (class in *fipy.solvers.convergence*), 447
[runGold](#), 1107
[runLeveler](#), 1114
[runSimpleTrenchSystem](#), 1125
- ## S
- [ScalarQuantityOutOfRangeWarning](#), 494
[ScharfetterGummelFaceVariable](#) (class in *fipy.variables.scharfetterGummelFaceVariable*), 839
[ScientificPython](#), 123
[SciPy](#), 123, 1147
[scipy](#) module, 1006, 1011, 1161, 1207
[SciPy_BreakdownDivergence](#) (class in *fipy.solvers.scipy.scipyConvergence*), 485
[SciPy_Convergence](#) (class in *fipy.solvers.scipy.scipyConvergence*), 485
[SciPy_IterationDivergence](#) (class in *fipy.solvers.scipy.scipyConvergence*), 485

ScipyKrylovSolver (class in *fipy.solvers.scipy.scipyKrylovSolver*), 486

ScipyPreconditioner (class in *fipy.solvers.scipy.preconditioners.scipyPreconditioner*), 485

ScipySolver (class in *fipy.solvers.scipy.scipySolver*), 487

scramble() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 690

scramble() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 739

scramble() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 765

scramble() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 782

scramble() (*fipy.variables.noiseVariable.NoiseVariable* method), 837

scramble() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 890

serialComm (in module *fipy.tools*), 620

SerialEpetraCommWrapper (class in *fipy.solvers.trilinos.comms.serialEpetraCommWrapper*), 500

SerialPETScCommWrapper (class in *fipy.solvers.petsc.comms.serialPETScCommWrapper*), 450

SerialSolverError, 442

setLimits() (examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer method), 1121

setLimits() (examples.levelSet.electroChem.mayaviSurfactantViewer.MayaviSurfactantViewer method), 1124

setLimits() (examples.phase.phaseViewer.PhaseViewer method), 1181

setLimits() (*fipy.viewers.DummyViewer* method), 902

setLimits() (*fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* method), 909

setLimits() (*fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* method), 912

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* method), 914

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* method), 917

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* method), 920

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* method), 923

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* method), 925

setLimits() (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* method), 930

setLimits() (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* method), 933

setLimits() (*fipy.viewers.mayaviViewer.MayaviClient* method), 937

setLimits() (*fipy.viewers.mayaviViewer.mayaviClient.MayaviClient* method), 940

setLimits() (*fipy.viewers.multiViewer.MultiViewer* method), 942

setLimits() (*fipy.viewers.tsvViewer.TSVViewer* method), 945

setLimits() (*fipy.viewers.viewer.AbstractViewer* method), 947

setLimits() (*fipy.viewers.vtkViewer.VTKCellViewer* method), 949

setLimits() (*fipy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer* method), 953

setLimits() (*fipy.viewers.vtkViewer.VTKFaceViewer* method), 950

setLimits() (*fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer* method), 955

