

# **FiPy Manual**

***Release 3.4.4+22.gf9cd26767***

**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

Feb 22, 2024

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

<b>I</b>	<b>Introduction</b>	<b>1</b>
1	Overview	3
2	Installation	7
3	Git practices	15
4	Continuous Integration	17
5	Conda Lockfiles	19
6	Making a Release	21
7	Solvers	25
8	Viewers	29
9	Using FiPy	31
10	Frequently Asked Questions	49
11	Efficiency	59
12	Theoretical and Numerical Background	61
13	Design and Implementation	69
14	Virtual Kinetics of Materials Laboratory	75
15	Contributors	77
16	Publications	79
17	Presentations	81
18	Change Log	83
19	Glossary	107

<b>II Examples</b>	<b>111</b>
20 Selected Examples	115
<b>III fipy Package Documentation</b>	<b>119</b>
21 How to Read the Modules Documentation	121
22 fipy	125
23 examples	919
<b>Bibliography</b>	<b>1177</b>
<b>Python Module Index</b>	<b>1179</b>
<b>Index</b>	<b>1185</b>

## **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<sup>o</sup>nda.org 3.4.4

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

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

## 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 future packaging mpich \  
  mpi4py petsc4py mayavi "gms <4.0|>=4.5.2"
```

or:

```
$ conda create --name <MYFIPYENV> --channel conda-forge \
  python=2.7 numpy scipy matplotlib-base future packaging \
  pyparse mayavi "traitsui<7.0.0" "gmsh<4.0"
```

**Attention:** Bit rot has started to set in for Python 2.7. One consequence is that *VTKViewers* can raise errors (probably other uses of *Mayavi*, too). Hence, the constraint of “*traitsui<7.0.0*”.

### 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. Significantly, you will need to download and install *Gmsh* manually when using Python 2.7.

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

## 2.3 Obtaining FiPy

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

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

## 2.4 Installing FiPy

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

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

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

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

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

or by unpacking the archive and typing:

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

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

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

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

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

## 2.5 Required Packages

### 2.5.1 Python

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

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

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

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

---

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

---

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

## 2.5.2 NumPy

<http://numpy.scipy.org>

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

## 2.6 Optional Packages

### 2.6.1 Gmsh

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

*Gmsh* is an application that allows the creation of irregular meshes. When running in parallel, *FiPy* requires a version of *Gmsh*  $\geq 2.5$  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 setup.py develop
```

## 2.9 Git usage

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

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



## 2.9.1 Git client

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

## 2.9.2 Git branches

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

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

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

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

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

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

### **master**

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

Past releases of *FiPy* are tagged as

### **x.y.z**

Any released version of *FiPy* will be designated with a fixed tag: The current version of *FiPy* is 3.4.4+22.gf9cd26767. (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 "python setup.py test"
```

---

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

---

# Chapter 3

## Git practices

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

### 3.1 Branches

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

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

Edit and add to branch:

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

#### 3.1.1 Merging changes from master to the branch

Make sure master is up to date:

```
$ git fetch origin
```

Merge updated state of master to the branch:

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

Resolve any conflicts and test:

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

### 3.1.2 Submit branch for code review

If necessary, fork the `fipy` repository.

Add a “remote” link to your fork:

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

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

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

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

---

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

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

---

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

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

### 3.1.3 Refactoring complete: merge branch to master

**Attention: Administrators Only!**

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

---

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

---

# Chapter 4

## Continuous Integration



We use the *Azure* and *GitHub Actions* cloud services for *Continuous Integration* (CI). These CIs are configured in *FiPySource/.azure/pipelines.yml*, *FiPySource/.github/workflows/NISTtheDocs2Death.yml*, and *FiPySource/.github/workflows/nix.yml*.



# Chapter 5

## Conda Lockfiles

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

```
$ for solver in petsc pyparse 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.





# Chapter 6

## Making a Release

**Attention: Administrators Only!**

### 6.1 Source

Make sure master is ready for release:

```
$ git checkout master
```

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

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

**Note:** You can use:

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

or:

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

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

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

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

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

## 6.2 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`
- `dist-Windows_NT/FiPy-x.y.zip`
- `dist-docs/FiPy-x.y.pdf`
- `dist-docs/html-x.y.tar.gz`

From the `FiPySource` directory, unpack `dist/html-x.y.tar.gz` into `docs/build` with:

```
$ tar -xzf dist/html-{x.y}.tar.gz -C docs/build
```

## 6.3 Upload

### Attach

- `dist/FiPy-x.y-none-any.whl`
- `dist/FiPy-x.y.tar.gz`
- `dist/FiPy-x.y.zip`
- `dist/FiPy-x.y.pdf`

to a [GitHub release](#) associated with tag `x.y`.

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

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

Upload the web site to CTCMS

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

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

## 6.4 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

## 6.5 Announce

Make an announcement to [fipy@list.nist.gov](mailto:fipy@list.nist.gov)



# Chapter 7

## Solvers

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

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

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

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

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

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

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

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

## 7.1 PETSc

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

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

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

---

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

---

## 7.2 Pysparse

<http://pysparse.sourceforge.net>

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

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

## 7.3 SciPy

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

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

## 7.4 PyAMG

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

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

## 7.5 pyamgx

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

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

## 7.6 Trilinos

<http://trilinos.sandia.gov>

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

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

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

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

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

---

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

---

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

**Gen\_Prolongator warning : Max eigen <= 0.0**

which is not significant to *FiPy*.

**Aztec status AZ\_loss: loss of precision**

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

**Aztec status AZ\_ill\_cond: GMRES hessenberg ill-conditioned**

which indicates that GMRES is having trouble with the problem, and may indicate that trying a different solver or preconditioner may give more accurate results if GMRES fails.

**Aztec status AZ\_breakdown: numerical breakdown**

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





## Viewers

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

### 8.1 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.

### 8.2 Mayavi

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

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

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

---

**Note:** MayaVi 1 is no longer supported.

---



# Chapter 9

## 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*.

## 9.1 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.

## 9.2 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:

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

---

**Note:** You may need to first run:

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

for this to work properly.

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

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

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Skipped 131 doctest examples because `gmsh` cannot be found on the $PATH
Skipped 42 doctest examples because the `vtk` 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.

## 9.2.1 Parallel Tests

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

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

or:

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

## 9.3 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.

### 9.3.1 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:

**--pysparse**

Forces the use of the *Pysparse* solvers.

**--trilinos**

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

**--no-pysparse**

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

**--scipy**

Forces the use of the *SciPy* solvers.

**--pyamg**

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

**--pyamgx**

Forces the use of the *pyamgx* solvers.

**--lsmllib**

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

**--skfmm**

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

### 9.3.2 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”, “scipy”, “pysparse”, “trilinos”, “no-pysparse”, and “pyamg”.

**FIPY\_VERBOSE\_SOLVER**

If present, causes the *LinearGeneralSolver* to print a variety of diagnostic information. All other solvers should use *Logging* and *FIPY\_LOG\_CONFIG*.

**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*.

## 9.4 Solving in Parallel

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

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

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

```
$ export OMP_NUM_THREADS=1
```

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

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

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

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

or:

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

instead of:

```
$ python myScript.py
```

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

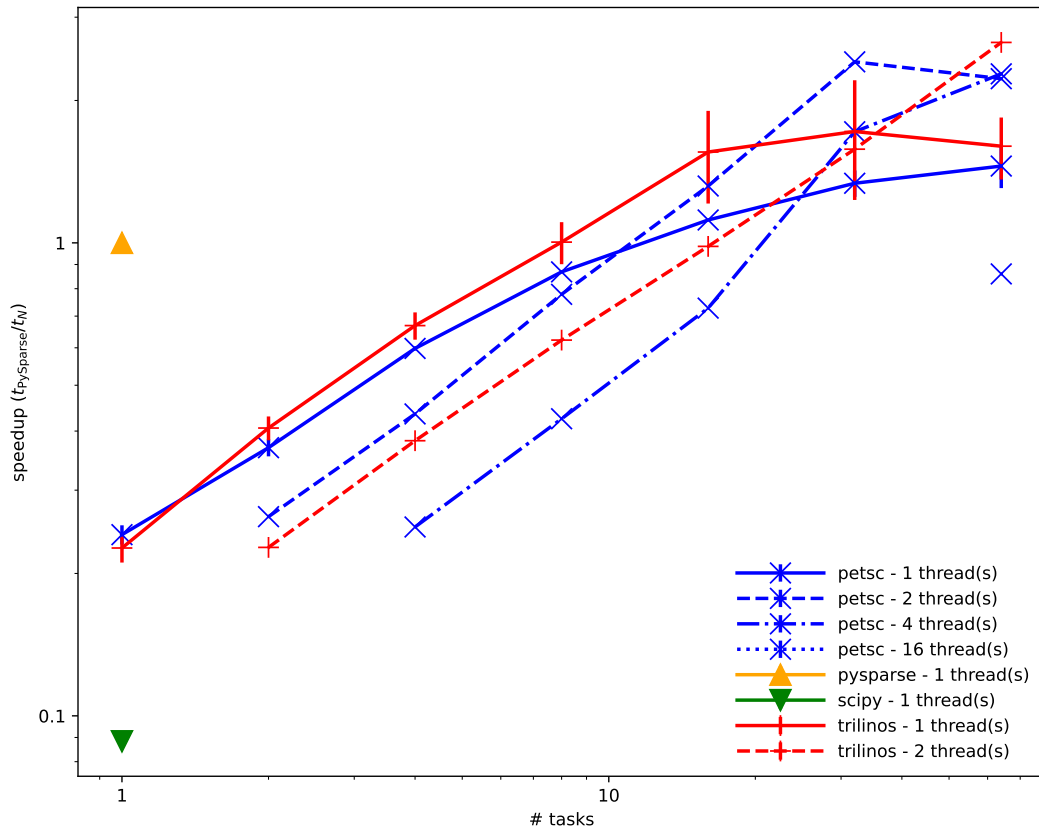


Fig. 1: Scaling behavior of different solver packages



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

A few things can be observed in this plot:

- Both *PETSc* and *Trilinos* exhibit power law scaling, but the power is only about 0.7. At least one source of this poor scaling is that our “...Grid...” meshes parallelize by dividing the mesh into slabs, which leads to more communication overhead than more compact partitions. The “...Gmsh...” meshes partition more efficiently, but carry more overhead in other ways. We’ll be making efforts to improve the partitioning of the “...Grid...” meshes in a future release.
- *PETSc* and *Trilinos* have fairly comparable performance, but lag *Pysparse* by a considerable margin. The *SciPy* solvers are even worse. Some of this discrepancy may be because the different packages are not all doing the same thing. Different solver packages have different default solvers and preconditioners. Moreover, the meaning of the solution tolerance depends on the normalization the solver uses and it is not always obvious which of several possibilities a particular package employs. We will endeavor to normalize the normalizations in a future release.
- *PETSc* with one thread is faster than with two threads until the number of tasks reaches about 10 and is faster than with four threads until the number of tasks reaches more than 20. *Trilinos* with one thread is faster than with two threads until the number of tasks is more than 30. We don’t fully understand the reasons for this, but there may be a *modest* benefit, *when using a large number of cpus*, to allow two to four *OpenMP* threads per *MPI* rank. See *OpenMP Threads vs. MPI Ranks* for caveats and more information.

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

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

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

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

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

which should print out:

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

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

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

indicates *mpi4py* is properly communicating with *MPI* and is running in parallel, but that *Trilinos* is not, and is running three separate serial environments. As a result, *FiPy* is limited to three separate serial operations, too. In this instance,

the problem is that although *Trilinos* was compiled with *MPI* enabled, it was compiled against a different *MPI* library than is currently available (and which *mpi4py* was compiled against). The solution, in this instance, is to solve with *PETSc* or to rebuild *Trilinos* against the active *MPI* libraries.

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

---

**Note:** One option for debugging in parallel is:

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

---

### 9.4.1 OpenMP Threads vs. MPI Ranks

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

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

```
$ export OMP_NUM_THREADS=1
```

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

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

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

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

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

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

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

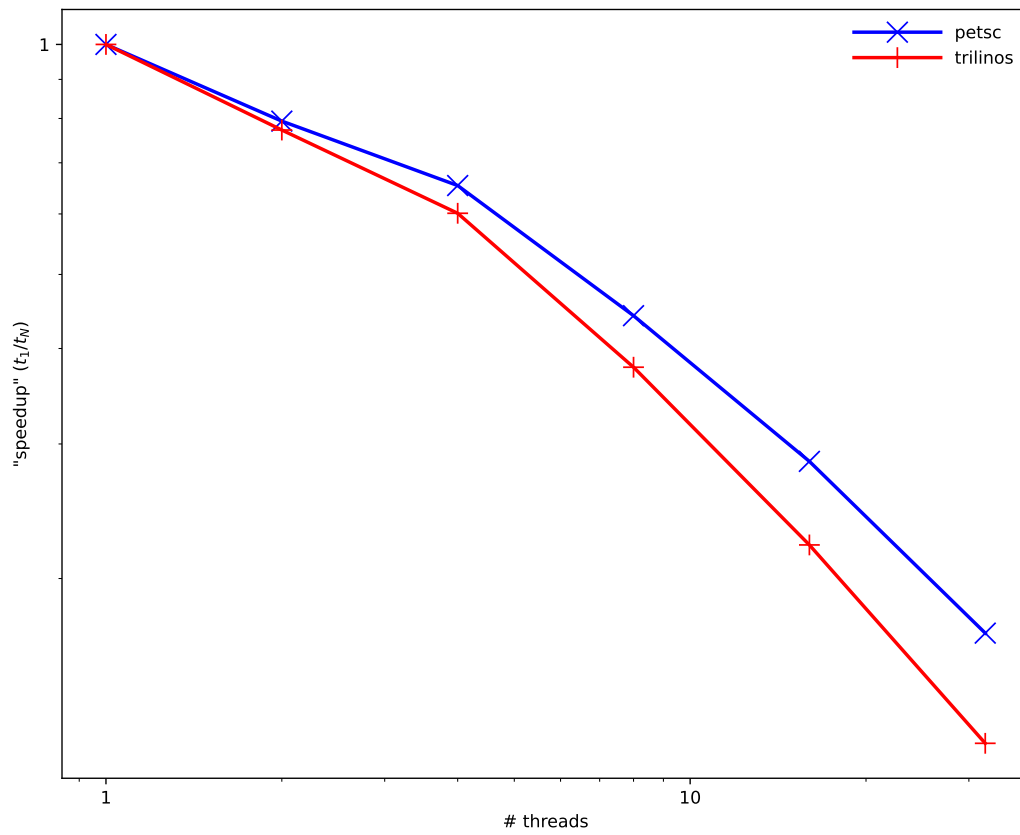


Fig. 2: Effect of having more *OpenMP* threads for each *MPI* rank

## 9.5 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.

## 9.6 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

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

*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

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

## 9.7 Boundary Conditions

### 9.7.1 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

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

### 9.7.2 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.

### 9.7.3 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)
```

### 9.7.4 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.

---

### 9.7.5 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.

### 9.7.6 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](#)

### 9.7.7 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:

$$\begin{aligned} \phi_f &\approx \phi_P + (\vec{d}_{Pf} \cdot \nabla\phi)_f \\ &\approx \phi_P + (\hat{n} \cdot \nabla\phi)_f (\vec{d}_{Pf} \cdot \hat{n})_f \end{aligned} \quad (9.1)$$

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:

$$\begin{aligned} \hat{n} \cdot (\vec{a}\phi + b\nabla\phi)_f &= g \\ \hat{n} \cdot \left[ \vec{a}\phi_P + \vec{a}(\hat{n} \cdot \nabla\phi)_f (\vec{d}_{Pf} \cdot \hat{n})_f + b\nabla\phi \right]_f &\approx g \\ (\hat{n} \cdot \nabla\phi)_f &\approx \frac{g_f - (\hat{n} \cdot \vec{a})_f \phi_P}{(\vec{d}_{Pf} \cdot \vec{a})_f + b_f} \end{aligned} \quad (9.2)$$

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 (9.1)

$$\begin{aligned}\phi_f &\approx \phi_P + \frac{g_f - (\hat{n} \cdot \vec{a})_f \phi_P}{\left(\vec{d}_{Pf} \cdot \vec{a}\right)_f + b_f} \left(\vec{d}_{Pf} \cdot \hat{n}\right)_f \\ &\approx \frac{g_f \left(\hat{n} \cdot \vec{d}_{Pf}\right)_f + b_f \phi_P}{\left(\vec{d}_{Pf} \cdot \vec{a}\right)_f + b_f}\end{aligned}\tag{9.3}$$

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

Substituting (9.2) 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 (9.3):

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

**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  $g \equiv h T_\infty$ .



## 9.7.8 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  $\phi = 0.25$ .

## 9.8 Running under Python 2

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

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

Because the faster *PySparse* and *Trilinos* solvers are not available under *Python 3*, we will maintain *Python 2.x* support

as long as practical. Be aware that the `conda-forge` packages that *FiPy* depends upon are not well-maintained on *Python* 2.x and our support for that configuration is rapidly becoming impractical, despite the present performance benefits. Hopefully, we will learn how to optimize our use of *PETSc* and/or *Trilinos* 12.12 will become available on `conda-forge`.

## 9.9 Manual

You can view the manual online at <http://pages.nist.gov/fipy>. 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. Python 2.7 probably won't work.

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 10

## Frequently Asked Questions

### 10.1 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*  $\phi$  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*.

#### 10.1.1 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.

### 10.1.2 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  $\phi$  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  $\xi$ , consistent with the matrix that is formed by *DiffusionTerm* for  $\phi$ .

---

#### How do I represent a convective source?

The convection of an independent field  $\xi$  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  $\xi$ , 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 (12.3), as

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

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

### 10.1.3 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  $\xi$  with a mobility  $D_2\phi$  that is a function of the dependent variable  $\phi$ . For *FiPy*'s purposes, however, this term represents the convection of  $\phi$ , with a velocity  $D_2\nabla\xi$ , due to the counter-diffusion of  $\xi$ , 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*.

### 10.1.4 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!*.

## 10.2 How can I see what I'm doing?

### 10.2.1 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](#).

### 10.2.2 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>.

### 10.2.3 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*.



## 10.2.4 Why doesn't the Viewer look the way I want?

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

## 10.3 Iterations, timesteps, and sweeps? Oh, my!

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

### iterations

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

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

---

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

---

Solver iterations are changed from their defaults in [examples.flow.stokesCavity](#) and [examples.updating.update0\\_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 = LinearPCGSolver(iterations=1234, tolerance=5e-6)
:
:
>>> while elapsedTime < totalElapsedTime:
...     myVar.updateOld()
...     while residual > desiredResidual:
...         residual = eq.sweep(var=myVar, dt=timeStep, ...)
...         elapsedTime += timeStep
```

## 10.4 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.

## 10.5 How do I represent boundary conditions?

See the *Boundary Conditions* section for more details.

## 10.6 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/maillinglists/mailman?fn=scipy-dev/2005-June/003010.html>.

### **numerix Numeric 23.6**

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

## 10.7 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.

## 10.8 How can I tell if I'm running in parallel?

See *Solving in Parallel*.

## 10.9 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*.

## 10.10 What if my question isn't answered here?

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



# Chapter 11

## 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.

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

## 11.1 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)
<i>examples.phase. impingement.mesh20x20</i>	4.3	4.1	1.4	2.3
<i>examples.phase.anisotropy</i>	3.5	3.2	1.1	1.9
<i>examples.cahnHilliard. mesh2D</i>	3.0	2.5	1.1	1.4
<i>examples.levelSet. electroChem. simpleTrenchSystem</i>	62	62	2.0	2.8

## 11.2 Efficiency discussion of Pysparse and Trilinos

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



## Theoretical and Numerical Background

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

### 12.1 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  $\phi$  such as species concentration, pH, or temperature, there exists an evolution equation of the form

$$\frac{\partial \phi}{\partial t} = H(\phi, \lambda_i) \quad (12.1)$$

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

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

$$\underbrace{\frac{\partial(\rho\phi)}{\partial t}}_{\text{transient}} + \underbrace{\nabla \cdot (\vec{u}\phi)}_{\text{convection}} = \underbrace{[\nabla \cdot (\Gamma_i \nabla)]^n}_{\text{diffusion}} \phi + \underbrace{S_\phi}_{\text{source}} \quad (12.2)$$

where  $\rho$ ,  $\vec{u}$  and  $\Gamma_i$  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  $\phi$  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.