setLimits() (*fipy.viewers.vtkViewer.vtkViewer.VTKViewer* method), 956

setName() (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 663

setup_source() (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 942

setValue() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 690

setValue() (*fipy.variables.cellVariable.CellVariable* method), 705

setValue() (*fipy.variables.distanceVariable.DistanceVariable* method), 723

setValue() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 739

setValue() (*fipy.variables.faceVariable.FaceVariable* method), 750

setValue() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 766

setValue() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 782

setValue() (*fipy.variables.histogramVariable.HistogramVariable* method), 796

setValue() (*fipy.variables.meshVariable.MeshVariable* method), 807

setValue() (*fipy.variables.modularVariable.ModularVariable* method), 822

setValue() (*fipy.variables.noiseVariable.NoiseVariable* method), 837

setValue() (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 848

setValue() (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 860

setValue() (*fipy.variables.surfactantVariable.SurfactantVariable* method), 875

setValue() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 890

setValue() (*fipy.variables.variable.Variable* method), 900

shape (*fipy.tools.dimensions.physicalField.PhysicalField* property), 654

shape (*fipy.tools.dimensions.physicalField.PhysicalField* property), 632

shape (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 690

shape (*fipy.variables.cellVariable.CellVariable* property), 705

shape (*fipy.variables.distanceVariable.DistanceVariable* property), 724

shape (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 739

shape (*fipy.variables.faceVariable.FaceVariable* property), 750

shape (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 766

shape (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 782

shape (*fipy.variables.histogramVariable.HistogramVariable* property), 797

shape (*fipy.variables.meshVariable.MeshVariable* property), 808

shape (*fipy.variables.modularVariable.ModularVariable* property), 823

- shape (*fipy.variables.noiseVariable.NoiseVariable* property), 837
- shape (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 848
- shape (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 860
- shape (*fipy.variables.surfactantVariable.SurfactantVariable* property), 875
- shape (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 890
- shape (*fipy.variables.variable.Variable* property), 900
- SharedTemporaryFile() (in module *fipy.tools*), 635
- SharedTemporaryFile() (in module *fipy.tools.sharedtempfile*), 672
- sign() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 654
- sign() (*fipy.tools.PhysicalField* method), 632
- sin() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 654
- sin() (*fipy.tools.PhysicalField* method), 633
- sinh() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 655
- sinh() (*fipy.tools.PhysicalField* method), 633
- SkewedGrid2D (class in *fipy.meshes.skewedGrid2D*), 379
- SmoothedAggregationPreconditioner (class in *fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner*), 467
- Smoother (class in *fipy.solvers.pyamgx.smoothers*), 478
- SolutionVariableNumberError, 521
- SolutionVariableRequiredError, 521
- solve, 1162
- solve() (*fipy.terms.advectionTerm.AdvectionTerm* method), 531
- solve() (*fipy.terms.cellTerm.CellTerm* method), 535
- solve() (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 540
- solve() (*fipy.terms.diffusionTerm.DiffusionTerm* method), 544
- solve() (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* method), 547
- solve() (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* method), 551
- solve() (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* method), 554
- solve() (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 559
- solve() (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 564
- solve() (*fipy.terms.faceTerm.FaceTerm* method), 569
- solve() (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 574
- solve() (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 579
- solve() (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 583
- solve() (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 589
- solve() (*fipy.terms.residualTerm.ResidualTerm* method), 593
- solve() (*fipy.terms.sourceTerm.SourceTerm* method), 597
- solve() (*fipy.terms.term.Term* method), 600
- solve() (*fipy.terms.transientTerm.TransientTerm* method), 605
- solve() (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 610
- solve() (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 615
- Solver (class in *fipy.solvers.solver*), 495
- solver (*fipy.solvers.convergence.ConvergenceBase* attribute), 444
- SolverConvergenceWarning, 495
- SolverModifyingPreconditioner (class in *fipy.solvers.preconditioner*), 467
- SourceTerm (class in *fipy.terms.sourceTerm*), 594
- SphericalGrid1D() (in module *fipy.meshes.factory.Meshes*), 188
- SphericalNonUniformGrid1D (class in *fipy.meshes.sphericalNonUniformGrid1D*), 388
- SphericalUniformGrid1D (class in *fipy.meshes.sphericalUniformGrid1D*), 397
- Sphinx, 123
- sqrt, 972, 1109, 1170, 1201
- arcsin; cos, 1006, 1011
- sqrt() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 655
- sqrt() (*fipy.tools.PhysicalField* method), 633
- sqrtDot() (in module *fipy.tools.numerix*), 670
- SSORPreconditioner (class in *fipy.solvers.petsc.preconditioners*), 464
- SSORPreconditioner (class in *fipy.solvers.petsc.preconditioners.ssorPreconditioner*), 466
- StagnatedDivergence (class in *fipy.solvers.convergence*), 448
- StagnatedSolverWarning, 496
- status_code (*fipy.solvers.convergence.ConvergenceBase* attribute), 445
- status_name (*fipy.solvers.convergence.ConvergenceBase* attribute), 445
- std() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 691
- std() (*fipy.variables.cellVariable.CellVariable* method), 706
- std() (*fipy.variables.distanceVariable.DistanceVariable* method), 724
- std() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 740
- std() (*fipy.variables.faceVariable.FaceVariable* method), 751
- std() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 767
- std() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 783
- std() (*fipy.variables.histogramVariable.HistogramVariable* method), 797
- std() (*fipy.variables.meshVariable.MeshVariable* method), 808
- std() (*fipy.variables.modularVariable.ModularVariable* method), 823
- std() (*fipy.variables.noiseVariable.NoiseVariable* method), 838
- std() (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 849
- std() (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 861
- std() (*fipy.variables.surfactantVariable.SurfactantVariable* method), 876
- std() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 891
- std() (*fipy.variables.variable.Variable* method), 901
- SteadyConvectionDiffusionScEquation, 1222
- Stepper (class in *fipy.steps.stepper*), 518
- steppingstones, 124
- subtract() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 655
- subtract() (*fipy.tools.PhysicalField* method), 633
- sum() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 655
- sum() (*fipy.tools.PhysicalField* method), 634
- sum() (in module *fipy.tools.numerix*), 670
- SurfactantConvectionVariable (class in *fipy.variables.surfactantConvectionVariable*), 850
- SurfactantVariable, 1111