## 12.2 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.

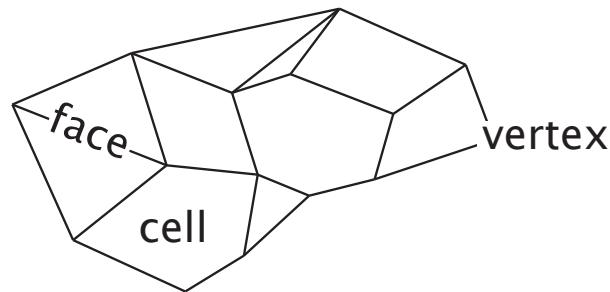


Fig. 1: Mesh

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.

### 12.2.1 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 (12.2), the boundary condition is

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

## 12.2.2 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.

## 12.3 Discretization

The first step in the discretization of Equation (12.2) 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 (12.2) will be examined separately.

### 12.3.1 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

$$\int_V \frac{\partial(\rho\phi)}{\partial t} dV \simeq \frac{(\rho_P\phi_P - \rho_P^{\text{old}}\phi_P^{\text{old}})V_P}{\Delta t} \quad (12.3)$$

where  $\phi_P$  represents the average value of  $\phi$  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  $\Delta t$  is the time step size.

This term is represented in *FiPy* as

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

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

The discretization for the convection term is given by

$$\begin{aligned} \int_V \nabla \cdot (\vec{u}\phi) dV &= \int_S (\vec{n} \cdot \vec{u})\phi dS \\ &\simeq \sum_f (\vec{n} \cdot \vec{u})_f \phi_f A_f \end{aligned} \quad (12.4)$$

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  $\phi_f$  must depend on the average value in adjacent cell  $\phi_A$  and the average value in the cell of interest  $\phi_P$ , such that

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

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

This term is represented in *FiPy* as

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

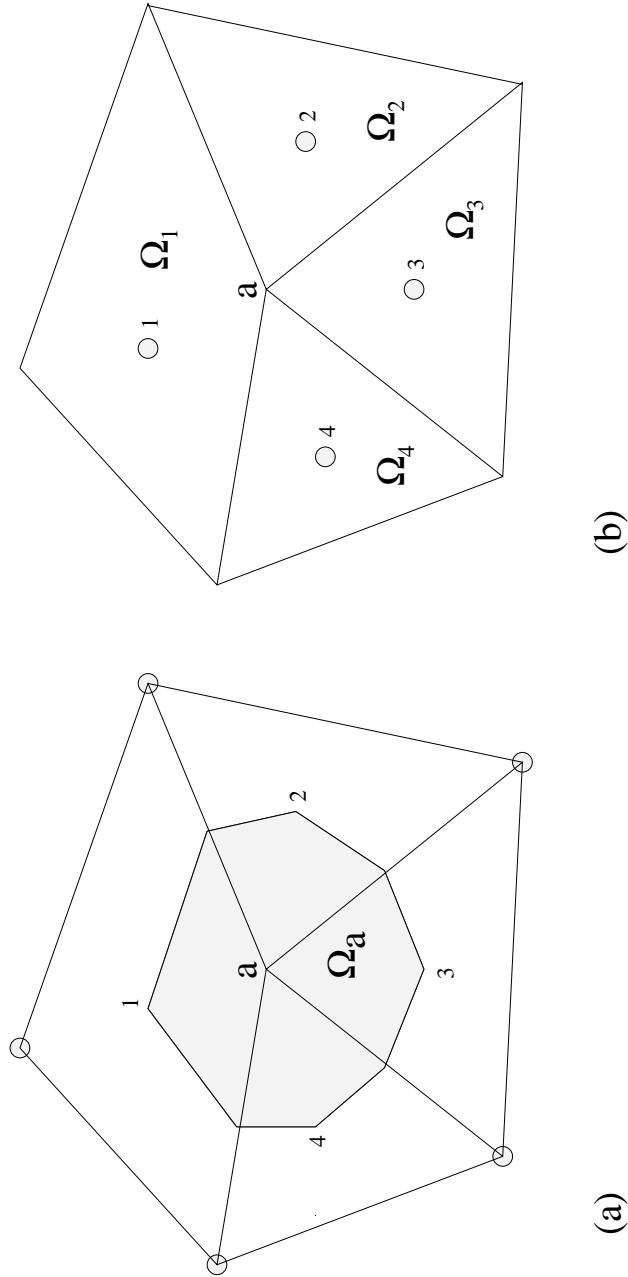


Fig. 2: CV structure for an unstructured mesh  
 (a)  $\Omega_a$  represents a vertex-based CV and (b)  $\Omega_1$ ,  $\Omega_2$ ,  $\Omega_3$  and  $\Omega_4$  represent cell centered CVs.

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.

### 12.3.3 Diffusion Term $\nabla \cdot (\Gamma_1 \nabla \phi)$

The discretization for the diffusion term is given by

$$\int_V \nabla \cdot (\Gamma \nabla \{ \dots \}) dV = \int_S \Gamma (\vec{n} \cdot \nabla \{ \dots \}) dS \tag{12.5}$$

$$\simeq \sum_f \Gamma_f (\vec{n} \cdot \nabla \{ \dots \})_f A_f$$

`{...}` indicates recursive application of the specified operation on  $\phi$ , 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.

---

### 12.3.4 Source Term

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

$$\int_V S_\phi dV \simeq S_\phi V_P. \quad (12.6)$$

Including any negative dependence of  $S_\phi$  on  $\phi$  increases solution stability. The dependence can only be included in a linear manner so Equation (12.6) becomes

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

where  $S_0$  is the source which is independent of  $\phi$  and  $S_1$  is the coefficient of the source which is linearly dependent on  $\phi$ .

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))
```

## 12.4 Linear Equations

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

Combining Equations (12.3), (12.4), (12.5) and (12.6), the complete discretization for equation (12.2) 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 (12.7) 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

$$a_P \phi_P = \sum_f a_A \phi_A + b_P, \quad (12.7)$$

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

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

## 12.5 Numerical Schemes

The coefficients of equation (12.7) must remain positive, since an increase in a neighboring value must result in an increase in  $\phi_P$  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

$$\frac{1}{1 - \alpha_f} > P_f > -\frac{1}{\alpha_f} \quad (12.8)$$

must be satisfied. The parameter  $\alpha_f$  is defined by the chosen scheme, depending on Equation (12.8). The various differencing schemes are:

### the central differencing scheme,

where

$$\alpha_f = \frac{1}{2} \quad (12.9)$$

so that  $|P_f| < 2$  satisfies Equation (12.8). Thus, the central differencing scheme is only numerically stable for a low values of  $P_f$ .

### the upwind scheme,

where

$$\alpha_f = \begin{cases} 1 & \text{if } P_f > 0, \\ 0 & \text{if } P_f < 0. \end{cases} \quad (12.10)$$

Equation (12.10) satisfies the inequality in Equation (12.8) 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

$$\alpha_f = \frac{(P_f - 1) \exp(P_f) + 1}{P_f (\exp(P_f) - 1)}. \quad (12.11)$$

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

$$\alpha_f = \begin{cases} \frac{P_f-1}{P_f} & \text{if } P_f > 2, \\ \frac{1}{2} & \text{if } |P_f| < 2, \\ -\frac{1}{P_f} & \text{if } P_f < -2. \end{cases} \quad (12.12)$$

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

$$\alpha_f = \begin{cases} \frac{P_f-1}{P_f} & \text{if } P_f > 10, \\ \frac{(P_f-1)+(1-P_f/10)^5}{P_f} & \text{if } 0 < P_f < 10, \\ \frac{(1-P_f/10)^5-1}{P_f} & \text{if } -10 < P_f < 0, \\ -\frac{1}{P_f} & \text{if } P_f < -10. \end{cases} \quad (12.13)$$

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.

### 13.1 Design

#### 13.1.1 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.

#### 13.1.2 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.

### 13.1.3 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.

### 13.1.4 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.

### 13.1.5 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 time<sup>1</sup>, 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)

---

<sup>1</sup> ... 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 inefficient<sup>2</sup>.

<sup>2</sup> A discussion of efficiency issues can be found in *Efficiency*.

### 13.1.6 Python Programming Language

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

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

## 13.2 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  $\phi$  at the centers of the cells of the *Mesh*. A *CellVariable* describes the values of the field  $\phi$ , 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 (12.2) 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 (12.2), 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 (12.2).

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

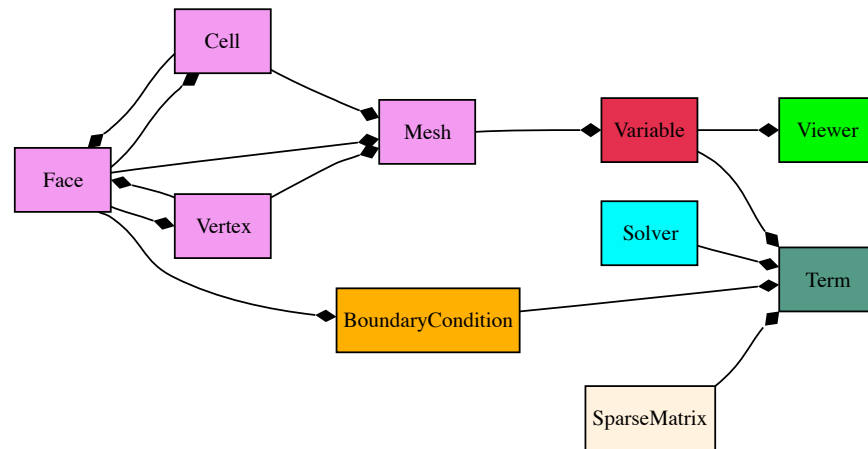


Fig. 1: Primary object relationships in *FiPy*.

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.



# 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 15

## 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 16

## 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. (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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.
- 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.

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

## 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 18

## Change Log

### 18.1 Version 3.4.4+22.gf9cd26767

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

#### 18.1.1 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)

#### 18.1.2 Fixes

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

### 18.2 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.

## 18.2.1 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)

## 18.2.2 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

## 18.3 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.

### 18.3.1 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)

### 18.3.2 Fixes

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

## 18.4 Version 3.4.1 - 2020-02-14

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

### 18.4.1 Pulls

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

### 18.4.2 Fixes

- #703: FORTRAN array ordering is deprecated

## 18.5 Version 3.4 - 2020-02-06

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

### 18.5.1 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)

### 18.5.2 Fixes

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

## 18.6 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.

### 18.6.1 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.

### 18.6.2 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

## 18.7 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*).

## 18.7.1 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)

## 18.7.2 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”

## 18.8 Version 3.1.3 - 2017-01-17

### 18.8.1 Fixes

- #502: `gmane` is defunct

## 18.9 Version 3.1.2 - 2016-12-24

### 18.9.1 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)

## 18.9.2 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  $\geq 2.10$  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

## 18.10 Version 3.1.1 - 2015-12-17

### 18.10.1 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  $\geq 2.0$  errors on *zizou*

## 18.11 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.

### 18.11.1 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?*

## 18.12 Version 3.0.1 - 2012-10-03

### 18.12.1 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

## 18.13 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.

### 18.13.1 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

## 18.14 Version 2.1.3 - 2012-01-17

### 18.14.1 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

## 18.15 Version 2.1.2 - 2011-04-20

The significant changes since version 2.1.1 are:

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

### 18.15.1 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*

## 18.16 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.

### 18.16.1 Fixes

- #199: dummy viewer results in “*NotImplementedError: can't instantiate abstract base class*”
- #198: bug problem with *CylindricalGridID*
- #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

## 18.17 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...<sup>1</sup>) 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.

---

<sup>1</sup> Only two examples from *FiPy* 2.0 fail when run with *FiPy* 2.1:

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

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

### 18.17.1 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

## 18.18 Version 2.0.3 - 2010-03-17

### 18.18.1 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: `gmsExport.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

## 18.19 Version 2.0.2 - 2009-06-11

### 18.19.1 Fixes

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

## 18.20 Version 2.0.1 - 2009-04-23

### 18.20.1 Fixes

- #154: Update manuals

## 18.21 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

## 18.22 Version 1.2.3 - 2009-01-0

### 18.22.1 Fixes

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

## 18.23 Version 1.2.2 - 2008-12-30

### 18.23.1 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

## 18.24 Version 1.2.1 - 2008-02-08

### 18.24.1 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

## 18.25 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.

### 18.25.1 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

## 18.26 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](#).

## 18.27 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`.

### 18.27.1 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

## 18.28 Version 0.1.1

## 18.29 Version 0.1 - 2004-11-05

Original release

# Chapter 19

## Glossary

### AppVeyor

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

### Azure

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

### Buildbot

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

### CircleCI

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

### conda

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

### Continuous Integration

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

### FiPy

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

### GitHub Actions

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

### 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/>.

### 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>.

**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>.

**PETSc**

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

**petsc4py**

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

**pip**

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

**PyAMG**

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



**pyamgx**

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

**PyPI**

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

**Pyrex**

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

**Pysparse**

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

**Python**

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

**Python 3**

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

**PyTrilinos**

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

**PyxViewer**

A now defunct python viewer.

**ScientificPython**

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

**SciPy**

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

**Sphinx**

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

**TravisCI**

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

**Trilinos**

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

**Weave**

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

**Windows**

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



**Part II**

**Examples**



---

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

---

These examples can be used in at least four ways:

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

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

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

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

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

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

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

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

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

---

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

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

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

---

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



# Chapter 20

## Selected Examples

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

### 20.1 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`

### 20.2 Convection Examples

Selected illustrations of convection problems.

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

## 20.3 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*

## 20.4 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*

## 20.5 Cahn-Hilliard Examples

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

- *examples.cahnHilliard.mesh2DCoupled*
- *examples.cahnHilliard.sphere*

## 20.6 Fluid Flow Examples

Selected illustrations of fluid flow problems.

- *examples.flow.stokesCavity*



## 20.7 Reactive Wetting Examples

Selected illustrations of multi-phase evolution.

- `examples.reactiveWetting.liquidVapor1D`

## 20.8 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 21

## 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.

### 21.1 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.

#### 21.1.1 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 22

## 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)</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)`

Test *FiPy*. Equivalent to:

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

Use

```
>>> import fipy
>>> 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', '--inline')
```

At the command line this would be:

```
$ python setup.py test --modules --trilinos --inline
```

## Modules

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

## 22.1 fipy.boundaryConditions

Boundary conditions

### Modules

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

## 22.1.1 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.

<b>Attention:</b> 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).

## 22.1.2 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).

### 22.1.3 fipy.boundaryConditions.fixedFlux

Boundary condition of order 1

#### Classes

---

<code>FixedFlux(faces, value)</code>	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)`.

### 22.1.4 fipy.boundaryConditions.fixedValue

Boundary condition of order 0

#### Classes

---

<code>FixedValue(faces, value)</code>	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  $G_{\text{face}}$  for the coefficient matrix. The parameter  $G_{\text{face}}$  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).

## 22.1.5 fipy.boundaryConditions.nthOrderBoundaryCondition

Boundary condition of specified derivative order

### Classes

<i>NthOrderBoundaryCondition</i> (faces, value, order)	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).

## 22.1.6 fipy.boundaryConditions.test

Test boundary conditions

## 22.2 fipy.matrices

Sparse matrices

### Modules

---

`fipy.matrices.offsetSparseMatrix`

`fipy.matrices.petscMatrix`

`fipy.matrices.pysparseMatrix`

`fipy.matrices.scipyMatrix`

`fipy.matrices.sparseMatrix`

`fipy.matrices.test`

`fipy.matrices.trilinosMatrix`

---

### 22.2.1 fipy.matrices.offsetSparseMatrix

#### Functions

---

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

---

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

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

### 22.2.2 fipy.matrices.petscMatrix

### 22.2.3 fipy.matrices.pysparseMatrix

### 22.2.4 fipy.matrices.scipyMatrix

### 22.2.5 fipy.matrices.sparseMatrix

### 22.2.6 fipy.matrices.test

### 22.2.7 fipy.matrices.trilinosMatrix

## 22.3 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*

## 22.3.1 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.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.
```

**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`.**22.3.2** `fipy.meshes.builders`

## Modules

*fipy.meshes.builders.abstractGridBuilder*

*fipy.meshes.builders.grid1DBuilder*

*fipy.meshes.builders.grid2DBuilder*

*fipy.meshes.builders.grid3DBuilder*

*fipy.meshes.builders.periodicGrid1DBuilder*

*fipy.meshes.builders.utilityClasses*

---

**fipy.meshes.builders.abstractGridBuilder**

**fipy.meshes.builders.grid1DBuilder**

**fipy.meshes.builders.grid2DBuilder**

**fipy.meshes.builders.grid3DBuilder**

**fipy.meshes.builders.periodicGrid1DBuilder**

**fipy.meshes.builders.utilityClasses**

### 22.3.3 fipy.meshes.cylindricalGrid1D

### 22.3.4 fipy.meshes.cylindricalGrid2D

### 22.3.5 fipy.meshes.cylindricalNonUniformGrid1D

1D Mesh

## Classes

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

```
class fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D(dx=1.0, nx=None,
                                                                              origin=(0,),
                                                                              overlap=2,
                                                                              communicator=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.
```

### 22.3.6 fipy.meshes.cylindricalNonUniformGrid2D

2D rectangular Mesh

#### Classes

<code>CylindricalNonUniformGrid2D</code> ([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,
                                                                              commu-
                                                                              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.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.

```

## 22.3.7 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, communicator=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.
```

### 22.3.8 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.
```

### 22.3.9 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*).

## 22.3.10 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>	<b>Exception</b> raised for Gmsh error conditions.
<code>MeshExportError</code>	<b>Exception</b> 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
>>> from builtins import range
>>> for refine in range(4):
...     square = Gmsh2D(geo, background=bkg)
...     x, y = square.cellCenters
...     bkg = CellVariable(mesh=square, value=abs(x / 4) + 0.01)
...     error.append(((2 * numerix.sqrt(square.cellVolumes) / bkg - 1)**2).
→cellVolumeAverage)

```

Check that the mesh is (semi)monotonically approaching the desired density (the first step may increase, depending on the number of partitions)

```
>>> print(numerix.greater(error[:-1], error[1:]).all())
True
```

and that the final density is close enough to the desired density

```
>>> print(error[-1] < 0.02)
True
```

The initial mesh doesn't have to be from Gmsh

```
>>> from fipy import Tri2D
```

```
>>> trisquare = Tri2D(nx=1, ny=1)
>>> x, y = trisquare.cellCenters
>>> bkg = CellVariable(mesh=trisquare, value=abs(x / 4) + 0.01)
>>> std1 = (numerix.sqrt(2 * trisquare.cellVolumes) / bkg).std()
```

```
>>> square = Gmsh2D(geo, background=bkg)
>>> x, y = square.cellCenters
>>> bkg = CellVariable(mesh=square, value=abs(x / 4) + 0.01)
>>> std2 = (numerix.sqrt(2 * square.cellVolumes) / bkg).std()
```

```
>>> print(std1 > std2)
True
```

### Parameters

- **arg** (*str*) – (i) the path to an *MSH* file, (ii) a path to a Gmsh geometry (*.geo*) file, or (iii) a Gmsh geometry script
- **coordDimensions** (*int*) – Dimension of shapes
- **overlap** (*int*) – The number of overlapping cells for parallel simulations. Generally 1 is adequate. Higher order equations or discretizations require more. If *overlap* is greater than one, communication reverts to serial, as Gmsh only provides one layer of ghost cells.
- **background** (*CellVariable*) – Specifies the desired characteristic lengths of the mesh cells

### 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.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.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 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.)
```

(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          ]]))
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 '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`

### 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

### 22.3.11 fipy.meshes.grid1D

### 22.3.12 fipy.meshes.grid2D

### 22.3.13 fipy.meshes.grid3D

### 22.3.14 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`.

### 22.3.15 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.
```

### 22.3.16 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.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.

```

## 22.3.17 fipy.meshes.nonUniformGrid1D

1D Mesh

### Classes

<code>NonUniformGrid1D(dx, nx, overlap, ...)</code>	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.
```

## 22.3.18 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 '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.
...      0.83333333, 0.5,      0.16666667, 0.5,
...      0.16666667, 0.5,      0.83333333,
↳ 16666667],
...
↳ [ 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]))

```

(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.
```

## 22.3.19 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.
```

## 22.3.20 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.
```

## 22.3.21 fipy.meshes.periodicGrid2D

2D periodic rectangular Mesh

### Classes

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

```

(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 '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]))

```

(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-
```

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]])
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.

```

**22.3.22 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 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****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)

```
...
True
cellCenters))
```

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.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.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.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.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.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.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.

```

**22.3.23 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****22.3.24 fipy.meshes.skewedGrid2D****Classes**

<code>SkewedGrid2D</code> ([dx, dy, nx, ny, rand])	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 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.)
```

(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.
```

### 22.3.25 fipy.meshes.sphericalNonUniformGrid1D

1D Mesh

#### Classes

---

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

---

```
class fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D(dx=1.0, nx=None,
                                                                    origin=(0,), overlap=2,
                                                                    commu-
                                                                    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-
```

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****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.
```

## 22.3.26 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.
```

### 22.3.27 fipy.meshes.test

Test implementation of the mesh

### 22.3.28 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

### 22.3.29 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-
```

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]])
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 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.
```

### 22.3.30 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.

```

**22.3.31 fipy.meshes.uniformGrid1D**

1D Mesh

**Classes**

<code>UniformGrid1D</code> ([dx, nx, origin, overlap, ...])	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.

```

**22.3.32 fipy.meshes.uniformGrid2D**

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

**Classes**

*UniformGrid2D*([dx, dy, nx, ny, origin, ...])

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 '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**

**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.
```

**22.3.33 fipy.meshes.uniformGrid3D****Classes**

<code>UniformGrid3D</code> ([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.allequal((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.
```

## 22.4 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.petsc</code>	
<code>fiPy.solvers.pyAMG</code>	
<code>fiPy.solvers.pyamgx</code>	
<code>fiPy.solvers.pysparse</code>	
<code>fiPy.solvers.pysparseMatrixSolver</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>	

**22.4.1 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.linearGMRESSolver</code>
<code>fiPy.solvers.petsc.linearLUSolver</code>
<code>fiPy.solvers.petsc.linearPCGSolver</code>
<code>fiPy.solvers.petsc.petscKrylovSolver</code>
<code>fiPy.solvers.petsc.petscSolver</code>

## fiPy solvers.petsc.comms

### Modules

```
fiPy.solvers.petsc.comms.  
parallelPETScCommWrapper  
fiPy.solvers.petsc.comms.petscCommWrapper
```

```
fiPy.solvers.petsc.comms.  
serialPETScCommWrapper
```

## 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

<code>PETScCommWrapper([petsc4py_comm])</code>	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**


---

*SerialPETScCommWrapper*()
 

---

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

 Bases: *PETScCommWrapper*
**\_\_getstate\_\_**()

Helper for pickle.

**\_\_repr\_\_**()

Return repr(self).

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


---

*DummySolver*(\*args, \*\*kwargs)
 

---

Solver that doesn't do anything.

**class** fipy.solvers.petsc.dummySolver.**DummySolver**(\*args, \*\*kwargs)

 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 error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

**\_\_repr\_\_**()

Return repr(self).

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


---

*LinearBicgSolver*([tolerance, iterations, precon])
 

---

 The *LinearBicgSolver* is an interface to the biconjugate gradient solver in PETSc, using no preconditioner by default.

```
class fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver(tolerance=1e-10, iterations=1000,
                                                         precon=None)
```

Bases: *PETScKrylovSolver*

The *LinearBicgSolver* is an interface to the biconjugate gradient solver in PETSc, using no preconditioner by default.

**Parameters**

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

`__repr__()`

Return repr(self).

### fipy.solvers.petsc.linearCGSSolver

#### Classes

<i>LinearCGSSolver</i> ([tolerance, iterations, precon])	The <i>LinearCGSSolver</i> is an interface to the conjugate gradient squared solver in PETSc, using no preconditioner by default.
--	---

```
class fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver(tolerance=1e-10, iterations=1000,
                                                         precon=None)
```

Bases: *PETScKrylovSolver*

The *LinearCGSSolver* is an interface to the conjugate gradient squared solver in PETSc, using no preconditioner by default.

**Parameters**

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

`__repr__()`

Return repr(self).

### fipy.solvers.petsc.linearGMRESSolver

#### Classes

<i>LinearGMRESSolver</i> ([tolerance, iterations, ...])	The <i>LinearGMRESSolver</i> is an interface to the GMRES solver in PETSc, using no preconditioner by default.
---	--

**class** `fiPy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver`(*tolerance=1e-10, iterations=1000, precon=None*)

Bases: *PETScKrylovSolver*

The *LinearGMRESSolver* is an interface to the GMRES solver in PETSc, using no preconditioner by default.

#### Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

`__repr__()`

Return repr(self).

## fiPy.solvers.petsc.linearLUSolver

### Classes

<i>LinearLUSolver</i> ([ <i>tolerance, iterations, precon</i> ])	The <i>LinearLUSolver</i> is an interface to the LU preconditioner in PETSc.
--	--

**class** `fiPy.solvers.petsc.linearLUSolver.LinearLUSolver`(*tolerance=1e-10, iterations=10, precon='lu'*)

Bases: *PETScSolver*

The *LinearLUSolver* is an interface to the LU preconditioner in PETSc. A direct solve is performed.

#### Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Ignored.

`__repr__()`

Return repr(self).

## fiPy.solvers.petsc.linearPCGSolver

### Classes

<i>LinearPCGSolver</i> ([ <i>tolerance, iterations, precon</i> ])	The <i>LinearPCGSolver</i> is an interface to the cg solver in PETSc, using no preconditioner by default.
---	---

**class** `fiPy.solvers.petsc.linearPCGSolver.LinearPCGSolver`(*tolerance=1e-10, iterations=1000, precon=None*)

Bases: *PETScKrylovSolver*

The *LinearPCGSolver* is an interface to the cg solver in PETSc, using no preconditioner by default.

#### Parameters

- *tolerance*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

`__repr__()`

Return repr(self).

## fiPy.solvers.petsc.petscKrylovSolver

### Classes

`PETScKrylovSolver`([tolerance, iterations, ...])

**Attention:**

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

`class fiPy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver`(*tolerance=1e-10, iterations=1000, precon=None*)

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*: The required error tolerance.
- *iterations*: The maximum number of iterative steps to perform.
- *precon*: Preconditioner to use (string).

`__repr__()`

Return repr(self).

**fipy.solvers.petsc.petscSolver****Classes***PETScSolver*(\*args, \*\*kwargs)**Attention:**

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

**class** `fipy.solvers.petsc.petscSolver.PETScSolver`(\*args, \*\*kwargs)Bases: *Solver*

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

Create a *Solver* object.**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

**\_\_repr\_\_**()

Return repr(self).

**22.4.2 fipy.solvers.pyAMG**

## Modules

`fipy.solvers.pyAMG.linearCGSSolver`

`fipy.solvers.pyAMG.linearGMRESSolver`

`fipy.solvers.pyAMG.linearGeneralSolver`

`fipy.solvers.pyAMG.linearLUSolver`

`fipy.solvers.pyAMG.linearPCGSolver`

`fipy.solvers.pyAMG.preconditioners`

---

## fipy.solvers.pyAMG.linearCGSSolver

### Classes

<code>LinearCGSSolver</code> ([tolerance, iterations, precon])	The <code>LinearCGSSolver</code> is an interface to the CGS solver in Scipy, using the PyAMG <code>SmoothedAggregationPreconditioner</code> by default.
--	---

---

**class** `fipy.solvers.pyAMG.linearCGSSolver.LinearCGSSolver`(*tolerance=1e-15, iterations=2000, precon=<fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner>*)

Bases: `LinearCGSSolver`

The `LinearCGSSolver` is an interface to the CGS solver in Scipy, using the PyAMG `SmoothedAggregationPreconditioner` by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (`SmoothedAggregationPreconditioner`, *optional*) –

`__repr__`()

Return `repr(self)`.

## fipy.solvers.pyAMG.linearGMRESSolver

### Classes

<code>LinearGMRESSolver</code> ([tolerance, iterations, ...])	The <code>LinearGMRESSolver</code> is an interface to the GMRES solver in Scipy, using the PyAMG <code>SmoothedAggregationPreconditioner</code> by default.
---	---

---



```
class fipy.solvers.pyAMG.linearGMRESSolver.LinearGMRESSolver(tolerance=1e-15, iterations=2000,
                                                             pre-
                                                             con=<fipy.solvers.pyAMG.preconditioners.smoothedAg
                                                             object>)
```

Bases: *LinearGMRESSolver*

The *LinearGMRESSolver* is an interface to the GMRES solver in Scipy, using the PyAMG *SmoothedAggregationPreconditioner* by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, *optional*) –

**\_\_repr\_\_**()

Return repr(self).

## fipy.solvers.pyAMG.linearGeneralSolver

### Classes

<i>LinearGeneralSolver</i> ([tolerance, iterations, ...])	The <i>LinearGeneralSolver</i> is an interface to the generic PyAMG, which solves the arbitrary system $Ax=b$ with the best out-of-the box choice for a solver.
---	---

```
class fipy.solvers.pyAMG.linearGeneralSolver.LinearGeneralSolver(tolerance=1e-10,
                                                                  iterations=1000,
                                                                  precon=None)
```

Bases: *\_ScipySolver*

The *LinearGeneralSolver* is an interface to the generic PyAMG, which solves the arbitrary system  $Ax=b$  with the best out-of-the box choice for a solver. See *pyAMG.solve* for details.

Create a *Solver* object.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

**\_\_repr\_\_**()

Return repr(self).

## fiPy.solvers.pyAMG.linearLUSolver

### Classes

---

<i>LinearLUSolver</i> ([tolerance, iterations, precon])	Create a <i>Solver</i> object.
---	--------------------------------

---

```
class fiPy.solvers.pyAMG.linearLUSolver.LinearLUSolver(tolerance=1e-10, iterations=1000,
                                                       precon=None)
```

Bases: *LinearLUSolver*

Create a *Solver* object.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

`__repr__()`

Return repr(self).

## fiPy.solvers.pyAMG.linearPCGSolver

### Classes

---

<i>LinearPCGSolver</i> ([tolerance, iterations, precon])	The <i>LinearPCGSolver</i> is an interface to the PCG solver in Scipy, using the PyAMG <i>SmoothedAggregationPreconditioner</i> by default.
--	---

---

```
class fiPy.solvers.pyAMG.linearPCGSolver.LinearPCGSolver(tolerance=1e-15, iterations=2000, precon=<fiPy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner object>)
```

Bases: *LinearPCGSolver*

The *LinearPCGSolver* is an interface to the PCG solver in Scipy, using the PyAMG *SmoothedAggregationPreconditioner* by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*SmoothedAggregationPreconditioner*, *optional*) –

`__repr__()`

Return repr(self).

**fipy.solvers.pyAMG.preconditioners****Modules**


---

*fipy.solvers.pyAMG.preconditioners.  
smoothedAggregationPreconditioner*

---

**fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner****Classes**


---

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

---

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

Bases: `object`

Preconditioner based on PyAMG `smoothed_aggregation_solver`

**22.4.3 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.smoothers*

---

## fipy.solvers.pyamgx.aggregationAMGSolver

### Classes

<code>AggregationAMGSolver</code> ([tolerance, ...])	The <i>AggregationAMGSolver</i> is an interface to the aggregation AMG solver in AMGX, with a Jacobi smoother by default.
--	---

```
class fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver(tolerance=1e-10,
                                                                    iterations=2000,
                                                                    precon=None,
                                                                    smoother={'max_iters': 1,
                                                                    'solver':
                                                                    'BLOCK_JACOBI'},
                                                                    **kwargs)
```

Bases: *PyAMGXSolver*

The *AggregationAMGSolver* is an interface to the aggregation AMG solver in AMGX, with a Jacobi smoother by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- **\*\*kwargs** – Other AMGX solver options

`__repr__`()

Return repr(self).

## fipy.solvers.pyamgx.classicalAMGSolver

### Classes

<code>ClassicalAMGSolver</code> ([tolerance, iterations, ...])	The <i>ClassicalAMGSolver</i> is an interface to the classical AMG solver in AMGX, with a Jacobi smoother by default.
--	---

```
class fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver(tolerance=1e-10,
                                                                    iterations=2000, precon=None,
                                                                    smoother={'max_iters': 1,
                                                                    'solver': 'BLOCK_JACOBI'},
                                                                    **kwargs)
```

Bases: *PyAMGXSolver*

The *ClassicalAMGSolver* is an interface to the classical AMG solver in AMGX, with a Jacobi smoother by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.

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

`__repr__()`

Return repr(self).

## fiPy.solvers.pyamgx.linearBiCGStabSolver

### Classes

*LinearBiCGStabSolver*([tolerance, ...])

The *LinearBiCGStabSolver* is an interface to the PBICGSTAB solver in AMGX, with a Jacobi preconditioner by default.

```
class fiPy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver(tolerance=1e-10,
                                                                    iterations=2000,
                                                                    precon={'max_iters': 1,
                                                                    'solver':
                                                                    'BLOCK_JACOBI'},
                                                                    **kwargs)
```

Bases: *PyAMGXSolver*

The *LinearBiCGStabSolver* is an interface to the PBICGSTAB solver in AMGX, with a Jacobi preconditioner by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- **\*\*kwargs** – Other AMGX solver options

`__repr__()`

Return repr(self).

## fiPy.solvers.pyamgx.linearCGSolver

### Classes

*LinearCGSolver*([tolerance, iterations, precon])

The *LinearCGSolver* is an interface to the PCG solver in AMGX, with no preconditioning by default.

*LinearPCGSolver*

alias of *LinearCGSolver*

```
class fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver(tolerance=1e-10, iterations=2000,
                                                       precon={'max_iters': 1, 'solver':
                                                         'BLOCK_JACOBI'}, **kwargs)
```

Bases: *PyAMGXSolver*

The *LinearCGSolver* is an interface to the PCG solver in AMGX, with no preconditioning by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- **\*\*kwargs** – Other AMGX solver options

```
__repr__()
```

Return repr(self).

```
fipy.solvers.pyamgx.linearCGSolver.LinearPCGSolver
```

alias of *LinearCGSolver*

## fipy.solvers.pyamgx.linearFGMRESSolver

### Classes

---

```
LinearFGMRESSolver([tolerance, iterations, ...])
```

The *LinearFGMRESSolver* is an interface to the FGMRES solver in AMGX, with a Jacobi preconditioner by default.

---

```
class fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver(tolerance=1e-10,
                                                                iterations=2000,
                                                                precon={'max_iters': 1, 'solver':
                                                                  'BLOCK_JACOBI'}, **kwargs)
```

Bases: *PyAMGXSolver*

The *LinearFGMRESSolver* is an interface to the FGMRES solver in AMGX, with a Jacobi preconditioner by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- **\*\*kwargs** – Other AMGX solver options

```
__repr__()
```

Return repr(self).

**fipy.solvers.pyamgx.linearGMRESSolver****Classes**

<code>LinearGMRESSolver</code> ([tolerance, iterations, ...])	The <i>LinearGMRESSolver</i> is an interface to the GMRES solver in AMGX, with a Jacobi preconditioner by default.
---	--

```
class fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver(tolerance=1e-10, iterations=2000,
                                                             precon={'max_iters': 1, 'solver':
                                                             'BLOCK_JACOBI'}, **kwargs)
```

Bases: *PyAMGXSolver*

The *LinearGMRESSolver* is an interface to the GMRES solver in AMGX, with a Jacobi preconditioner by default.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*, *optional*) –
- **\*\*kwargs** – Other AMGX solver options

`__repr__()`

Return repr(self).

**fipy.solvers.pyamgx.linearLUSolver**

```
class fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver(tolerance=1e-10, iterations=1000,
                                                         precon=None)
```

Bases: *\_ScipySolver*

The *LinearLUSolver* solves a linear system of equations using LU-factorization. The *LinearLUSolver* is a wrapper class for the the Scipy *scipy.sparse.linalg.splu* module.

Create a *Solver* object.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

`__repr__()`

Return repr(self).

## fipy.solvers.pyamgx.preconditioners

### Modules

`fipy.solvers.pyamgx.preconditioners.preconditioners`

---

## fipy.solvers.pyamgx.preconditioners.preconditioners

### Classes

`Preconditioner`(preconditioner\_type, \*\*kwargs)      Interface to pyamgx preconditioner configuration.

---

**class** fipy.solvers.pyamgx.preconditioners.preconditioners.**Preconditioner**(preconditioner\_type, \*\*kwargs)

Bases: `object`

Interface to pyamgx preconditioner configuration.

`__call__`(\*\*kwargs)

#### Parameters

**\*\*kwargs** – Other AMGX solver options

## fipy.solvers.pyamgx.pyAMGXSolver

### Classes

`PyAMGXSolver`(config\_dict[, tolerance, ...])

**param config\_dict**  
AMGX configuration options

---

**class** fipy.solvers.pyamgx.pyAMGXSolver.**PyAMGXSolver**(config\_dict, tolerance=1e-10, iterations=2000, precon=None, smoother=None, \*\*kwargs)

Bases: `Solver`

#### Parameters

- **config\_dict** (*dict*) – AMGX configuration options
- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (`Preconditioner`, *optional*) –
- **smoother** (`Smoother`, *optional*) –
- **\*\*kwargs** – Other AMGX solver options



`__repr__()`

Return repr(self).

## fiPy.solvers.pyamgx.smoothers

### Modules

*fiPy.solvers.pyamgx.smoothers.smoothers*

## fiPy.solvers.pyamgx.smoothers.smoothers

### Classes

*Smoother*(smoother\_type)

Interface to pyamgx smoother configuration.

**class** fiPy.solvers.pyamgx.smoothers.smoothers.**Smoother**(smoother\_type)

Bases: object

Interface to pyamgx smoother configuration.

`__call__`(\*\*kwargs)

Call self as a function.

## 22.4.4 fiPy.solvers.pysparse

### Modules

*fiPy.solvers.pysparse.linearCGSSolver*

*fiPy.solvers.pysparse.linearGMRESSolver*

*fiPy.solvers.pysparse.linearJORSolver*

*fiPy.solvers.pysparse.linearLUSolver*

*fiPy.solvers.pysparse.linearPCGSolver*

*fiPy.solvers.pysparse.preconditioners*

*fiPy.solvers.pysparse.pysparseSolver*

## fipy.solvers.pysparse.linearCGSSolver

### Classes

<i>LinearCGSSolver</i> ([precon])	The <i>LinearCGSSolver</i> solves a linear system of equations using the conjugate gradient squared method (CGS), a variant of the biconjugate gradient method (BiCG).
-----------------------------------	--

**class** fipy.solvers.pysparse.linearCGSSolver.**LinearCGSSolver**(precon=None, \*args, \*\*kwargs)

Bases: *PysparseSolver*

The *LinearCGSSolver* solves a linear system of equations using the conjugate gradient squared method (CGS), a variant of the biconjugate gradient method (BiCG). CGS solves linear systems with a general non-symmetric coefficient matrix.

The *LinearCGSSolver* is a wrapper class for the the *Pysparse itsolvers.cgs()* method.

**Parameters**

**precon** (*Preconditioner*, optional) –

**\_\_repr\_\_**()

Return repr(self).

## fipy.solvers.pysparse.linearGMRESSolver

### Classes

<i>LinearGMRESSolver</i> ([precon])	The <i>LinearGMRESSolver</i> solves a linear system of equations using the generalized minimal residual method (GMRES) with Jacobi preconditioning.
-------------------------------------	---

**class** fipy.solvers.pysparse.linearGMRESSolver.**LinearGMRESSolver**(precon=<fipy.solvers.pysparse.preconditioners.jacobi object>, \*args, \*\*kwargs)

Bases: *PysparseSolver*

The *LinearGMRESSolver* solves a linear system of equations using the generalized minimal residual method (GMRES) with Jacobi preconditioning. GMRES solves systems with a general non-symmetric coefficient matrix.

The *LinearGMRESSolver* is a wrapper class for the the *Pysparse itsolvers.gmres()* and *precon.jacobi()* methods.

**Parameters**

**precon** (*Preconditioner*, optional) –

**\_\_repr\_\_**()

Return repr(self).

**fipy.solvers.pysparse.linearJORSolver****Classes**

<code>LinearJORSolver</code> ([tolerance, iterations, ...])	The <code>LinearJORSolver</code> solves a linear system of equations using Jacobi over-relaxation.
---	--