SurfactantVariable (class in *fipy.variables.surfactantVariable*), 861
 sweep, 1094, 1162, 1206
 sweep() (*fipy.terms.advectionTerm.AdvectionTerm* method), 531
 sweep() (*fipy.terms.cellTerm.CellTerm* method), 535
 sweep() (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 540
 sweep() (*fipy.terms.diffusionTerm.DiffusionTerm* method), 544
 sweep() (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* method), 548
 sweep() (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* method), 551
 sweep() (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* method), 554
 sweep() (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 560
 sweep() (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 565
 sweep() (*fipy.terms.faceTerm.FaceTerm* method), 569
 sweep() (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 574
 sweep() (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 579
 sweep() (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 584
 sweep() (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 589
 sweep() (*fipy.terms.residualTerm.ResidualTerm* method), 593
 sweep() (*fipy.terms.sourceTerm.SourceTerm* method), 597
 sweep() (*fipy.terms.term.Term* method), 601
 sweep() (*fipy.terms.transientTerm.TransientTerm* method), 605
 sweep() (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 611
 sweep() (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 616
 sweepMonotonic() (in module *fipy.steps*), 517

T

take() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 655
 take() (*fipy.tools.PhysicalField* method), 634
 take() (in module *fipy.tools.numerix*), 670
 tan, 1135, 1140
 tan() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 656
 tan() (*fipy.tools.PhysicalField* method), 634
 tanh, 1201
 tanh() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 656
 tanh() (*fipy.tools.PhysicalField* method), 634
 Term (class in *fipy.terms.term*), 598
 TermMultiplyError, 522
 test() (in module *fipy*), 139
 testmod() (in module *fipy.tests.doctestPlus*), 617
 TestProgram (class in *fipy.tests.testProgram*), 619
 Timer (class in *fipy.tools.timer*), 673
 title (exam-
 ples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer
 property), 1122
 title (exam-
 ples.levelSet.electroChem.mayaviSurfactantViewer.MayaviSurfactantViewer
 property), 1125
 title (examples.phase.phaseViewer.PhaseViewer property), 1181