```
class fipy.solvers.pysparse.linearJORSolver.LinearJORSolver(tolerance=1e-10, iterations=1000,
                                                         relaxation=1.0)
```

Bases: `PysparseSolver`

The `LinearJORSolver` solves a linear system of equations using Jacobi over-relaxation. This method solves systems with a general non-symmetric coefficient matrix.

The `Solver` class should not be invoked directly.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **relaxation** (*float*) – Fraction of update to apply

`__repr__`()

Return repr(self).

**fipy.solvers.pysparse.linearLUSolver****Classes**

<code>LinearLUSolver</code> ([tolerance, iterations, ...])	The <code>LinearLUSolver</code> solves a linear system of equations using LU-factorization.
--	---

```
class fipy.solvers.pysparse.linearLUSolver.LinearLUSolver(tolerance=1e-10, iterations=10,
                                                         maxIterations=10, precon=None)
```

Bases: `PysparseSolver`

The `LinearLUSolver` solves a linear system of equations using LU-factorization. This method solves systems with a general non-symmetric coefficient matrix using partial pivoting.

The `LinearLUSolver` is a wrapper class for the the `Pysparse superlu.factorize()` method.

Creates a `LinearLUSolver`.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (`Preconditioner`) – *ignored*

`__repr__`()

Return repr(self).

## fiPy solvers.pysparse.linearPCGSolver

### Classes

<code>LinearPCGSolver([precon])</code>	The <i>LinearPCGSolver</i> solves a linear system of equations using the preconditioned conjugate gradient method (PCG) with symmetric successive over-relaxation (SSOR) preconditioning by default.
--	--

**class** `fiPy.solvers.pysparse.linearPCGSolver.LinearPCGSolver`(*precon*=<`fiPy.solvers.pysparse.preconditioners.ssorPreconditioner` object>, \*args, \*\*kwargs)

Bases: *PysparseSolver*

The *LinearPCGSolver* solves a linear system of equations using the preconditioned conjugate gradient method (PCG) with symmetric successive over-relaxation (SSOR) preconditioning by default. Alternatively, Jacobi preconditioning can be specified through *precon*. The PCG method solves systems with a symmetric positive definite coefficient matrix.

The *LinearPCGSolver* is a wrapper class for the the *Pysparse* `itsolvers.pcg()` and `precon.ssor()` methods.

#### Parameters

**precon** (*Preconditioner*, optional) –

`__repr__()`

Return `repr(self)`.

## fiPy solvers.pysparse.preconditioners

### Modules

<code>fiPy.solvers.pysparse.preconditioners.jacobiPreconditioner</code>
<code>fiPy.solvers.pysparse.preconditioners.preconditioner</code>
<code>fiPy.solvers.pysparse.preconditioners.ssorPreconditioner</code>

## fiPy solvers.pysparse.preconditioners.jacobiPreconditioner

### Classes

<code>JacobiPreconditioner()</code>	Jacobi preconditioner for Pysparse.
-------------------------------------	-------------------------------------

**class** `fiPy.solvers.pysparse.preconditioners.jacobiPreconditioner.JacobiPreconditioner`

Bases: *Preconditioner*

Jacobi preconditioner for Pysparse. Really just a wrapper class for `pysparse.precon.jacobi`.

Create a *Preconditioner* object.

**fipy.solvers.pysparse.preconditioners.preconditioner****Classes**

<i>Preconditioner</i> ()	Base preconditioner class
--------------------------	---------------------------

**class** fipy.solvers.pysparse.preconditioners.preconditioner.**Preconditioner**

Bases: `object`

Base preconditioner class

<b>Attention:</b> This class is abstract. Always create one of its subclasses.
--

Create a *Preconditioner* object.

**fipy.solvers.pysparse.preconditioners.ssorPreconditioner****Classes**

<i>SsorPreconditioner</i> ()	SSOR preconditioner for Pysparse.
------------------------------	-----------------------------------

**class** fipy.solvers.pysparse.preconditioners.ssorPreconditioner.**SsorPreconditioner**

Bases: *Preconditioner*

SSOR preconditioner for Pysparse. Really just a wrapper class for *pysparse.precon.jacobi*.

Create a *Preconditioner* object.

**fipy.solvers.pysparse.pysparseSolver****Classes**

<i>PysparseSolver</i> (*args, **kwargs)	The base <i>pysparseSolver</i> class.
---	---------------------------------------

**class** fipy.solvers.pysparse.pysparseSolver.**PysparseSolver**(\*args, \*\*kwargs)

Bases: `_PysparseMatrixSolver`

The base *pysparseSolver* class.

<b>Attention:</b> This class is abstract. Always create one of its subclasses.
--

Create a *Solver* object.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.

- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

`__repr__()`

Return repr(self).

## 22.4.5 fipy.solvers.pysparseMatrixSolver

## 22.4.6 fipy.solvers.scipy

### Modules

*fipy.solvers.scipy.linearBicgstabSolver*

*fipy.solvers.scipy.linearCGSSolver*

*fipy.solvers.scipy.linearGMRESSolver*

*fipy.solvers.scipy.linearLUSolver*

*fipy.solvers.scipy.linearPCGSolver*

*fipy.solvers.scipy.scipyKrylovSolver*

*fipy.solvers.scipy.scipySolver*

### fipy.solvers.scipy.linearBicgstabSolver

#### Classes

*LinearBicgstabSolver*([tolerance, ...])

The *LinearBicgstabSolver* is an interface to the Bicgstab solver in Scipy, with no preconditioning by default.

```
class fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver(tolerance=1e-15,
                                                                    iterations=2000,
                                                                    precon=None)
```

Bases: `_ScipyKrylovSolver`

The *LinearBicgstabSolver* is an interface to the Bicgstab solver in Scipy, with no preconditioning by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__repr__()`

Return repr(self).

**fipy.solvers.scipy.linearCGSSolver****Classes**


---

<i>LinearCGSSolver</i> ([tolerance, iterations, precon])	The <i>LinearCGSSolver</i> is an interface to the CGS solver in Scipy, with no preconditioning by default.
--	--

---

```
class fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver(tolerance=1e-15, iterations=2000,
                                                    precon=None)
```

Bases: `_ScipyKrylovSolver`

The *LinearCGSSolver* is an interface to the CGS solver in Scipy, with no preconditioning by default.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

```
__repr__()
```

Return repr(self).

**fipy.solvers.scipy.linearGMRESSolver****Classes**


---

<i>LinearGMRESSolver</i> ([tolerance, iterations, ...])	The <i>LinearGMRESSolver</i> is an interface to the GMRES solver in Scipy, with no preconditioning by default.
---	--

---

```
class fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver(tolerance=1e-15, iterations=2000,
                                                    precon=None)
```

Bases: `_ScipyKrylovSolver`

The *LinearGMRESSolver* is an interface to the GMRES solver in Scipy, with no preconditioning by default.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

```
__repr__()
```

Return repr(self).

**fipy.solvers.scipy.linearLUSolver****Classes**

---

<i>LinearLUSolver</i> ([tolerance, iterations, precon])	The <i>LinearLUSolver</i> solves a linear system of equations using LU-factorization.
---	---

---

```
class fipy.solvers.scipy.linearLUSolver.LinearLUSolver(tolerance=1e-10, iterations=1000,  
precon=None)
```

Bases: `_ScipySolver`

The *LinearLUSolver* solves a linear system of equations using LU-factorization. The *LinearLUSolver* is a wrapper class for the the Scipy *scipy.sparse.linalg.splu* module.

Create a *Solver* object.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

`__repr__()`

Return `repr(self)`.

**fipy.solvers.scipy.linearPCGSolver****Classes**

---

<i>LinearPCGSolver</i> ([tolerance, iterations, precon])	The <i>LinearPCGSolver</i> is an interface to the CG solver in Scipy, with no preconditioning by default.
--	---

---

```
class fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver(tolerance=1e-15, iterations=2000,  
precon=None)
```

Bases: `_ScipyKrylovSolver`

The *LinearPCGSolver* is an interface to the CG solver in Scipy, with no preconditioning by default.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use.

`__repr__()`

Return `repr(self)`.



`fipy.solvers.scipy.scipyKrylovSolver`

`fipy.solvers.scipy.scipySolver`

## 22.4.7 `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

`Solver`([tolerance, iterations, precon])

The base *LinearXSolver* class.

### Exceptions

`IllConditionedPreconditionerWarning`(solver, ...)

`MatrixIllConditionedWarning`(solver, iter, relres)

`MaximumIterationWarning`(solver, iter, relres)

`PreconditionerNotPositiveDefiniteWarning`(...)

`PreconditionerWarning`(solver, iter, relres)

`ScalarQuantityOutOfRangeWarning`(solver, ...)

`SolverConvergenceWarning`(solver, iter, relres)

`StagnatedSolverWarning`(solver, iter, relres)

**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.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)`.

**\_\_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=1e-10, iterations=1000, precon=None)

Bases: `object`

The base *LinearXSolver* class.

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

Create a *Solver* object.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

**\_\_repr\_\_()**

Return repr(self).

**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.

### 22.4.8 fipy.solvers.test

### 22.4.9 fipy.solvers.trilinos

## Modules

---

*fiPy.solvers.trilinos.comms*

*fiPy.solvers.trilinos.linearBicgstabSolver*

*fiPy.solvers.trilinos.linearCGSSolver*

*fiPy.solvers.trilinos.linearGMRESSolver*

*fiPy.solvers.trilinos.linearLUSolver*

*fiPy.solvers.trilinos.linearPCGSolver*

*fiPy.solvers.trilinos.preconditioners*

*fiPy.solvers.trilinos.*  
*trilinosAztecOOSolver*

*fiPy.solvers.trilinos.trilinosMLTest*

*fiPy.solvers.trilinos.*  
*trilinosNonlinearSolver*

*fiPy.solvers.trilinos.trilinosSolver*

---

## **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

---

*EpetraCommWrapper()*

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())  
>>> from builtins import range  
>>> 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</code> ([tolerance, ...])	The <i>LinearBicgstabSolver</i> is an interface to the biconjugate gradient stabilized solver in Trilinos, using the <i>JacobiPreconditioner</i> by default.
--	--

```
class fiPy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver(tolerance=1e-10,
                                                                    iterations=1000, pre-
                                                                    con=<fiPy.solvers.trilinos.preconditioners.j
                                                                    object>)
```

Bases: *TrilinosAztec00Solver*

The *LinearBicgstabSolver* is an interface to the biconjugate gradient stabilized solver in Trilinos, using the *JacobiPreconditioner* by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) –

`__repr__()`  
 Return repr(self).

## fiPy.solvers.trilinos.linearCGSSolver

### Classes

<code>LinearCGSSolver</code> ([tolerance, iterations, precon])	The <i>LinearCGSSolver</i> is an interface to the CGS solver in Trilinos, using the <i>MultilevelSGSPreconditioner</i> by default.
--	--

```
class fiPy.solvers.trilinos.linearCGSSolver.LinearCGSSolver(tolerance=1e-10, iterations=1000,
                                                            pre-
                                                            con=<fiPy.solvers.trilinos.preconditioners.multilevelDD
                                                            object>)
```

Bases: *TrilinosAztec00Solver*

The *LinearCGSSolver* is an interface to the CGS solver in Trilinos, using the *MultilevelSGSPreconditioner* by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) –

`__repr__()`

Return repr(self).

## fiPy.solvers.trilinos.linearGMRESSolver

### Classes

---

*LinearGMRESSolver*([tolerance, iterations, ...])

The *LinearGMRESSolver* is an interface to the GMRES solver in Trilinos, using a the *MultilevelDDPreconditioner* by default.

---

```
class fiPy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver(tolerance=1e-10,
                                                                iterations=1000, precon=<fiPy.solvers.trilinos.preconditioners.multilevel
                                                                object>)
```

Bases: *TrilinosAztec00Solver*

The *LinearGMRESSolver* is an interface to the GMRES solver in Trilinos, using a the *MultilevelDDPreconditioner* by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) –

`__repr__()`

Return repr(self).

## fiPy.solvers.trilinos.linearLUSolver

### Classes

---

*LinearLUSolver*([tolerance, iterations, ...])

The *LinearLUSolver* is an interface to the Amesos KLU solver in Trilinos.

---

```
class fiPy.solvers.trilinos.linearLUSolver.LinearLUSolver(tolerance=1e-10, iterations=10,
                                                         precon=None, maxIterations=10)
```

Bases: *TrilinosSolver*

The *LinearLUSolver* is an interface to the Amesos KLU solver in Trilinos.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.

- **precon** – *ignored*

`__repr__()`

Return repr(self).

## fiPy.solvers.trilinos.linearPCGSolver

### Classes

*LinearPCGSolver*([tolerance, iterations, precon])

The *LinearPCGSolver* is an interface to the cg solver in Trilinos, using the *MultilevelSGSPreconditioner* by default.

```
class fiPy.solvers.trilinos.linearPCGSolver.LinearPCGSolver(tolerance=1e-10, iterations=1000,
                                                            precon=<fiPy.solvers.trilinos.preconditioners.multilevelDD
                                                            object>)
```

Bases: *TrilinosAztec00Solver*

The *LinearPCGSolver* is an interface to the cg solver in Trilinos, using the *MultilevelSGSPreconditioner* by default.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) –

`__repr__()`

Return repr(self).

## fiPy solvers.trilinos.preconditioners

### Modules

---

<code>fiPy.solvers.trilinos.preconditioners.domDecompPreconditioner</code>
<code>fiPy.solvers.trilinos.preconditioners.icPreconditioner</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.multilevelSAPreconditioner</code>
<code>fiPy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner</code>
<code>fiPy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner</code>
<code>fiPy.solvers.trilinos.preconditioners.preconditioner</code>

---

### fiPy solvers.trilinos.preconditioners.domDecompPreconditioner

#### Classes

---

<code>DomDecompPreconditioner()</code>	Domain Decomposition preconditioner for Trilinos solvers.
--	---

---

#### class

`fiPy.solvers.trilinos.preconditioners.domDecompPreconditioner.DomDecompPreconditioner`

Bases: *Preconditioner*

Domain Decomposition preconditioner for Trilinos solvers.

Create a *Preconditioner* object.

### fiPy solvers.trilinos.preconditioners.icPreconditioner

#### Classes

---

<code>ICPreconditioner()</code>	Incomplete Cholesky Preconditioner from IFFPACK for Trilinos Solvers.
---------------------------------	---

---

**class** `fipy.solvers.trilinos.preconditioners.icPreconditioner.ICPreconditioner`

Bases: *Preconditioner*

Incomplete Cholesky Preconditioner from IFPACK for Trilinos Solvers.

Create a *Preconditioner* object.

**fipy.solvers.trilinos.preconditioners.jacobiPreconditioner**

### Classes

---

*JacobiPreconditioner*()

Jacobi Preconditioner for Trilinos solvers.

---

**class** `fipy.solvers.trilinos.preconditioners.jacobiPreconditioner.JacobiPreconditioner`

Bases: *Preconditioner*

Jacobi Preconditioner for Trilinos solvers.

Create a *Preconditioner* object.

**fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner**

### Classes

---

*MultilevelDDMLPreconditioner*()

Multilevel preconditioner for Trilinos solvers.

---

**class** `fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner.MultilevelDDMLPreconditioner`

Bases: *Preconditioner*

Multilevel preconditioner for Trilinos solvers. 3-level algebraic domain decomposition.

Create a *Preconditioner* object.

**fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner**

### Classes

---

*MultilevelDDPreconditioner*()

Multilevel preconditioner for Trilinos solvers.

---

**class** `fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner.MultilevelDDPreconditioner`

Bases: *Preconditioner*

Multilevel preconditioner for Trilinos solvers. A classical smoothed aggregation-based 2-level domain decomposition.

Create a *Preconditioner* object.

## fiPy solvers.trilinos.preconditioners.multilevelNSSAPreconditioner

### Classes

<code>MultilevelNSSAPreconditioner()</code>	Energy-based minimizing smoothed aggregation suitable for highly convective non-symmetric fluid flow problems.
---	--

**class** `fiPy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner.MultilevelNSSAPreconditioner`

Bases: *Preconditioner*

Energy-based minimizing smoothed aggregation suitable for highly convective non-symmetric fluid flow problems.

Create a *Preconditioner* object.

## fiPy solvers.trilinos.preconditioners.multilevelSAPreconditioner

### Classes

<code>MultilevelSAPreconditioner()</code>	Multilevel preconditioner for Trilinos solvers suitable classical smoothed aggregation for symmetric positive definite or nearly symmetric positive definite systems.
---	---

**class** `fiPy.solvers.trilinos.preconditioners.multilevelSAPreconditioner.MultilevelSAPreconditioner`

Bases: *Preconditioner*

Multilevel preconditioner for Trilinos solvers suitable classical smoothed aggregation for symmetric positive definite or nearly symmetric positive definite systems.

Create a *Preconditioner* object.

## fiPy solvers.trilinos.preconditioners.multilevelSGSPreconditioner

### Classes

<code>MultilevelSGSPreconditioner([levels])</code>	Multilevel preconditioner for Trilinos solvers using Symmetric Gauss-Seidel smoothing.
--	--

**class** `fiPy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner.MultilevelSGSPreconditioner` (*levels*)

Bases: *Preconditioner*

Multilevel preconditioner for Trilinos solvers using Symmetric Gauss-Seidel smoothing.

Initialize the multilevel preconditioner

- *levels*: Maximum number of levels

**fiPy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner****Classes**


---

*MultilevelSolverSmootherPreconditioner*([levels] Multilevel preconditioner for Trilinos solvers using Aztec solvers as smoothers.

---

**class** `fiPy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner.MultilevelSolverSmootherPreconditioner`

Bases: *Preconditioner*

Multilevel preconditioner for Trilinos solvers using Aztec solvers as smoothers.

Initialize the multilevel preconditioner

- *levels*: Maximum number of levels

**fiPy.solvers.trilinos.preconditioners.preconditioner****Classes**


---

*Preconditioner*() The base Preconditioner class.

---

**class** `fiPy.solvers.trilinos.preconditioners.preconditioner.Preconditioner`

Bases: `object`

The base Preconditioner class.

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

Create a *Preconditioner* object.

fiPy.solvers.trilinos.trilinosAztecOOSolver

Classes

*TrilinosAztecOOSolver*([tolerance, ...])

**Attention:**

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

**class** `fiPy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver`(*tolerance=1e-10, iterations=1000, precon=<fiPy.solvers.trilinos.preconditioner object>*)

Bases: *TrilinosSolver*

**Attention:** This class is abstract, always create on of its subclasses. It provides the code to call all solvers from the Trilinos AztecOO package.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** (*Preconditioner*) –

**\_\_repr\_\_**()

Return repr(self).



**fipy.solvers.trilinos.trilinosMLTest****Classes**

<code>TrilinosMLTest</code> ([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=1e-10, 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 <http://trilinos.sandia.gov/packages/ml/documentation.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.

`__repr__`()

Return repr(self).

**fipy.solvers.trilinos.trilinosNonlinearSolver****Classes**

<code>TrilinosNonlinearSolver</code> (equation[, ...])	Create a <i>Solver</i> object.
--	--------------------------------

```
class fipy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver(equation,
                                                                           jacobian=None,
                                                                           tolerance=1e-10,
                                                                           iterations=1000,
                                                                           printingOptions=None,
                                                                           solverOptions=None,
                                                                           linearSolverOptions=None,
                                                                           lineSearchOptions=None,
                                                                           directionOptions=None,
                                                                           newtonOptions=None)
```

Bases: *TrilinosSolver*

Create a *Solver* object.

**Parameters**

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

`__repr__()`

Return repr(self).

**fipy.solvers.trilinos.trilinosSolver**

**Classes**

*TrilinosSolver*(\*args, \*\*kwargs)

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

```
class fipy.solvers.trilinos.trilinosSolver.TrilinosSolver(*args, **kwargs)
```

Bases: *Solver*

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

Create a *Solver* object.

#### Parameters

- **tolerance** (*float*) – Required error tolerance.
- **iterations** (*int*) – Maximum number of iterative steps to perform.
- **precon** – Preconditioner to use. Not all solver suites support preconditioners.

```
__repr__()
```

Return repr(self).

## 22.5 fipy.steps

(Obsolete) utilities for iterating time steps

Use [steppyngstounes](#) instead.

### Functions

*L1error*(var, matrix, RHSvector)

*L2error*(var, matrix, RHSvector)

*LINFerror*(var, matrix, RHSvector)

*error*(var, matrix, RHSvector[, norm])

*residual*(var, matrix, RHSvector)

Determines the residual for the current solution matrix and variable.

*sweepMonotonic*(fn, \*args, \*\*kwargs)

Repeatedly calls `fn(*args, **kwargs)()` until the residual returned by `fn()` is no longer decreasing.

`fipy.steps.L1error`(var, matrix, RHSvector)

$$\frac{\|\text{var} - \text{var}^{\text{old}}\|_1}{\|\text{var}^{\text{old}}\|_1}$$

where  $\|\vec{x}\|_1$  is the  $L^1$  norm of  $\vec{x}$ .

#### Parameters

- **var** (*CellVariable*) – The *CellVariable* in question.
- **matrix** – (*ignored*)
- **RHSvector** – (*ignored*)

`fipy.steppers.L2error`(*var*, *matrix*, *RHSvector*)

$$\frac{\|\text{var} - \text{var}^{\text{old}}\|_2}{\|\text{var}^{\text{old}}\|_2}$$

where  $\|\vec{x}\|_2$  is the  $L^2$  norm of  $\vec{x}$ .

**Parameters**

- **var** (*CellVariable*) – The *CellVariable* in question.
- **matrix** – (*ignored*)
- **RHSvector** – (*ignored*)

`fipy.steppers.LINFerror`(*var*, *matrix*, *RHSvector*)

$$\frac{\|\text{var} - \text{var}^{\text{old}}\|_\infty}{\|\text{var}^{\text{old}}\|_\infty}$$

where  $\|\vec{x}\|_\infty$  is the  $L^\infty$  norm of  $\vec{x}$ .

**Parameters**

- **var** (*CellVariable*) – The *CellVariable* in question.
- **matrix** – (*ignored*)
- **RHSvector** – (*ignored*)

`fipy.steppers.error`(*var*, *matrix*, *RHSvector*, *norm*=<function *LInorm*>)

$$\frac{\|\text{var} - \text{var}^{\text{old}}\|_?}{\|\text{var}^{\text{old}}\|_?}$$

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: *LInorm()*).

`fipy.steppers.residual`(*var*, *matrix*, *RHSvector*)

Determines the residual for the current solution matrix and variable.

$$\|L\vec{x} - \vec{b}\|_\infty$$

where  $\|\vec{\xi}\|_\infty$  is the  $L^\infty$  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.steps.sweepMonotonic(fn, *args, **kwargs)`

Repeatedly calls `fn(*args, **kwargs)()` until the residual returned by `fn()` is no longer decreasing.

#### Parameters

- **fn** (*function*) – The function to call
- **\*args** –
- **\*\*kwargs** –

#### Return type

float

## Modules

`fiPy.steps.pidStepper`

`fiPy.steps.pseudoRKQSStepper`

`fiPy.steps.stepper`

## 22.5.1 fiPy.steps.pidStepper

### Classes

`PIDStepper`([vardata, proportional, ...])

Adaptive stepper using a PID controller, based on.

**class** `fiPy.steps.pidStepper.PIDStepper`(vardata=(), proportional=0.075, integral=0.175, derivative=0.01)

Bases: `Stepper`

Adaptive stepper using a PID controller, based on:

```
@article{PIDpaper,
  author = {A. M. P. Valli and G. F. Carey and A. L. G. A. Coutinho},
  title = {Control strategies for timestep selection in finite element
    simulation of incompressible flows and coupled
    reaction-convection-diffusion processes},
  journal = {Int. J. Numer. Meth. Fluids},
  volume = 47,
  year = 2005,
  pages = {201-231},
}
```

## 22.5.2 fipy.steppers.pseudoRKQSStepper

### Classes

---

<i>PseudoRKQSStepper</i> ([vardata, safety, pgrow, ...])	Adaptive stepper based on the rkqs (Runge-Kutta "quality-controlled" stepper) algorithm of Numerical Recipes in C: 2nd Edition, Section 16.2.
--	---

---

```
class fipy.steppers.pseudoRKQSStepper.PseudoRKQSStepper(vardata=(), safety=0.9, pgrow=-0.2,
                                                         pshrink=-0.25, errcon=0.000189)
```

Bases: *Stepper*

Adaptive stepper based on the rkqs (Runge-Kutta "quality-controlled" stepper) algorithm of Numerical Recipes in C: 2nd Edition, Section 16.2.

Not really appropriate, since we're not doing Runge-Kutta steps in the first place, but works OK.

## 22.5.3 fipy.steppers.stepper

### Classes

---

<i>Stepper</i> ([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.

## 22.6 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).

**\_\_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.VectorCoeffError` (*s='The coefficient must be a vector value.'*)

Bases: [TypeError](#)

**\_\_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

*fiPy.terms.abstractBinaryTerm*

*fiPy.terms.abstractConvectionTerm*

*fiPy.terms.abstractDiffusionTerm*

*fiPy.terms.abstractUpwindConvectionTerm*

*fiPy.terms.advectionTerm*

*fiPy.terms.asymmetricConvectionTerm*

*fiPy.terms.binaryTerm*

*fiPy.terms.cellTerm*

*fiPy.terms.centralDiffConvectionTerm*

*fiPy.terms.coupledBinaryTerm*

*fiPy.terms.diffusionTerm*

*fiPy.terms.diffusionTermCorrection*

*fiPy.terms.diffusionTermNoCorrection*

*fiPy.terms.explicitDiffusionTerm*

*fiPy.terms.explicitSourceTerm*

*fiPy.terms.explicitUpwindConvectionTerm*

*fiPy.terms.exponentialConvectionTerm*

*fiPy.terms.faceTerm*

*fiPy.terms.firstOrderAdvectionTerm*

*fiPy.terms.hybridConvectionTerm*

*fiPy.terms.implicitDiffusionTerm*

*fiPy.terms.implicitSourceTerm*

*fiPy.terms.nonDiffusionTerm*

*fiPy.terms.powerLawConvectionTerm*

*fiPy.terms.residualTerm*

---

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>

### 22.6.1 `fiPy.terms.abstractBinaryTerm`

### 22.6.2 `fiPy.terms.abstractConvectionTerm`

### 22.6.3 `fiPy.terms.abstractDiffusionTerm`

### 22.6.4 `fiPy.terms.abstractUpwindConvectionTerm`

### 22.6.5 `fiPy.terms.advectionTerm`

#### Classes

<code>AdvectionTerm</code> ([coeff])	The <code>AdvectionTerm</code> object constructs the $b$ 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** (~*fipy.variables.cellVariable.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*

## 22.6.6 fipy.terms.asymmetricConvectionTerm

## 22.6.7 fipy.terms.binaryTerm

## 22.6.8 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** (~fipy.variables.cellVariable.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*

## 22.6.9 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  $\alpha_f$  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)

```

...
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.]]), 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  $\vec{e} = L\vec{x}_{old} - \vec{b}$

### 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** (~fipy.variables.cellVariable.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*

## 22.6.10 fipy.terms.coupledBinaryTerm

## 22.6.11 fipy.terms.diffusionTerm

### Classes

*DiffusionTerm*([coeff, var])

This term represents a higher order diffusion term. The order of the term is determined by the number of *coeffs*, 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** (*~fipy.variables.cellVariable.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

## 22.6.12 fipy.terms.diffusionTermCorrection

### Classes

<code>DiffusionTermCorrection([coeff, var])</code>	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** (*~fipy.variables.cellVariable.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*

### 22.6.13 fipy.terms.diffusionTermNoCorrection

**Classes**

<i>DiffusionTermNoCorrection</i> ([ <i>coeff, var</i> ])	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** (*~fipy.variables.cellVariable.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*

## 22.6.14 fipy.terms.explicitDiffusionTerm

### Classes

<code>ExplicitDiffusionTerm</code> ([coeff, var])	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  $\phi_A^{\text{old}}$  and  $\phi_P^{\text{old}}$  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** (~fipy.variables.cellVariable.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*

### 22.6.15 fipy.terms.explicitSourceTerm

### 22.6.16 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  $\alpha_f$  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** (*~fipy.variables.cellVariable.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*

## 22.6.17 fipy.terms.exponentialConvectionTerm

### Classes

<i>ExponentialConvectionTerm</i> ([coeff, var])	The discretization for this <i>Term</i> 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  $\alpha_f$  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** (*~fipy.variables.cellVariable.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](#)

## 22.6.18 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** (*~fipy.variables.cellVariable.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

## 22.6.19 fipy.terms.firstOrderAdvectionTerm

### Classes

<code>FirstOrderAdvectionTerm([coeff])</code>	The <code>FirstOrderAdvectionTerm</code> 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:

$$u_P|\nabla\phi|_P = \max(u_P, 0) \left[ \sum_A \min\left(\frac{\phi_A - \phi_P}{d_{AP}}, 0\right) \right]^2^{1/2} + \min(u_P, 0) \left[ \sum_A \max\left(\frac{\phi_A - \phi_P}{d_{AP}}, 0\right) \right]^2^{1/2}$$

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** (`~fipy.variables.cellVariable.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*

## 22.6.20 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  $\alpha_f$  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)

```

...
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  $\vec{e} = L\vec{x}_{old} - \vec{b}$

### 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** (~fipy.variables.cellVariable.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*

### 22.6.21 fipy.terms.implicitDiffusionTerm

`fipy.terms.implicitDiffusionTerm.ImplicitDiffusionTerm`

alias of *DiffusionTerm*

### 22.6.22 fipy.terms.implicitSourceTerm

**Classes**

<i>ImplicitSourceTerm</i> ([coeff, var])	The <i>ImplicitSourceTerm</i> 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  $\phi$  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** (*~fipy.variables.cellVariable.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*

### 22.6.23 fipy.terms.nonDiffusionTerm

### 22.6.24 fipy.terms.powerLawConvectionTerm

**Classes**

<i>PowerLawConvectionTerm</i> ([coeff, var])	The discretization for this <i>Term</i> 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  $\alpha_f$  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):
...
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,))))).
```

(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** (*~fipy.variables.cellVariable.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*

## 22.6.25 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  $\phi$  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** (*~fipy.variables.cellVariable.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*

## 22.6.26 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  $\phi$  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** (*~fipy.variables.cellVariable.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

## 22.6.27 fipy.terms.term

### Classes

*Term*([coeff, var])

**Attention:**

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

**class** fipy.terms.term.Term(*coeff=1.0, var=None*)

Bases: object

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

Create a *Term*.

**Parameters**

**coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

**property RHSvector**

Return the RHS vector calculated in *solve()* or *sweep()*. The *cacheRHSvector()* method should be called before *solve()* or *sweep()* to cache the vector.

**\_\_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** (*~fipy.variables.cellVariable.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*

## 22.6.28 fipy.terms.test

## 22.6.29 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  $\rho$  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  $\phi_0$  is the initial value.

```
>>> phi0 = 1.
>>> k = 1.
>>> dt = 1.
>>> relaxationFactor = 1.5
>>> steps = 2
>>> sweeps = 8
```

```
>>> from fipy.meshes import Grid1D
>>> mesh = Grid1D(nx = 1)
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(mesh = mesh, value = phi0, hasOld = 1)
>>> from fipy.terms.transientTerm import TransientTerm
>>> from fipy.terms.implicitSourceTerm import ImplicitSourceTerm
```

Relaxation, given by *relaxationFactor*, is required for a converged solution.

```
>>> eq = TransientTerm(var) == ImplicitSourceTerm(-relaxationFactor) \
...     + var * relaxationFactor + k
```

A number of sweeps at each time step are required to let the relaxation take effect.

```
>>> from builtins import range
>>> for step in range(steps):
...     var.updateOld()
...     for sweep in range(sweeps):
...         eq.solve(var, dt = dt)
```

Compare the final result with the analytical solution.

```
>>> from fipy.tools import numerix
>>> print(var.allclose(numerix.sqrt(k * dt * steps + phi0**2)))
1
```

Create a *Term*.

#### Parameters

**coeff** (*float* or *CellVariable* or *FaceVariable*) – Coefficient for the term. *FaceVariable* objects are only acceptable for diffusion or convection terms.

#### 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** (*~fipy.variables.cellVariable.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*

## 22.6.30 fipy.terms.unaryTerm

## 22.6.31 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  $\alpha_f$  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.)
```

(continues on next page)

(continued from previous page)

```

Traceback (most recent call last):
...
VectorCoeffError: The coefficient must be a vector value.
>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.],
↳0.,  0.,  0.,  0.],
      [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  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** (*~fipy.variables.cellVariable.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

**22.6.32 fipy.terms.vanLeerConvectionTerm****Classes**

<code>VanLeerConvectionTerm([coeff, var])</code>	Create a <code>_AbstractConvectionTerm</code> 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)
>>> 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.
```

(continues on next page)

(continued from previous page)

```

>>> m2 = Grid2D(nx=2, ny=1)
>>> cv2 = CellVariable(mesh=m2)
>>> vcv2 = CellVariable(mesh=m2, rank=1)
>>> vfv2 = FaceVariable(mesh=m2, rank=1)
>>> __ConvectionTerm(coeff=vcv2)
__ConvectionTerm(coeff=_ArithmeticCellToFaceVariable(value=array([[ 0.,  0.,  0.,  0.],
→0.,  0.,  0.,  0.],
      [ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0,
→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.,  0.,  0.]]), mesh=UniformGrid2D(dx=1.0, nx=2, dy=1.0,
→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** (*~fipy.variables.cellVariable.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*



## 22.7 fipy.testFiPy

Test suite for *FiPy* modules

## 22.8 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>	

### 22.8.1 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_CDIF
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
```

(continues on next page)

(continued from previous page)

```
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.

## 22.8.2 fipy.tests.lateImportTest

Classes to enable accumulating tests without importing them

Prevent failure to import one test from stopping execution of other tests.

## 22.8.3 fipy.tests.test

### Classes

<code>DeprecationErroringTestProgram([module, ...])</code>	<i>TestProgram</i> that overrides inability of standard <i>TestProgram</i> to throw errors on <i>DeprecationWarning</i>
<code>test(dist, **kw)</code>	Construct the command for dist, updating vars(self) with any keyword parameters.

```
class fipy.tests.test.DeprecationErroringTestProgram(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`

*TestProgram* that overrides inability of standard *TestProgram* to throw errors on *DeprecationWarning*

```
class fipy.tests.test.test(dist, **kw)
```

Bases: `test`

Construct the command for dist, updating vars(self) with any keyword parameters.

**finalize\_options()**

Set final values for all the options that this command supports. This is always called as late as possible, ie. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if ‘foo’ depends on ‘bar’, then it is safe to set ‘foo’ from ‘bar’ as long as ‘foo’ still has the same value it was assigned in ‘initialize\_options()’.

This method must be implemented by all command classes.

**initialize\_options()**

Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, ‘initialize\_options()’ implementations are just a bunch of “self.foo = None” assignments.

This method must be implemented by all command classes.

## 22.8.4 fipy.tests.testProgram

### Classes

<code>TestProgram</code> ([module, defaultTest, argv, ...])	A command-line program that runs a set of tests
<code>main</code>	alias of <code>TestProgram</code>

```
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`

## 22.9 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 Hinsen's original `PhysicalQuantity`). The *value* can be a `Numeric array`:

```
>>> a = numerix.array((3., 4.), (5., 6.))
>>> print(PhysicalField(value = a, unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or a *tuple*:

```
>>> print(PhysicalField(value = ((3., 4.), (5., 6.)), unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or as a single value to be applied to every element of a supplied array:

```
>>> print(PhysicalField(value = 2., unit = "m", array = a))
[[ 2.  2.]
 [ 2.  2.]] m
```

Every element in an array has the same unit, which is stored only once for the whole array.

`__abs__()`

Return the absolute value of the quantity. The *unit* is unchanged.

```
>>> print(abs(PhysicalField((3., -2.), (-1., 4.)), 'm'))
[[ 3.  2.]
 [ 1.  4.]] m
```

`__add__(other)`

Add two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'm'))
10.01 km
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

`__array__(t=None)`

Return a dimensionless `PhysicalField` as a `Numeric array`.

```
>>> print(numerix.array(PhysicalField((2., 3.), (4., 5.)), "m/m"))
[[ 2.  3.]
 [ 4.  5.]]
```

As a special case, fields with angular units are converted to base units (radians) and then assumed dimensionless.

```
>>> print(numerix.array(PhysicalField(((2., 3.), (4., 5.)), "deg")))
[[ 0.03490659  0.05235988]
 [ 0.06981317  0.08726646]]
```

If the array is not dimensionless, the numerical value in its base units is returned.

```
>>> numerix.array(PhysicalField(((2., 3.), (4., 5.)), "mm"))
array([[ 0.002,  0.003],
       [ 0.004,  0.005]])
```

### `__array_wrap__`(*arr*, *context=None*)

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 *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
```

**add**(*other*)

Add two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'm'))
10.01 km
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

**allclose**(*other*, *atol=None*, *rtol=1e-08*)

This function tests whether or not *self* and *other* are equal subject to the given relative and absolute tolerances. The formula used is:

```
| self - other | < atol + rtol * | other |
```

This means essentially that both elements are small compared to *atol* or their difference divided by *other*'s value is small compared to *rtol*.

**allequal**(*other*)

This function tests whether or not *self* and *other* are exactly equal.

**arccos**()

Return the inverse cosine of the *PhysicalField* in radians

```
>>> print(PhysicalField(0).arccos().allclose("1.57079632679 rad"))
1
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round(PhysicalField("1 m").arccos(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

### arccosh()

Return the inverse hyperbolic cosine of the *PhysicalField*

```
>>> print(numerix.allclose(PhysicalField(2).arccosh(),
...                        1.31695789692))
1
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round(PhysicalField("1. m").arccosh(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

### arcsin()

Return the inverse sine of the *PhysicalField* in radians

```
>>> print(PhysicalField(1).arcsin().allclose("1.57079632679 rad"))
1
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round(PhysicalField("1 m").arcsin(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

### arctan()

Return the arctangent of the *PhysicalField* in radians

```
>>> print(numerix.round(PhysicalField(1).arctan(), 6))
0.785398
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round(PhysicalField("1 m").arctan(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

### arctan2(*other*)

Return the arctangent of *self* divided by *other* in radians

```
>>> print(numerix.round(PhysicalField(2.).arctan2(PhysicalField(5.)), 6))
0.380506
```

The input *PhysicalField* objects must be in the same dimensions

```
>>> print(numerix.round(PhysicalField(2.54, "cm").arctan2(PhysicalField(1.,
↪ "inch")), 6))
0.785398
```

```
>>> print(numerix.round(PhysicalField(2.).arctan2(PhysicalField("5. m")), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

### arctanh()

Return the inverse hyperbolic tangent of the *PhysicalField*

```
>>> print(PhysicalField(0.5).arctanh())
0.549306144334
```

The input *PhysicalField* must be dimensionless

```
>>> print(numerix.round(PhysicalField("1 m").arctanh(), 6))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

### ceil()

Return the smallest integer greater than or equal to the *PhysicalField*.