title (*fipy.viewers.DummyViewer* property), 903
 title (*fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* property), 910
 title (*fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* property), 912
 title (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 915
 title (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* property), 917
 title (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* property), 920
 title (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* property), 923
 title (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* property), 926
 title (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 930
 title (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* property), 934
 title (*fipy.viewers.mayaviViewer.MayaviClient* property), 937
 title (*fipy.viewers.mayaviViewer.mayaviClient.MayaviClient* property), 941
 title (*fipy.viewers.multiViewer.MultiViewer* property), 943
 title (*fipy.viewers.tsvViewer.TSVViewer* property), 946
 title (*fipy.viewers.viewer.AbstractViewer* property), 948
 title (*fipy.viewers.vtkViewer.VTKCellViewer* property), 949
 title (*fipy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer* property), 953
 title (*fipy.viewers.vtkViewer.VTKFaceViewer* property), 951
 title (*fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer* property), 955
 title (*fipy.viewers.vtkViewer.vtkViewer.VTKViewer* property), 957
 tolerance_scale (*fipy.solvers.convergence.ConvergenceBase* attribute), 444
 ToleranceDivergence (class in *fipy.solvers.convergence*), 448
 toString() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 656
 toString() (*fipy.tools.PhysicalField* method), 635
 toString() (in module *fipy.tools.numerix*), 670
 TransientTerm, 1024, 1140, 1171, 1175, 1203
 TransientTerm (class in *fipy.terms.transientTerm*), 601
 TransientTermError, 522
 TravisCI, 124
 Tri2D (class in *fipy.meshes.tri2D*), 404
 TrilinosAztecOOSolver (class in *fipy.solvers.trilinos.trilinosAztecOOSolver*), 511
 TrilinosMLTest (class in *fipy.solvers.trilinos.trilinosMLTest*), 512
 TrilinosNonlinearSolver (class in *fipy.solvers.trilinos.trilinosNonlinearSolver*), 513
 TrilinosPreconditioner (class in *fipy.solvers.trilinos.preconditioners.trilinosPreconditioner*), 511
 TrilinosSolver (class in *fipy.solvers.trilinos.trilinosSolver*), 515
 TSVViewer (class in *fipy.viewers.tsvViewer*), 944

U

UniformGrid (class in *fipy.meshes.uniformGrid*), 413
 UniformGrid1D (class in *fipy.meshes.uniformGrid1D*), 420

UniformGrid2D (class in *fipy.meshes.uniformGrid2D*), 427
 UniformGrid3D (class in *fipy.meshes.uniformGrid3D*), 434
 UniformNoiseVariable (class in *fipy.variables.uniformNoiseVariable*), 877
 unit (*fipy.tools.dimensions.physicalField.PhysicalField* property), 656
 unit (*fipy.tools.PhysicalField* property), 635
 unit (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 691
 unit (*fipy.variables.cellVariable.CellVariable* property), 706
 unit (*fipy.variables.distanceVariable.DistanceVariable* property), 724
 unit (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 740
 unit (*fipy.variables.faceVariable.FaceVariable* property), 751
 unit (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 767
 unit (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 783
 unit (*fipy.variables.histogramVariable.HistogramVariable* property), 797
 unit (*fipy.variables.meshVariable.MeshVariable* property), 808
 unit (*fipy.variables.modularVariable.ModularVariable* property), 823
 unit (*fipy.variables.noiseVariable.NoiseVariable* property), 838
 unit (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 849
 unit (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 861
 unit (*fipy.variables.surfactantVariable.SurfactantVariable* property), 876
 unit (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 891
 unit (*fipy.variables.variable.Variable* property), 901
 update_pipeline() (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 942
 updateOld() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 691
 updateOld() (*fipy.variables.cellVariable.CellVariable* method), 706
 updateOld() (*fipy.variables.distanceVariable.DistanceVariable* method), 725
 updateOld() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 740
 updateOld() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 767
 updateOld() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 783
 updateOld() (*fipy.variables.histogramVariable.HistogramVariable* method), 798
 updateOld() (*fipy.variables.modularVariable.ModularVariable* method), 823
 updateOld() (*fipy.variables.noiseVariable.NoiseVariable* method), 838
 updateOld() (*fipy.variables.surfactantVariable.SurfactantVariable* method), 876
 updateOld() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 891
 UpwindConvectionTerm (class in *fipy.terms.upwindConvectionTerm*), 606

V

value (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 691

value (*fipy.variables.cellVariable.CellVariable* property), 706
 value (*fipy.variables.distanceVariable.DistanceVariable* property), 725
 value (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 740
 value (*fipy.variables.faceVariable.FaceVariable* property), 751
 value (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 767
 value (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 783
 value (*fipy.variables.histogramVariable.HistogramVariable* property), 798
 value (*fipy.variables.meshVariable.MeshVariable* property), 809
 value (*fipy.variables.modularVariable.ModularVariable* property), 824
 value (*fipy.variables.noiseVariable.NoiseVariable* property), 838
 value (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 849
 value (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 861
 value (*fipy.variables.surfactantVariable.SurfactantVariable* property), 876
 value (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 891
 value (*fipy.variables.variable.Variable* property), 901
 VanLeerConvectionTerm (class in *fipy.terms.vanLeerConvectionTerm*), 611
 Variable, 1142, 1156, 1205
 Variable (class in *fipy.variables.variable*), 892
 vars (examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer property), 1122
 vars (examples.levelSet.electroChem.mayaviSurfactantViewer.MayaviSurfactantViewer property), 1125
 vars (examples.phase.phaseViewer.PhaseViewer property), 1181
 vars (*fipy.viewers.DummyViewer* property), 903
 vars (*fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* property), 910
 vars (*fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* property), 912
 vars (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 915
 vars (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* property), 918
 vars (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* property), 921
 vars (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* property), 923
 vars (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* property), 926
 vars (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 930
 vars (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* property), 934
 vars (*fipy.viewers.mayaviViewer.MayaviClient* property), 937