```
>>> print(PhysicalField(2.2, "m").ceil())
3.0 m
```

### conjugate()

Return the complex conjugate of the *PhysicalField*.

```
>>> print(PhysicalField(2.2 - 3j, "ohm").conjugate() == PhysicalField(2.2 + 3j,
↪ "ohm"))
True
```

### convertToUnit(*unit*)

Changes the unit to *unit* and adjusts the value such that the combination is equivalent. The new unit is by a string containing its name. The new unit must be compatible with the previous unit of the object.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> e.convertToUnit('kcal/mol')
>>> print(e)
1694.27557621 kcal/mol
```

### copy()

Make a duplicate.

```
>>> a = PhysicalField(1, unit = 'inch')
>>> b = a.copy()
```

The duplicate will not reflect changes made to the original

```
>>> a.convertToUnit('cm')
>>> print(a)
2.54 cm
>>> print(b)
1 inch
```

Likewise for arrays

```
>>> a = PhysicalField(numerix.array((0, 1, 2)), unit = 'm')
>>> b = a.copy()
>>> a[0] = 3
>>> print(a)
[3 1 2] m
>>> print(b)
[0 1 2] m
```

### cos()

Return the cosine of the *PhysicalField*

```
>>> print(numerix.round(PhysicalField(2*numerix.pi/6, "rad").cos(), 6))
0.5
>>> print(numerix.round(PhysicalField(60., "deg").cos(), 6))
0.5
```

The units of the *PhysicalField* must be an angle

```
>>> PhysicalField(60., "m").cos()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

### cosh()

Return the hyperbolic cosine of the *PhysicalField*

```
>>> PhysicalField(0.).cosh()
1.0
```

The units of the *PhysicalField* must be dimensionless

```
>>> PhysicalField(60., "m").cosh()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

### divide(*other*)

Divide two physical quantities. The unit of the result is the unit of the first operand divided by the unit of the second.

```
>>> print(PhysicalField(10., 'm') / PhysicalField(2., 's'))
5.0 m/s
```

As a special case, if the result is dimensionless, the value is returned without units, rather than with a dimensionless unit of *1*. This facilitates passing physical quantities to packages such as `Numeric` that cannot use units, while ensuring the quantities have the desired units

```
>>> print((PhysicalField(1., 'inch')
...       / PhysicalField(1., 'mm'))
25.4
```

### `dot(other)`

Return the dot product of *self* with *other*. The resulting unit is the product of the units of *self* and *other*.

```
>>> v = PhysicalField(((5., 6.), (7., 8.)), "m")
>>> print(PhysicalField(((1., 2.), (3., 4.)), "m").dot(v))
[ 26.  44.] m**2
```

### `floor()`

Return the largest integer less than or equal to the *PhysicalField*.

```
>>> print(PhysicalField(2.2, "m").floor())
2.0 m
```

### `getsctype(default=None)`

Returns the NumPy sctype of the underlying array.

```
>>> PhysicalField(1, 'm').getsctype() == numerix.NUMERIX.obj2sctype(numerix.
└─array(1))
True
>>> PhysicalField(1., 'm').getsctype() == numerix.NUMERIX.obj2sctype(numerix.
└─array(1.))
True
>>> PhysicalField((1, 1.), 'm').getsctype() == numerix.NUMERIX.
└─obj2sctype(numerix.array((1., 1.)))
True
```

### `inBaseUnits()`

Return the quantity with all units reduced to their base SI elements.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

### `inDimensionless()`

Returns the numerical value of a dimensionless quantity.

```
>>> print(PhysicalField(((2., 3.), (4., 5.))).inDimensionless())
[[ 2.  3.]
 [ 4.  5.]]
```

It's an error to convert a quantity with units

```
>>> print(PhysicalField((2., 3.), (4., 5.)), "m").inDimensionless()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

**inRadians()**

Converts an angular quantity to radians and returns the numerical value.

```
>>> print(PhysicalField((2., 3.), (4., 5.)), "rad").inRadians()
[[ 2.  3.]
 [ 4.  5.]]
>>> print(PhysicalField((2., 3.), (4., 5.)), "deg").inRadians()
[[ 0.03490659  0.05235988]
 [ 0.06981317  0.08726646]]
```

As a special case, assumes a dimensionless quantity is already in radians.

```
>>> print(PhysicalField((2., 3.), (4., 5.)).inRadians())
[[ 2.  3.]
 [ 4.  5.]]
```

It's an error to convert a quantity with non-angular units

```
>>> print(PhysicalField((2., 3.), (4., 5.)), "m").inRadians()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

**inSIUnits()**

Return the quantity with all units reduced to SI-compatible elements.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print(e.inSIUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

**inUnitsOf(\*units)**

Returns one or more *PhysicalField* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *PhysicalField*.

```
>>> freeze = PhysicalField('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *PhysicalField* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = PhysicalField(314159., 's')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h
↵', 'min', 's'),
```

(continues on next page)

(continued from previous page)

```

...
↳ '15.0 min', '59.0 s']]),
...
... True))
1

```

**itemset(value)**

Assign the value of a scalar array, performing appropriate conversions.

```

>>> a = PhysicalField(4., "m")
>>> a.itemset(PhysicalField("6 ft"))
>>> print(a.allclose("1.8288 m"))
1
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> try:
...     a.itemset(PhysicalField("6 ft"))
... except IndexError:
...     # NumPy 1.7 has changed the exception type
...     raise ValueError("can only place a scalar for an array of size 1")
Traceback (most recent call last):
...
ValueError: can only convert an array of size 1 to a Python scalar
>>> a.itemset(PhysicalField("2 min"))
Traceback (most recent call last):
...
TypeError: Incompatible units

```

**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.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
<code>fiPy.tools.version</code>	Shim for version checking

**22.9.1 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)`.

## 22.9.2 fiPy.tools.debug

### Functions

---

*PRINT*(*label*[, *arg*, *stall*])Display *label* and *arg* 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.

## 22.9.3 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 docstring after the warning.

#### Returns

**old\_func** – The deprecated function.

#### Return type

function

## 22.9.4 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 Hinsen’s `PhysicalQuantity` <<http://dirac.cnrs-orleans.fr/ScientificPython/>>.

This module provides a data type that represents a physical quantity together with its unit. It is possible to add and subtract these quantities if the units are compatible, and a quantity can be converted to another compatible unit. Multiplication, subtraction, and raising to integer powers is allowed without restriction, and the result will have the correct

unit. A quantity can be raised to a non-integer power only if the result can be represented by integer powers of the base units.

The values of physical constants are taken from the 2002 recommended values from CODATA. Other conversion factors (e.g. for British units) come from Appendix B of NIST Special Publication 811.

**Warning:** We can't guarantee for the correctness of all entries in the unit table, so use this at your own risk!

Base SI units:

m, kg, s, A, K, mol, cd, rad, sr

SI prefixes:

```

Y = 1e+24
Z = 1e+21
E = 1e+18
P = 1e+15
T = 1e+12
G = 1e+09
M = 1e+06
k = 1000
h = 100
da = 10
d = 0.1
c = 0.01
m = 0.001
mu = 1e-06
n = 1e-09
p = 1e-12
f = 1e-15
a = 1e-18
z = 1e-21
y = 1e-24
    
```

Units derived from SI (accepting SI prefixes):

```

1 Bq = 1 1/s
1 C = 1 s*A
1 degC = 1 K
1 F = 1 s**4*A**2/m**2/kg
1 Gy = 1 m**2/s**2
1 H = 1 m**2*kg/s**2/A**2
1 Hz = 1 1/s
1 J = 1 m**2*kg/s**2
1 lm = 1 cd*sr
1 lx = 1 cd*sr/m**2
1 N = 1 m*kg/s**2
1 ohm = 1 m**2*kg/s**3/A**2
1 Pa = 1 kg/m/s**2
1 S = 1 s**3*A**2/m**2/kg
1 Sv = 1 m**2/s**2
1 T = 1 kg/s**2/A
    
```

(continues on next page)



(continued from previous page)

```

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**2*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 = 0.004546090000000002 m**3
1 galUS = 0.0037854120960000006 m**3
1 gn = 9.80665 m/s**2
1 Grav = 6.6742e-11 m**3/kg/s**2
1 h = 3600.0 s
1 ha = 10000.0 m**2
1 Hartree = 4.35974417680088e-18 m**2*kg/s**2
1 hbar = 1.0545716823644548e-34 m**2*kg/s
1 hpEl = 746.0 m**2*kg/s**3
1 hplanck = 6.6260693e-34 m**2*kg/s
1 hpUK = 745.7 m**2*kg/s**3
1 inch = 0.025400000000000002 m
1 invcm = 1.9864456023253395e-23 m**2*kg/s**2
1 kB = 1.3806505e-23 m**2*kg/s**2/K
1 kcal = 4184.0 m**2*kg/s**2
1 kcali = 4186.8 m**2*kg/s**2
1 Ken = 1.3806505e-23 m**2*kg/s**2

```

(continues on next page)

(continued from previous page)

```

1 l = 0.00100000000000000002 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 = 1.00000000000000002e-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 Hinsen's original `PhysicalQuantity`). The *value* can be a `Numeric array`:

```
>>> a = numerix.array((3., 4.), (5., 6.))
>>> print(PhysicalField(value = a, unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or a *tuple*:

```
>>> print(PhysicalField(value = ((3., 4.), (5., 6.)), unit = "m"))
[[ 3.  4.]
 [ 5.  6.]] m
```

or as a single value to be applied to every element of a supplied array:

```
>>> print(PhysicalField(value = 2., unit = "m", array = a))
[[ 2.  2.]
 [ 2.  2.]] m
```

Every element in an array has the same unit, which is stored only once for the whole array.

`__abs__()`

Return the absolute value of the quantity. The *unit* is unchanged.

```
>>> print(abs(PhysicalField((3., -2.), (-1., 4.)), 'm'))
[[ 3.  2.]
 [ 1.  4.]] m
```

`__add__(other)`

Add two physical quantities, so long as their units are compatible. The unit of the result is the unit of the first operand.

```
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'm'))
10.01 km
>>> print(PhysicalField(10., 'km') + PhysicalField(10., 'J'))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

`__array__(t=None)`

Return a dimensionless `PhysicalField` as a `Numeric array`.

```
>>> print(numerix.array(PhysicalField((2., 3.), (4., 5.)), "m/m"))
[[ 2.  3.]
 [ 4.  5.]]
```

As a special case, fields with angular units are converted to base units (radians) and then assumed dimensionless.

```
>>> print(numerix.array(PhysicalField(((2., 3.), (4., 5.)), "deg")))
[[ 0.03490659  0.05235988]
 [ 0.06981317  0.08726646]]
```

If the array is not dimensionless, the numerical value in its base units is returned.

```
>>> numerix.array(PhysicalField(((2., 3.), (4., 5.)), "mm"))
array([[ 0.002,  0.003],
       [ 0.004,  0.005]])
```

### `__array_wrap__`(*arr*, *context=None*)

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 *I*. This facilitates passing physical quantities to packages such as `Numeric` that cannot use units, while ensuring the quantities have the desired units

```
>>> print((PhysicalField(1., 'inch')
...       / PhysicalField(1., 'mm')))
25.4
```

### `__eq__`(*other*)

Return `self==value`.

### `__float__`()

Return a dimensionless *PhysicalField* quantity as a float.

```
>>> float(PhysicalField("2. m/m"))
2.0
```

As a special case, quantities with angular units are converted to base units (radians) and then assumed dimensionless.

```
>>> print(numerix.round(float(PhysicalField("2. deg")), 6))
0.034907
```

If the quantity is not dimensionless, the conversion fails.

```
>>> float(PhysicalField("2. m"))
Traceback (most recent call last):
...
TypeError: Not possible to convert a PhysicalField with dimensions to float
```

Just as a *Numeric array* cannot be cast to float, neither can *PhysicalField* arrays

```
>>> float(PhysicalField(((2., 3.), (4., 5.)), "m/m"))
Traceback (most recent call last):
...
TypeError: only ...-1 arrays can be converted to Python scalars
```

`__ge__(other)`

Return self >= value.

`__getitem__(index)`

Return the specified element of the array. The unit of the result will be the unit of the array.

```
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> print(a[1, 1])
6.0 m
```

`__gt__(other)`

Compare *self* to *other*, returning an array of Boolean values corresponding to the test against each element.

```
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> print(numerix.allclose(a > PhysicalField("13 ft"),
...                        [[False, True], [ True, True]]))
True
```

Appropriately formatted dimensional quantity strings can also be compared.

```
>>> print(numerix.allclose(a > "13 ft",
...                        [[False, True], [ True, True]]))
True
```

Arrays are compared element to element

```
>>> print(numerix.allclose(a > PhysicalField(((3., 13.), (17., 6.)), "ft"),
...                        [[ True, True], [False, True]]))
True
```

Units must be compatible

```
>>> print(a > PhysicalField("1 lb"))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

And so must array dimensions

```
>>> print(a > PhysicalField(((3., 13., 4.), (17., 6., 2.)), "ft"))
Traceback (most recent call last):
...
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

**\_\_hash\_\_()**

Return hash(self).

**\_\_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 *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
```

**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  $1$ . This facilitates passing physical quantities to packages such as `Numeric` that cannot use units, while ensuring the quantities have the desired units

```
>>> print((PhysicalField(1., 'inch')
...       / PhysicalField(1., 'mm'))
25.4
```

### `dot(other)`

Return the dot product of *self* with *other*. The resulting unit is the product of the units of *self* and *other*.

```
>>> v = PhysicalField(((5., 6.), (7., 8.)), "m")
>>> print(PhysicalField(((1., 2.), (3., 4.)), "m").dot(v))
[ 26.  44.] m**2
```

### `floor()`

Return the largest integer less than or equal to the *PhysicalField*.

```
>>> print(PhysicalField(2.2, "m").floor())
2.0 m
```

### `getsctype(default=None)`

Returns the NumPy sctype of the underlying array.

```
>>> PhysicalField(1, 'm').getsctype() == numerix.NUMERIX.obj2sctype(numerix.
└─array(1))
True
>>> PhysicalField(1., 'm').getsctype() == numerix.NUMERIX.obj2sctype(numerix.
└─array(1.))
True
>>> PhysicalField((1, 1.), 'm').getsctype() == numerix.NUMERIX.
└─obj2sctype(numerix.array((1., 1.)))
True
```

### `inBaseUnits()`

Return the quantity with all units reduced to their base SI elements.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

### `inDimensionless()`

Returns the numerical value of a dimensionless quantity.

```
>>> print(PhysicalField(((2., 3.), (4., 5.))).inDimensionless())
[[ 2.  3.]
 [ 4.  5.]
```

It's an error to convert a quantity with units

```
>>> print(PhysicalField((2., 3.), (4., 5.)), "m").inDimensionless()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

**inRadians()**

Converts an angular quantity to radians and returns the numerical value.

```
>>> print(PhysicalField((2., 3.), (4., 5.)), "rad").inRadians()
[[ 2.  3.]
 [ 4.  5.]]
>>> print(PhysicalField((2., 3.), (4., 5.)), "deg").inRadians()
[[ 0.03490659  0.05235988]
 [ 0.06981317  0.08726646]]
```

As a special case, assumes a dimensionless quantity is already in radians.

```
>>> print(PhysicalField((2., 3.), (4., 5.)).inRadians())
[[ 2.  3.]
 [ 4.  5.]]
```

It's an error to convert a quantity with non-angular units

```
>>> print(PhysicalField((2., 3.), (4., 5.)), "m").inRadians()
Traceback (most recent call last):
...
TypeError: Incompatible units
```

**inSIUnits()**

Return the quantity with all units reduced to SI-compatible elements.

```
>>> e = PhysicalField('2.7 Hartree*Nav')
>>> print(e.inSIUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

**inUnitsOf(\*units)**

Returns one or more *PhysicalField* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *PhysicalField*.

```
>>> freeze = PhysicalField('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *PhysicalField* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = PhysicalField(314159., 's')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h
↵', 'min', 's'),
```

(continues on next page)

(continued from previous page)

```

...
↪ '15.0 min', '59.0 s']]),
...
... True))
1

```

**itemset(value)**

Assign the value of a scalar array, performing appropriate conversions.

```

>>> a = PhysicalField(4., "m")
>>> a.itemset(PhysicalField("6 ft"))
>>> print(a.allclose("1.8288 m"))
1
>>> a = PhysicalField(((3., 4.), (5., 6.)), "m")
>>> try:
...     a.itemset(PhysicalField("6 ft"))
... except IndexError:
...     # NumPy 1.7 has changed the exception type
...     raise ValueError("can only place a scalar for an array of size 1")
Traceback (most recent call last):
...
ValueError: can only convert an array of size 1 to a Python scalar
>>> a.itemset(PhysicalField("2 min"))
Traceback (most recent call last):
...
TypeError: Incompatible units

```

**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(meritix.round(PhysicalField("1 inch").numericValue, 6))
0.0254
```

**put**(*indices*, *values*)

*put* is the opposite of *take*. The values of *self* at the locations specified in *indices* are set to the corresponding value of *values*.

The *indices* can be any integer sequence object with values suitable for indexing into the flat form of *self*. The *values* must be any sequence of values that can be converted to the typecode of *self*.

```
>>> f = PhysicalField((1., 2., 3.), "m")
>>> f.put((2, 0), PhysicalField((2., 3.), "inch"))
>>> print(f)
[ 0.0762  2.      0.0508] m
```

The units of *values* must be compatible with *self*.

```
>>> f.put(1, PhysicalField(3, "kg"))
Traceback (most recent call last):
...
TypeError: Incompatible units
```

**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")
>>> 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>
```

(continues on next page)

(continued from previous page)

```
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.unit * d.unit == _findUnit('Hz*s')
True
>>> c.unit * d.inBaseUnits().unit
<PhysicalUnit 1>
```

or multiply units by numbers

```
>>> a.unit * 3.
<PhysicalUnit m*3.0>
```

Units must have zero offset to be multiplied

```
>>> e = PhysicalField("1. kB")
>>> f = PhysicalField("25. degC")
>>> e.unit * f.unit
Traceback (most recent call last):
...
TypeError: cannot multiply units with non-zero offset
>>> e.unit * f.inBaseUnits().unit
<PhysicalUnit kB*K>
```

**\_\_ne\_\_**(*other*)

Return self!=value.