- `vars (fipy.viewers.mayaviViewer.mayaviClient.MayaviClient property)`, 941
- `vars (fipy.viewers.multiViewer.MultiViewer property)`, 943
- `vars (fipy.viewers.tsvViewer.TSVViewer property)`, 946
- `vars (fipy.viewers.viewer.AbstractViewer property)`, 948
- `vars (fipy.viewers.vtkViewer.VTKCellViewer property)`, 949
- `vars (fipy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer property)`, 954
- `vars (fipy.viewers.vtkViewer.VTKFaceViewer property)`, 951
- `vars (fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer property)`, 955
- `vars (fipy.viewers.vtkViewer.vtkViewer.VTKViewer property)`, 957
- `VectorCoefficientError`, 523
- `view_data()`
 - (`fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon` method), 942
- `Viewer()` (in module `fipy.viewers`), 904
- `viewers`
 - module, 1198
- `VTKCellDataSet (fipy.meshes.abstractMesh.AbstractMesh property)`, 146
- `VTKCellDataSet`
 - (`fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D` property), 155
- `VTKCellDataSet`
 - (`fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D` property), 163
- `VTKCellDataSet`
 - (`fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D` property), 172
- `VTKCellDataSet`
 - (`fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D` property), 179
- `VTKCellDataSet (fipy.meshes.gmshMesh.Gmsh2D property)`, 193
- `VTKCellDataSet (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property)`, 202
- `VTKCellDataSet (fipy.meshes.gmshMesh.Gmsh3D property)`, 210
- `VTKCellDataSet (fipy.meshes.gmshMesh.GmshGrid2D property)`, 219
- `VTKCellDataSet (fipy.meshes.gmshMesh.GmshGrid3D property)`, 227
- `VTKCellDataSet (fipy.meshes.mesh.Mesh property)`, 237
- `VTKCellDataSet (fipy.meshes.mesh1D.Mesh1D property)`, 246
- `VTKCellDataSet (fipy.meshes.mesh2D.Mesh2D property)`, 254
- `VTKCellDataSet`
 - (`fipy.meshes.nonUniformGrid1D.NonUniformGrid1D` property), 263
- `VTKCellDataSet`
 - (`fipy.meshes.nonUniformGrid2D.NonUniformGrid2D` property), 271
- `VTKCellDataSet`
 - (`fipy.meshes.nonUniformGrid3D.NonUniformGrid3D` property), 280
- `VTKCellDataSet (fipy.meshes.periodicGrid1D.PeriodicGrid1D property)`, 288
- `VTKCellDataSet (fipy.meshes.periodicGrid2D.PeriodicGrid2D property)`, 298
- `VTKCellDataSet`
 - (`fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight` property), 306
- `VTKCellDataSet`
 - (`fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom` property), 315
- `VTKCellDataSet (fipy.meshes.periodicGrid3D.PeriodicGrid3D property)`, 325
- `VTKCellDataSet`
 - (`fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack` property), 332
- `VTKCellDataSet`
 - (`fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight` property), 340
- `VTKCellDataSet`
 - (`fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack` property), 348
- `VTKCellDataSet`
 - (`fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom` property), 356
- `VTKCellDataSet`
 - (`fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom` property), 363
- `VTKCellDataSet`
 - (`fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack` property), 371
- `VTKCellDataSet (fipy.meshes.skewedGrid2D.SkewedGrid2D property)`, 380
- `VTKCellDataSet`
 - (`fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D` property), 389
- `VTKCellDataSet`
 - (`fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D` property), 397
- `VTKCellDataSet (fipy.meshes.tri2D.Tri2D property)`, 405
- `VTKCellDataSet (fipy.meshes.uniformGrid.UniformGrid property)`, 413
- `VTKCellDataSet (fipy.meshes.uniformGrid1D.UniformGrid1D property)`, 420
- `VTKCellDataSet (fipy.meshes.uniformGrid2D.UniformGrid2D property)`, 427
- `VTKCellDataSet (fipy.meshes.uniformGrid3D.UniformGrid3D property)`, 434
- `VTKCellViewer` (class in `fipy.viewers.vtkViewer`), 948
- `VTKCellViewer` (class in `fipy.viewers.vtkViewer.vtkCellViewer`), 952
- `VTKFaceDataSet (fipy.meshes.abstractMesh.AbstractMesh property)`, 146
- `VTKFaceDataSet`
 - (`fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D` property), 155
- `VTKFaceDataSet`
 - (`fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D` property), 163
- `VTKFaceDataSet`
 - (`fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D` property), 172
- `VTKFaceDataSet`
 - (`fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D` property), 179
- `VTKFaceDataSet (fipy.meshes.gmshMesh.Gmsh2D property)`, 193
- `VTKFaceDataSet (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property)`, 202
- `VTKFaceDataSet (fipy.meshes.gmshMesh.Gmsh3D property)`, 210
- `VTKFaceDataSet (fipy.meshes.gmshMesh.GmshGrid2D property)`, 219
- `VTKFaceDataSet (fipy.meshes.gmshMesh.GmshGrid3D property)`, 227
- `VTKFaceDataSet (fipy.meshes.mesh.Mesh property)`, 237
- `VTKFaceDataSet (fipy.meshes.mesh1D.Mesh1D property)`, 246
- `VTKFaceDataSet (fipy.meshes.mesh2D.Mesh2D property)`, 254
- `VTKFaceDataSet`
 - (`fipy.meshes.nonUniformGrid1D.NonUniformGrid1D` property), 263
- `VTKFaceDataSet`
 - (`fipy.meshes.nonUniformGrid2D.NonUniformGrid2D` property), 271
- `VTKFaceDataSet`
 - (`fipy.meshes.nonUniformGrid3D.NonUniformGrid3D` property), 280
- `VTKFaceDataSet (fipy.meshes.periodicGrid1D.PeriodicGrid1D property)`, 289
- `VTKFaceDataSet (fipy.meshes.periodicGrid2D.PeriodicGrid2D property)`, 298

VTKFaceDataSet
(*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 306

VTKFaceDataSet
(*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 315

VTKFaceDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 325

VTKFaceDataSet
(*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 332

VTKFaceDataSet
(*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 340

VTKFaceDataSet
(*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 348

VTKFaceDataSet
(*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 356

VTKFaceDataSet
(*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 363

VTKFaceDataSet
(*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 371

VTKFaceDataSet (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 380

VTKFaceDataSet
(*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 389

VTKFaceDataSet
(*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 397

VTKFaceDataSet (*fipy.meshes.tri2D.Tri2D* property), 405

VTKFaceDataSet (*fipy.meshes.uniformGrid.UniformGrid* property), 413

VTKFaceDataSet (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 420

VTKFaceDataSet (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 427

VTKFaceDataSet (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 434

VTKFaceViewer (class in *fipy.viewers.vtkViewer*), 949

VTKFaceViewer (class in *fipy.viewers.vtkViewer.vtkFaceViewer*), 954

VTKViewer (class in *fipy.viewers.vtkViewer.vtkViewer*), 956

VTKViewer() (in module *fipy.viewers.vtkViewer*), 951

W

Weave, 124

Windows, 124

with_traceback() (*fipy.meshes.abstractMesh.MeshAdditionError* method), 153

with_traceback() (*fipy.meshes.gmshMesh.GmshException* method), 218

with_traceback() (*fipy.meshes.gmshMesh.MeshExportError* method), 236

with_traceback() (*fipy.meshes.mesh.MeshAdditionError* method), 245

with_traceback() (*fipy.solvers.convergence.DivergenceWarning* method), 446

with_traceback() (*fipy.solvers.SerialSolverError* method), 442

with_traceback() (*fipy.solvers.solver.BreakdownWarning* method), 489

with_traceback()
(*fipy.solvers.solver.IllConditionedPreconditionerWarning* method), 490

with_traceback()
(*fipy.solvers.solver.IllegalInputOrBreakdownWarning* method), 490

with_traceback() (*fipy.solvers.solver.LossOfPrecisionWarning* method), 491

with_traceback() (*fipy.solvers.solver.MatrixIllConditionedWarning* method), 491

with_traceback() (*fipy.solvers.solver.MaximumIterationWarning* method), 492

with_traceback() (*fipy.solvers.solver.NormalConvergence* method), 492

with_traceback() (*fipy.solvers.solver.ParameterWarning* method), 493

with_traceback()
(*fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning* method), 494

with_traceback() (*fipy.solvers.solver.PreconditionerWarning* method), 494

with_traceback()
(*fipy.solvers.solver.ScalarQuantityOutOfRangeWarning* method), 495