**\_\_pow\_\_**(*other*)

Raise a unit to an integer power

```
>>> a = PhysicalField("1. m")
>>> a.unit**2
<PhysicalUnit m**2>
>>> a.unit**-2
<PhysicalUnit 1/m**2>
```

Non-integer powers are not supported

```
>>> a.unit**0.5
Traceback (most recent call last):
...
TypeError: Illegal exponent
```

Units must have zero offset to be exponentiated

```
>>> b = PhysicalField("25. degC")
>>> b.unit**2
Traceback (most recent call last):
...
TypeError: cannot exponentiate units with non-zero offset
>>> b.inBaseUnits().unit**2
<PhysicalUnit K**2>
```

**\_\_rdiv\_\_**(*other*)

Divide something by a unit

```
>>> a = PhysicalField("1. m")
>>> 3. / a.unit
<PhysicalUnit 3.0/m>
```

Units must have zero offset to be divided

```
>>> b = PhysicalField("25. degC")
>>> 3. / b.unit
Traceback (most recent call last):
...
TypeError: cannot divide units with non-zero offset
>>> 3. / b.inBaseUnits().unit
<PhysicalUnit 3.0/K>
```

### `__repr__()`

Return representation of a physical unit

```
>>> PhysicalUnit('m', 1., [1, 0, 0, 0, 0, 0, 0, 0, 0])
<PhysicalUnit m>
```

### `__rmul__(other)`

Multiply units together

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.unit * b.unit == _findUnit('ft*m')
True
>>> a.unit * b.inBaseUnits().unit
<PhysicalUnit m**2>
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.unit * d.unit == _findUnit('Hz*s')
True
>>> c.unit * d.inBaseUnits().unit
<PhysicalUnit 1>
```

or multiply units by numbers

```
>>> a.unit * 3.
<PhysicalUnit m*3.0>
```

Units must have zero offset to be multiplied

```
>>> e = PhysicalField("1. kB")
>>> f = PhysicalField("25. degC")
>>> e.unit * f.unit
Traceback (most recent call last):
...
TypeError: cannot multiply units with non-zero offset
>>> e.unit * f.inBaseUnits().unit
<PhysicalUnit kB*K>
```

### `__rtruediv__(other)`

Divide something by a unit

```
>>> a = PhysicalField("1. m")
>>> 3. / a.unit
<PhysicalUnit 3.0/m>
```

Units must have zero offset to be divided

```
>>> b = PhysicalField("25. degC")
>>> 3. / b.unit
Traceback (most recent call last):
...
TypeError: cannot divide units with non-zero offset
>>> 3. / b.inBaseUnits().unit
<PhysicalUnit 3.0/K>
```

### `__str__()`

Return representation of a physical unit

```
>>> PhysicalUnit('m', 1., [1, 0, 0, 0, 0, 0, 0, 0, 0])
<PhysicalUnit m>
```

### `__truediv__(other)`

Divide one unit by another

```
>>> a = PhysicalField("1. m")
>>> b = PhysicalField("3. ft")
>>> a.unit / b.unit
<PhysicalUnit m/ft>
>>> a.unit / b.inBaseUnits().unit
<PhysicalUnit 1>
>>> c = PhysicalField("1. s")
>>> d = PhysicalField("3. Hz")
>>> c.unit / d.unit
<PhysicalUnit s/Hz>
>>> c.unit / d.inBaseUnits().unit
<PhysicalUnit s**2/1>
```

or divide units by numbers

```
>>> a.unit / 3.
<PhysicalUnit m/3.0>
```

Units must have zero offset to be divided

```
>>> e = PhysicalField("1. J")
>>> f = PhysicalField("25. degC")
>>> e.unit / f.unit
Traceback (most recent call last):
...
TypeError: cannot divide units with non-zero offset
>>> e.unit / f.inBaseUnits().unit
<PhysicalUnit J/K>
```

### `conversionFactorTo(other)`

Return the multiplication factor between two physical units

```
>>> a = PhysicalField("1. mm")
>>> b = PhysicalField("1. inch")
>>> print(numerix.round(b.unit.conversionFactorTo(a.unit), 6))
25.4
```

Units must have the same fundamental SI units

```
>>> c = PhysicalField("1. K")
>>> c.unit.conversionFactorTo(a.unit)
Traceback (most recent call last):
...
TypeError: Incompatible units
```

If units have different offsets, they must have the same factor

```
>>> d = PhysicalField("1. degC")
>>> c.unit.conversionFactorTo(d.unit)
1.0
>>> e = PhysicalField("1. degF")
>>> c.unit.conversionFactorTo(e.unit)
Traceback (most recent call last):
...
TypeError: Unit conversion (K to degF) cannot be expressed as a simple_
↳multiplicative factor
```

#### **conversionTupleTo(*other*)**

Return a *tuple* of the multiplication factor and offset between two physical units

```
>>> a = PhysicalField("1. K").unit
>>> b = PhysicalField("1. degF").unit
>>> from builtins import str
>>> [str(numerix.round(element, 6)) for element in b.conversionTupleTo(a)]
['0.555556', '459.67']
```

#### **isAngle()**

Returns *True* if the unit is an angle

```
>>> PhysicalField("1. deg").unit.isAngle()
1
>>> PhysicalField("1. rad").unit.isAngle()
1
>>> PhysicalField("1. inch").unit.isAngle()
0
```

#### **isCompatible(*other*)**

Returns a list of which fundamental SI units are compatible between *self* and *other*

```
>>> a = PhysicalField("1. mm")
>>> b = PhysicalField("1. inch")
>>> print(numerix.allclose(a.unit.isCompatible(b.unit),
...                          [True, True, True, True, True, True, True, True,
↳True]))
True
```

(continues on next page)

(continued from previous page)

```
>>> 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')
```

(continues on next page)



(continued from previous page)

```
>>> a
<PhysicalUnit meterpersecond>
```

## 22.9.5 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

## 22.9.6 fipy.tools.inline

## 22.9.7 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

## 22.9.8 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  $L^\infty$  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.allequal(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=10000000.0)`

find the indices of *data* that are closest to *points*

```
>>> from fipy import *
>>> m0 = Grid2D(dx=(.1, 1., 10.), dy=(.1, 1., 10.))
>>> m1 = Grid2D(nx=2, ny=2, dx=5., dy=5.)
>>> print(nearest(m0.cellCenters.globalValue, m1.cellCenters.globalValue))
[4 5 7 8]
>>> print(nearest(m0.cellCenters.globalValue, m1.cellCenters.globalValue, max_
↪ mem=100))
[4 5 7 8]
>>> print(nearest(m0.cellCenters.globalValue, m1.cellCenters.globalValue, max_
↪ mem=10000))
[4 5 7 8]
```

`fipy.tools.numerix.put(arr, ids, values)`

The opposite of *take*. The values of *arr* at the locations specified by *ids* are set to the corresponding value of *values*.

The following is to test improvements to puts with masked arrays. Places in the code were assuming incorrect put behavior.

```
>>> maskValue = 999999
```

```
>>> arr = zeros(3, '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*

**22.9.9 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

## 22.9.10 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

<code>SharedTemporaryFile([mode, buffering, ...])</code>	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`

## 22.9.11 fipy.tools.test

## 22.9.12 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 for Python 2.7

**property elapsed**

Time measured so far

## 22.9.13 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.

### 22.9.14 fipy.tools.version

Shim for version checking

`distutils.version` is deprecated, but `packaging.version` is unavailable in Python 2.7

## 22.10 fipy.variables

Collections of values supporting lazy evaluation

### Modules

`fipy.variables.addOverFacesVariable`

`fipy.variables.arithmeticCellToFaceVariable`

`fipy.variables.betaNoiseVariable`

`fipy.variables.binaryOperatorVariable`

`fipy.variables.cellToFaceVariable`

`fipy.variables.cellVariable`

`fipy.variables.constant`

continues on next page

Table 3 – continued from previous page

---

<i>fiPy.variables.constraintMask</i>
<i>fiPy.variables.coupledCellVariable</i>
<i>fiPy.variables.distanceVariable</i>
<i>fiPy.variables.exponentialNoiseVariable</i>
<i>fiPy.variables.faceGradContributionsVariable</i>
<i>fiPy.variables.faceGradVariable</i>
<i>fiPy.variables.faceVariable</i>
<i>fiPy.variables.gammaNoiseVariable</i>
<i>fiPy.variables.gaussCellGradVariable</i>
<i>fiPy.variables.gaussianNoiseVariable</i>
<i>fiPy.variables.harmonicCellToFaceVariable</i>
<i>fiPy.variables.histogramVariable</i>
<i>fiPy.variables.interfaceAreaVariable</i>
<i>fiPy.variables.interfaceFlagVariable</i>
<i>fiPy.variables.leastSquaresCellGradVariable</i>
<i>fiPy.variables.levelSetDiffusionVariable</i>
<i>fiPy.variables.meshVariable</i>
<i>fiPy.variables.minmodCellToFaceVariable</i>
<i>fiPy.variables.modCellGradVariable</i>
<i>fiPy.variables.modCellToFaceVariable</i>
<i>fiPy.variables.modFaceGradVariable</i>
<i>fiPy.variables.modPhysicalField</i>
<i>fiPy.variables.modularVariable</i>
<i>fiPy.variables.noiseVariable</i>
<i>fiPy.variables.operatorVariable</i>
<i>fiPy.variables.scharfetterGummelFaceVariable</i>

---

continues on next page

Table 3 – continued from previous page

<code>fiPy.variables.surfactantConvectionVariable</code>	
<code>fiPy.variables.surfactantVariable</code>	
<code>fiPy.variables.test</code>	Test numeric implementation of the mesh
<code>fiPy.variables.unaryOperatorVariable</code>	
<code>fiPy.variables.uniformNoiseVariable</code>	
<code>fiPy.variables.variable</code>	

### 22.10.1 `fiPy.variables.addOverFacesVariable`

### 22.10.2 `fiPy.variables.arithmeticCellToFaceVariable`

### 22.10.3 `fiPy.variables.betaNoiseVariable`

#### Classes

<code>BetaNoiseVariable(*args, **kwargs)</code>	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  $\alpha$ , a rate parameter  $\beta$ , 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

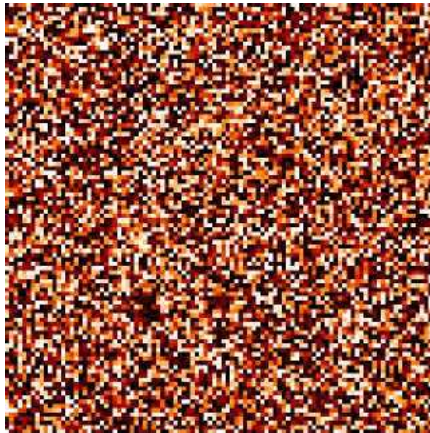
```
>>> from fipy.variables.cellVariable import CellVariable
>>> betadist = CellVariable(mesh = histogram.mesh)
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> betadist = ((Gamma(alpha + beta) / (Gamma(alpha) * Gamma(beta)))
...             * x**(alpha - 1) * (1 - x)**(beta - 1))
```

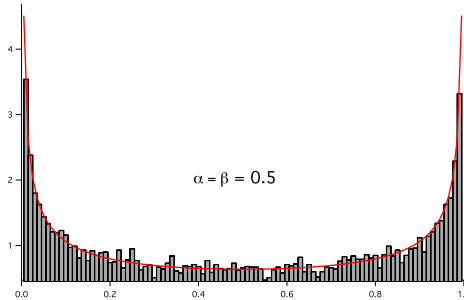
```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=1)
...     histoplot = Viewer(vars=(histogram, betadist),
...                          datamin=0, datamax=1.5)
```

```
>>> 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  $\alpha$ .
- **beta** (*float*) – The parameter  $\beta$ .

### `__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__(t=None)`

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

`__array_wrap__` (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples of ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0])))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

`__bool__` ()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↳ Use a.any() or a.all()
```

`__call__` (*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, \*\*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
```

### 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**property faceGrad**

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

**property faceGradAverage**

Deprecated since version 3.3: use `grad.arithmeticFaceValue` instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

**property faceValue**

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

**property gaussGrad**

Return  $\frac{1}{V_P} \sum_f \vec{n}_f \phi_f A_f$  as a rank-1 *CellVariable* (first-order gradient).

**getsctype**(*default=None*)

Returns the Numpy *sctype* of the underlying array.

```

>>> Variable(1).getscdtype() == numerix.NUMERIX.obj2scdtype(numerix.array(1))
True
>>> Variable(1.).getscdtype() == numerix.NUMERIX.obj2scdtype(numerix.array(1.))
True
>>> Variable((1, 1.)).getscdtype() == numerix.NUMERIX.obj2scdtype(numerix.
↳array((1., 1.)))
True

```

**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')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h',
↳ 'min', 's'),
...                                     ['3.0 d', '15.0 h',
↳ '15.0 min', '59.0 s'])],
...                               True))
1
```

**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

$$extother \cdot extself$$

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,))
>>> 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

```

## 22.10.4 fipy.variables.binaryOperatorVariable

## 22.10.5 fipy.variables.cellToFaceVariable

## 22.10.6 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\_\_**(*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

**\_\_array\_wrap\_\_**(*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0])))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

**\_\_bool\_\_()**

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
→ Use a.any() or a.all()
```

**\_\_call\_\_(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)
```

(continues on next page)

(continued from previous page)

```
>>> 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
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**property faceGrad**

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

**property faceGradAverage**

Deprecated since version 3.3: use *grad.arithmeticFaceValue* instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

**property faceValue**

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```

>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

```

>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

```

>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

**property gaussGrad**

Return  $\frac{1}{V_P} \sum_f \vec{n} \phi_f A_f$  as a rank-1 *CellVariable* (first-order gradient).

**getsctype**(*default=None*)

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↪array((1., 1.)))
True
```

**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')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h
↳', 'min', 's'),
...                                     ['3.0 d', '15.0 h',
↳ '15.0 min', '59.0 s'])]),
...                                     True))
1
```

**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.
```

(continues on next page)



(continued from previous page)

```
←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

$$extother \cdot extself$$

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.

```
>>> from fipy import *
>>> v = CellVariable(mesh=Grid1D(), hasOld=False)
>>> v.updateOld()
Traceback (most recent call last):
....
```

(continues on next page)

(continued from previous page)

```
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
```

**22.10.7 fipy.variables.constant****22.10.8 fipy.variables.constraintMask****22.10.9 fipy.variables.coupledCellVariable****22.10.10 fipy.variables.distanceVariable****Classes**


---

*DistanceVariable*(\*args, \*\*kwds)

A *DistanceVariable* object calculates  $\phi$  so it satisfies,

---

**class** fipy.variables.distanceVariable.**DistanceVariable**(\*args, \*\*kwds)

Bases: *CellVariable*

A *DistanceVariable* object calculates  $\phi$  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.))
```

(continues on next page)

(continued from previous page)

```
>>> 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.,
```

(continues on next page)

(continued from previous page)

```

...           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)

```

(continues on next page)

(continued from previous page)

```
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> print(numerix.allequal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]
```

**\_\_array\_\_** (*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

**\_\_array\_wrap\_\_** (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

**\_\_bool\_\_** ()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↪ Use a.any() or a.all()
```

**\_\_call\_\_** (*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))
```

(continues on next page)



(continued from previous page)

```
[ 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)
```

(continues on next page)

(continued from previous page)

```

>>> 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, \*\*kwds)****\_\_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)

```

(continues on next page)

(continued from previous page)

```
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> print(numerix.allequal((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
```

**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)
[[-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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**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 faceGradAverage**

Deprecated since version 3.3: use `grad.arithmeticFaceValue` instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

**property faceValue**

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

**property gaussGrad**

Return  $\frac{1}{V_p} \sum_f \vec{n}_f \phi_f A_f$  as a rank-1 *CellVariable* (first-order gradient).

**getsctype** (*default=None*)

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↪array((1., 1.)))
True
```

**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')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h
↳', 'min', 's'),
...                                     ['3.0 d', '15.0 h',
↳ '15.0 min', '59.0 s'])]),
...                                     True))
1
```

### 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

$$extoother \cdot extself$$

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
```

## 22.10.11 fipy.variables.exponentialNoiseVariable

### Classes

<code>ExponentialNoiseVariable(*args, **kwds)</code>	Represents an exponential distribution of random numbers with the probability distribution
--	--

**class** `fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable(*args, **kwds)`

Bases: `NoiseVariable`

Represents an exponential distribution of random numbers with the probability distribution

$$\mu^{-1}e^{-\frac{x}{\mu}}$$

with a mean parameter  $\mu$ .

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform Cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> mean = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = ExponentialNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), mean = mean)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 100)
```

and compare to a Gaussian distribution

```
>>> from fipy.variables.cellVariable import CellVariable
>>> expdist = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=5)
...     histplot = Viewer(vars=(histogram, expdist),
...                         datamin=0, datamax=1.5)
```

```
>>> from fipy.tools.numerix import arange, exp
```

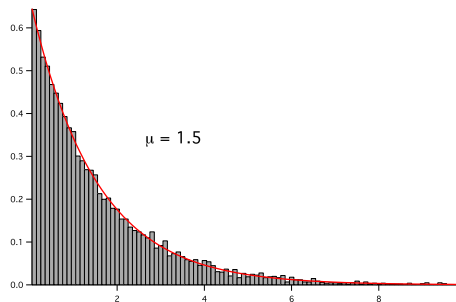
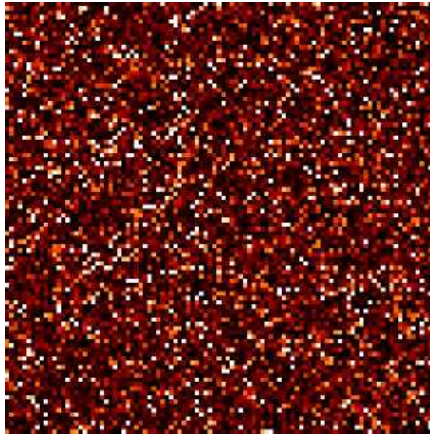
```
>>> for mu in arange(0.5, 3, 0.5):
...     mean.value = (mu)
...     expdist.value = ((1/mean)*exp(-x/mean))
...     if __name__ == '__main__':
...         import sys
```

(continues on next page)

(continued from previous page)

```
...     print("mean: %g" % mean, file=sys.stderr)
...     viewer.plot()
...     histplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```



### Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **mean** (*float*) – The mean of the distribution  $\mu$ .

### `__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))
```

(continues on next page)



(continued from previous page)

```

>>> 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.allequal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]

```

**\_\_array\_\_** (*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```

>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]

```

A dimensional *Variable* will convert to the numeric value in its base units

```

>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])

```

**\_\_array\_wrap\_\_** (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```

>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>

```

```

>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>

```

**\_\_bool\_\_** ()

```

>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↳ Use a.any() or a.all()

```

**\_\_call\_\_** (*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(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, **kwds)``__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(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
```

**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)
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```

>>> 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**property faceGrad**

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

**property faceGradAverage**

Deprecated since version 3.3: use *grad.arithmeticFaceValue* instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

**property faceValue**

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```

>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

```

>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))

```

(continues on next page)



(continued from previous page)

```
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

**property gaussGrad**

Return  $\frac{1}{V_P} \sum_f \vec{n}_f A_f$  as a rank-1 *CellVariable* (first-order gradient).

**getsctype** (*default=None*)

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↳array((1., 1.)))
True
```

**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')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h',
↳ 'min', 's'),
...
↳ '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

$$extother \cdot extself$$

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,))
>>> 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)

```

(continues on next page)

(continued from previous page)

```
>>> 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
```

**22.10.12 fipy.variables.faceGradContributionsVariable****22.10.13 fipy.variables.faceGradVariable****22.10.14 fipy.variables.faceVariable****Classes**

```
FaceVariable(*args, **kwds)
```

**param mesh**

the mesh that defines the geometry of this *Variable*

```
class fipy.variables.faceVariable.FaceVariable(*args, **kwds)
```

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)
>>> from fipy.variables.cellVariable import CellVariable
>>> a = CellVariable(value=(0, 0, 1, 1), mesh=mesh)
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> print(numerix.allegal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]
```

**\_\_array\_\_(t=None)**

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

`__array_wrap__(arr, context=None)`

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples of ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0])))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

`__bool__()`

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
→ Use a.any() or a.all()
```

`__call__()`

“Evaluate” the *Variable* and return its value

```
>>> a = Variable(value=3)
>>> print(a())
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b()
7
```

`__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
```

(continues on next page)



(continued from previous page)

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```

>>> 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.allequal((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, '1')))
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]
>>> 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)
>>> from builtins import range
>>> 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**getscdtype**(*default=None*)

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getscdtype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getscdtype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getscdtype() == numerix.NUMERIX.obj2sctype(numerix.
↳array((1., 1.)))
True
```

**inBaseUnits**()

Return the value of the *Variable* with all units reduced to their base SI elements.