with_traceback() (*fipy.solvers.solver.SolverConvergenceWarning* method), 496

with_traceback() (*fipy.solvers.solver.StagnatedSolverWarning* method), 497

with_traceback() (*fipy.terms.AbstractBaseClassError* method), 519

with_traceback() (*fipy.terms.ExplicitVariableError* method), 520

with_traceback() (*fipy.terms.IncorrectSolutionVariable* method), 521

with_traceback() (*fipy.terms.SolutionVariableNumberError* method), 521

with_traceback() (*fipy.terms.SolutionVariableRequiredError* method), 522

with_traceback() (*fipy.terms.TermMultiplyError* method), 522

with_traceback() (*fipy.terms.TransientTermError* method), 523

with_traceback() (*fipy.terms.VectorCoeffError* method), 523

with_traceback() (*fipy.viewers.MeshDimensionError* method), 904

write() (in module *fipy.tools.dump*), 664

X

x (*fipy.meshes.abstractMesh.AbstractMesh* property), 152

x
(*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 162

x
(*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 171

x (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 178

x (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 185

x (*fipy.meshes.gmshMesh.Gmsh2D* property), 201

x (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 209

x (*fipy.meshes.gmshMesh.Gmsh3D* property), 217

x (*fipy.meshes.gmshMesh.GmshGrid2D* property), 226

x (*fipy.meshes.gmshMesh.GmshGrid3D* property), 234

x (*fipy.meshes.mesh.Mesh* property), 244

x (*fipy.meshes.mesh1D.Mesh1D* property), 253

x (*fipy.meshes.mesh2D.Mesh2D* property), 262

x (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 270

x (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 279

x (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 287

x (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 296

x (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 306

x (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 314

x (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom property), 322

x (fipy.meshes.periodicGrid3D.PeriodicGrid3D property), 332

x (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property), 339

x (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property), 347

x (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property), 355

x (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property), 363

x (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property), 370

x (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property), 378

x (fipy.meshes.skewedGrid2D.SkewedGrid2D property), 387

x

(fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property), 396

x (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property), 403

x (fipy.meshes.tri2D.Tri2D property), 412

x (fipy.meshes.uniformGrid.UniformGrid property), 419

x (fipy.meshes.uniformGrid1D.UniformGrid1D property), 426

x (fipy.meshes.uniformGrid2D.UniformGrid2D property), 433

x (fipy.meshes.uniformGrid3D.UniformGrid3D property), 440

Y

y (fipy.meshes.abstractMesh.AbstractMesh property), 152

y

(fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D property), 162

y

(fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D property), 171

y (fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property), 178

y (fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property), 185

y (fipy.meshes.gmshMesh.Gmsh2D property), 201

y (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property), 209

y (fipy.meshes.gmshMesh.Gmsh3D property), 217

y (fipy.meshes.gmshMesh.GmshGrid2D property), 226

y (fipy.meshes.gmshMesh.GmshGrid3D property), 234

y (fipy.meshes.mesh.Mesh property), 244

y (fipy.meshes.mesh1D.Mesh1D property), 253

y (fipy.meshes.mesh2D.Mesh2D property), 262

y (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D property), 270

y (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D property), 279

y (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D property), 287

y (fipy.meshes.periodicGrid1D.PeriodicGrid1D property), 296

y (fipy.meshes.periodicGrid2D.PeriodicGrid2D property), 306

y (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight property), 314

y (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom property), 322

y (fipy.meshes.periodicGrid3D.PeriodicGrid3D property), 332

y (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property), 340

y (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property), 347

y (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property), 355

y (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property), 363

y (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property), 371

y (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property), 378

y (fipy.meshes.skewedGrid2D.SkewedGrid2D property), 387

y

(fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property), 396

y (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property), 403

y (fipy.meshes.tri2D.Tri2D property), 412

y (fipy.meshes.uniformGrid.UniformGrid property), 419

y (fipy.meshes.uniformGrid1D.UniformGrid1D property), 426

y (fipy.meshes.uniformGrid2D.UniformGrid2D property), 433

y (fipy.meshes.uniformGrid3D.UniformGrid3D property), 441

Z

z (fipy.meshes.abstractMesh.AbstractMesh property), 152

z

(fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D property), 162

z

(fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D property), 171

z (fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property), 178

z (fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property), 185

z (fipy.meshes.gmshMesh.Gmsh2D property), 201

z (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property), 210

z (fipy.meshes.gmshMesh.Gmsh3D property), 218

z (fipy.meshes.gmshMesh.GmshGrid2D property), 227

z (fipy.meshes.gmshMesh.GmshGrid3D property), 234

z (fipy.meshes.mesh.Mesh property), 245

z (fipy.meshes.mesh1D.Mesh1D property), 253

z (fipy.meshes.mesh2D.Mesh2D property), 262

z (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D property), 271

z (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D property), 279

z (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D property), 287

z (fipy.meshes.periodicGrid1D.PeriodicGrid1D property), 296

z (fipy.meshes.periodicGrid2D.PeriodicGrid2D property), 306

z (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight property), 314

z (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom property), 323

z (fipy.meshes.periodicGrid3D.PeriodicGrid3D property), 332

z (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property), 340

z (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property), 348

z (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property), 355

z (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property), 363

z (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property), 371

z (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property), 379

z (fipy.meshes.skewedGrid2D.SkewedGrid2D property), 388

z

(fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property), 396

z (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property), 403

z (fipy.meshes.tri2D.Tri2D property), 413

z (fipy.meshes.uniformGrid.UniformGrid property), 420

z (fipy.meshes.uniformGrid1D.UniformGrid1D property), 427

z (fipy.meshes.uniformGrid2D.UniformGrid2D property), 434

z (fipy.meshes.uniformGrid3D.UniformGrid3D property), 441