```
>>> e = Variable(value="2.7 Hartree*Nav")
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

**inUnitsOf**(\**units*)

Returns one or more *Variable* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *Variable*.

```
>>> freeze = Variable('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *Variable* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = Variable(value=314159., unit='s')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h
↳', 'min', 's')),
...                                     ['3.0 d', '15.0 h',
↳ '15.0 min', '59.0 s']]),
...                                     True))
1
```

### 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

$$extother \cdot extself$$

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
```

## 22.10.15 fipy.variables.gammaNoiseVariable

### Classes

<i>GammaNoiseVariable</i> (*args, **kws)	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  $\alpha$ , a rate parameter  $\beta$ , and  $\Gamma(z) = \int_0^\infty t^{z-1} e^{-t} dt$ .

Seed the random module for the sake of deterministic test results.

```
>>> from fipy import numerix
>>> numerix.random.seed(1)
```

We generate noise on a uniform Cartesian mesh

```
>>> from fipy.variables.variable import Variable
>>> alpha = Variable()
>>> beta = Variable()
>>> from fipy.meshes import Grid2D
>>> noise = GammaNoiseVariable(mesh = Grid2D(nx = 100, ny = 100), shape = alpha,
↳rate = beta)
```

We histogram the root-volume-weighted noise distribution

```
>>> from fipy.variables.histogramVariable import HistogramVariable
>>> histogram = HistogramVariable(distribution = noise, dx = 0.1, nx = 300)
```

and compare to a Gaussian distribution

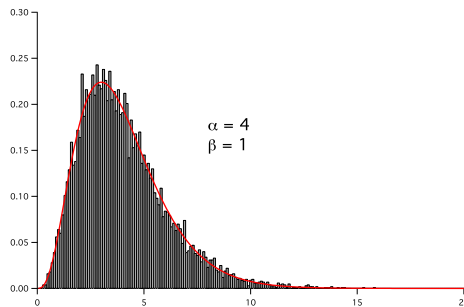
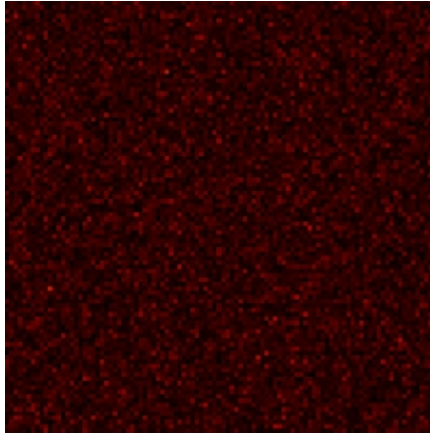
```
>>> from fipy.variables.cellVariable import CellVariable
>>> x = CellVariable(mesh=histogram.mesh, value=histogram.mesh.cellCenters[0])
>>> from scipy.special import gamma as Gamma
>>> from fipy.tools.numerix import exp
>>> gammadist = (x**(alpha - 1) * (beta**alpha * exp(-beta * x)) / Gamma(alpha))
```

```
>>> if __name__ == '__main__':
...     from fipy import Viewer
...     viewer = Viewer(vars=noise, datamin=0, datamax=30)
...     histoplot = Viewer(vars=(histogram, gammadist),
...                          datamin=0, datamax=1)
```

```
>>> from fipy.tools.numerix import arange
```

```
>>> for shape in arange(1, 8, 1):
...     alpha.value = shape
...     for rate in arange(0.5, 2.5, 0.5):
...         beta.value = rate
...         if __name__ == '__main__':
...             import sys
...             print("alpha: %g, beta: %g" % (alpha, beta), file=sys.stderr)
...             viewer.plot()
...             histoplot.plot()
```

```
>>> print(abs(noise.faceGrad.divergence.cellVolumeAverage) < 5e-15)
1
```



### Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **shape** (*float*) – The shape parameter,  $\alpha$ .
- **rate** (*float*) – The rate or inverse scale parameter,  $\beta$ .

### `__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.allequal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]
```

**\_\_array\_\_** (*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

**\_\_array\_wrap\_\_** (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0])))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

**\_\_bool\_\_** ()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↪ Use a.any() or a.all()
```

**\_\_call\_\_** (*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, **kwds)``__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(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)
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

(continues on next page)

(continued from previous page)

```

>>> 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**property faceGrad**

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

**property faceGradAverage**

Deprecated since version 3.3: use *grad.arithmeticFaceValue* instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

**property faceValue**

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```

>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

```

>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))

```

(continues on next page)

(continued from previous page)

```
>>> 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).

**getsctype** (*default=None*)

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↳array((1., 1.)))
True
```

**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')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h',
↳ 'min', 's'),
...                                     ['3.0 d', '15.0 h',
↳ '15.0 min', '59.0 s'])],
...                               True))
1
```

**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

$$extoother \cdot extself$$

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,))
>>> 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)

```

(continues on next page)



(continued from previous page)

```
>>> 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
```

**22.10.16 fipy.variables.gaussCellGradVariable****22.10.17 fipy.variables.gaussianNoiseVariable****Classes**

*GaussianNoiseVariable*(\*args, \*\*kwargs)

Represents a normal (Gaussian) distribution of random numbers with mean  $\mu$  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  $\mu$  and variance  $\langle \eta(\vec{r})\eta(\vec{r}') \rangle = \sigma^2$ , which has the probability distribution

$$\frac{1}{\sigma\sqrt{2\pi}} \exp - \frac{(x - \mu)^2}{2\sigma^2}$$

For example, the variance of thermal noise that is uncorrelated in space and time is often expressed as

$$\langle \eta(\vec{r}, t) \eta(\vec{r}', t') \rangle = M k_B T \delta(\vec{r} - \vec{r}') \delta(t - t')$$

which can be obtained with:

```
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

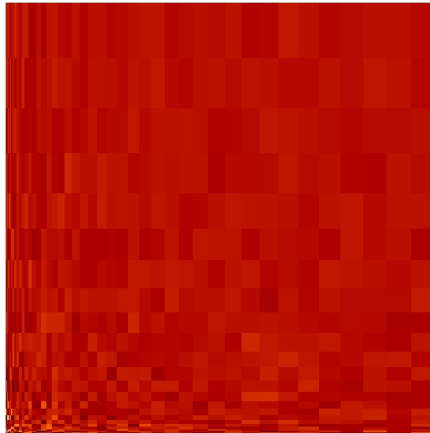
```
>>> gauss = CellVariable(mesh = histogram.mesh)
>>> x = histogram.mesh.cellCenters[0]
>>> gauss.value = ((1/(sqrt(variance * 2 * pi))) * exp(-(x - mean)**2 / (2 *
↪variance)))
```

```
>>> if __name__ == '__main__':
...     from fipy.viewers import Viewer
...     viewer = Viewer(vars=noise,
...                     datamin=-5, datamax=5)
...     histoplot = Viewer(vars=(histogram, gauss))
```

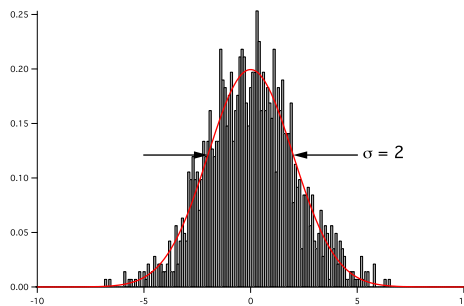
```
>>> from builtins import range
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histplot.plot()
```

```
>>> 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,  $\mu$ .
- **variance** (*float*) – The variance of the noise distribution,  $\sigma^2$ .

### `__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\_\_**(*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

**\_\_array\_wrap\_\_**(*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples of ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0])))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

**\_\_bool\_\_**()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
```

(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*, \*\**kwds*)**\_\_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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

### property faceGrad

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

### property faceGradAverage

Deprecated since version 3.3: use *grad.arithmeticFaceValue* instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

### property faceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{df_2}{d_{12}} + \phi_2$$

```

>>> 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).

**getsctype**(default=None)

Returns the Numpy *sctype* of the underlying array.

```

>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↪array((1., 1.)))
True

```

**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:

$$\phi_f = \frac{\phi_1 \phi_2}{(\phi_2 - \phi_1) \frac{d_{f2}}{d_{12}} + \phi_1}$$



(continued from previous page)

```
...
1 True))
```

**property leastSquaresGrad**

Return  $\nabla\phi$ , which is determined by solving for  $\nabla\phi$  in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n} \cdot \nabla\phi)_{AP}$$

The matrix equation is derived by minimizing the following least squares sum,

$$F(\phi_x, \phi_y) = \sqrt{\sum_f (d_{AP} \vec{n}_{AP} \cdot \nabla\phi - d_{AP} (\vec{n}_{AP} \cdot \nabla\phi)_{AP})^2}$$

Tests

```
>>> from fipy import Grid2D
>>> m = Grid2D(nx=2, ny=2, dx=0.1, dy=2.0)
>>> print(numerix.allclose(CellVariable(mesh=m, value=(0, 1, 3, 6)).
↳ leastSquaresGrad.globalValue, \
...                                     [[8.0, 8.0, 24.0, 24.0],
...                                     [1.2, 2.0, 1.2, 2.0]]))
True
```

```
>>> from fipy import Grid1D
>>> print(numerix.allclose(CellVariable(mesh=Grid1D(dx=(2.0, 1.0, 0.5)),
...                                     value=(0, 1, 2)).leastSquaresGrad.
↳ globalValue, [[0.461538461538, 0.8, 1.2]]))
True
```

**property 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

$$extother \cdot extself$$

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
```

## 22.10.18 fipy.variables.harmonicCellToFaceVariable

## 22.10.19 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\_\_** (*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

**\_\_array\_wrap\_\_** (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

**\_\_bool\_\_** ()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↳ Use a.any() or a.all()
```

**\_\_call\_\_** (*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(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
```

(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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**property faceGrad**

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

**property faceGradAverage**

Deprecated since version 3.3: use `grad.arithmeticFaceValue` instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

**property faceValue**

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```

>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

```

>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

```

>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

**property gaussGrad**

Return  $\frac{1}{V_p} \sum_f \vec{n}_f \phi_f A_f$  as a rank-1 *CellVariable* (first-order gradient).

**getsctype**(default=None)

Returns the Numpy *sctype* of the underlying array.

```

>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↪array((1., 1.)))
True

```

**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:

$$\phi_f = \frac{\phi_1 \phi_2}{(\phi_2 - \phi_1) \frac{d_{f2}}{d_{12}} + \phi_1}$$



(continued from previous page)

```
... 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

$$extoother \cdot extself$$

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.

```
>>> 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
```

### 22.10.20 fipy.variables.interfaceAreaVariable

### 22.10.21 fipy.variables.interfaceFlagVariable

### 22.10.22 fipy.variables.leastSquaresCellGradVariable

### 22.10.23 fipy.variables.levelSetDiffusionVariable

### 22.10.24 fipy.variables.meshVariable



## Classes

<code>MeshVariable(*args, **kwargs)</code>	Abstract base class for a <i>Variable</i> 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]))
```

(continues on next page)

(continued from previous page)

```
True
>>> print(a & b)
[0 0 0 1]
```

**\_\_array\_\_**(*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

**\_\_array\_wrap\_\_**(*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

**\_\_bool\_\_**()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↳ Use a.any() or a.all()
```

**\_\_call\_\_**()

“Evaluate” the *Variable* and return its value

```
>>> a = Variable(value=3)
>>> print(a())
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b()
7
```

**\_\_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
>>> 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
```

(continues on next page)

(continued from previous page)

```
(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(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
```

**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 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()

```

(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]))
```

**dot** (*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**getsctype** (*default=None*)

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↳array((1., 1.)))
True
```

**inBaseUnits**()

Return the value of the *Variable* with all units reduced to their base SI elements.

```
>>> e = Variable(value="2.7 Hartree*Nav")
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

**inUnitsOf**(\**units*)

Returns one or more *Variable* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *Variable*.

```
>>> freeze = Variable('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```



If several units are specified, the return value is a tuple of *Variable* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = Variable(value=314159., unit='s')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h
↳', 'min', 's'),
...                                     ['3.0 d', '15.0 h',
↳ '15.0 min', '59.0 s'])]),
...                                     True))
1
```

### 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

$$extother \cdot extself$$

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
```

### 22.10.25 fipy.variables.minmodCellToFaceVariable

### 22.10.26 fipy.variables.modCellGradVariable

### 22.10.27 fipy.variables.modCellToFaceVariable

### 22.10.28 fipy.variables.modFaceGradVariable

### 22.10.29 fipy.variables.modPhysicalField

### 22.10.30 fipy.variables.modularVariable

#### Classes

*ModularVariable*(\*args, \*\*kwargs)

The *ModularVariable* defines a variable that exists on the circle between  $-\pi$  and  $\pi$

**class** fipy.variables.modularVariable.**ModularVariable**(\*args, \*\*kwargs)

Bases: *CellVariable*

The *ModularVariable* defines a variable that exists on the circle between  $-\pi$  and  $\pi$

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)
>>> from fipy.variables.cellVariable import CellVariable
>>> a = CellVariable(value=(0, 0, 1, 1), mesh=mesh)
>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> print(numerix.allegal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]
```

**\_\_array\_\_** (*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

**\_\_array\_wrap\_\_** (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

**\_\_bool\_\_** ()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↳ Use a.any() or a.all()
```

`__call__` (*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
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

(continues on next page)

(continued from previous page)

```

>>> 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, \*\*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:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

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)
```

(continues on next page)

(continued from previous page)

```

[[ 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**property faceGrad**

Return  $\nabla\phi$  as a rank-1 *FaceVariable* (second-order gradient). Adjusted for a *ModularVariable*

**property faceGradAverage**

Deprecated since version 3.3: use *grad.arithmeticFaceValue* instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

**property 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).

### `getsctype`(*default=None*)

Returns the Numpy *sctype* of the underlying array.

```

>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↪array((1., 1.)))
True

```

### 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')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h',
↳ 'min', 's'),
...
↳ '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

$$extoother \cdot extself$$

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))

```

(continues on next page)



(continued from previous page)

```
>>> 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
>>> 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
```

**22.10.31 fipy.variables.noiseVariable**

## Classes

`NoiseVariable(*args, **kwds)`

**Attention:**

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

**class** `fipy.variables.noiseVariable.NoiseVariable(*args, **kwds)`

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__(t=None)`

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

### `__array_wrap__(arr, context=None)`

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

### `__bool__()`

```

>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↳ Use a.any() or a.all()

```

`__call__` (*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
(Variable(value=array(3)) != 4)
>>> b()
1
```

**static** **\_\_new\_\_(cls, \*args, \*\*kws)**

**\_\_nonzero\_\_()**

```
>>> print(bool(Variable(value=0)))
0
```

(continues on next page)

(continued from previous page)

```
>>> 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)))
```

(continues on next page)



(continued from previous page)

```
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.]
```

### 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

### property faceGrad

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

### property faceGradAverage

Deprecated since version 3.3: use *grad.arithmeticFaceValue* instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

### property faceValue

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{df_2}{d_{12}} + \phi_2$$

```

>>> 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).

**getsctype**(default=None)

Returns the Numpy *sctype* of the underlying array.

```

>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↪array((1., 1.)))
True

```

**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:

$$\phi_f = \frac{\phi_1 \phi_2}{(\phi_2 - \phi_1) \frac{d_{f2}}{d_{12}} + \phi_1}$$



(continued from previous page)

```
...
1 True))
```

**property leastSquaresGrad**

Return  $\nabla\phi$ , which is determined by solving for  $\nabla\phi$  in the following matrix equation,

$$\nabla\phi \cdot \sum_f d_{AP}^2 \vec{n}_{AP} \otimes \vec{n}_{AP} = \sum_f d_{AP}^2 (\vec{n} \cdot \nabla\phi)_{AP}$$

The matrix equation is derived by minimizing the following least squares sum,

$$F(\phi_x, \phi_y) = \sqrt{\sum_f (d_{AP} \vec{n}_{AP} \cdot \nabla\phi - d_{AP} (\vec{n}_{AP} \cdot \nabla\phi)_{AP})^2}$$

Tests

```
>>> from fipy import Grid2D
>>> m = Grid2D(nx=2, ny=2, dx=0.1, dy=2.0)
>>> print(numerix.allclose(CellVariable(mesh=m, value=(0, 1, 3, 6)).
↳leastSquaresGrad.globalValue, \
...                                     [[8.0, 8.0, 24.0, 24.0],
...                                     [1.2, 2.0, 1.2, 2.0]]))
True
```

```
>>> from fipy import Grid1D
>>> print(numerix.allclose(CellVariable(mesh=Grid1D(dx=(2.0, 1.0, 0.5)),
...                                     value=(0, 1, 2)).leastSquaresGrad.
↳globalValue, [[0.461538461538, 0.8, 1.2]]))
True
```

**property 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

$$extother \cdot extself$$

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
```

## 22.10.32 fipy.variables.operatorVariable

## 22.10.33 fipy.variables.scharfetterGummelFaceVariable

### Classes

```
ScharfetterGummelFaceVariable(*args, **kwds)
```

**param mesh**

the mesh that defines the geometry of this *Variable*

```
class fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable(*args,
                                                                                   **kwds)
```

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.allequal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]

```

**\_\_array\_\_**(*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```

>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]

```

A dimensional *Variable* will convert to the numeric value in its base units

```

>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])

```

**\_\_array\_wrap\_\_**(*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```

>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>

```

```

>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0])))
<class 'fipy.variables.unaryOperatorVariable...unOp'>

```

**\_\_bool\_\_**()

```

>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↪ Use a.any() or a.all()

```

**\_\_call\_\_**()

“Evaluate” the *Variable* and return its value

```

>>> a = Variable(value=3)
>>> print(a())
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b()
7

```

`__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(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.allequal((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, '1')))
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 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()

```

(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)
>>> from builtins import range
>>> 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**getscdtype**(*default=None*)

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getscdtype() == numerix.NUMERIX.obj2scdtype(numerix.array(1))
True
>>> Variable(1.).getscdtype() == numerix.NUMERIX.obj2scdtype(numerix.array(1.))
True
>>> Variable((1, 1.)).getscdtype() == numerix.NUMERIX.obj2scdtype(numerix.
↳array((1., 1.)))
True
```

**inBaseUnits()**

Return the value of the *Variable* with all units reduced to their base SI elements.

```
>>> e = Variable(value="2.7 Hartree*Nav")
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

**inUnitsOf(\*units)**

Returns one or more *Variable* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *Variable*.

```
>>> freeze = Variable('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *Variable* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = Variable(value=314159., unit='s')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h
↳', 'min', 's'),
...                                     ['3.0 d', '15.0 h',
↳'15.0 min', '59.0 s'])],
...                                     True))
1
```

**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

$$extoother \cdot extself$$

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

```

## 22.10.34 fipy.variables.surfactantConvectionVariable

### Classes

<code>SurfactantConvectionVariable(*args, **kws)</code>	Convection coefficient for the <i>ConservativeSurfactantEquation</i> .
---	--

```
class fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable(*args,
                                                                              **kws)
```

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.allequal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]

```

**\_\_array\_\_(t=None)**

Attempt to convert the *Variable* to a numerix *array* object

```

>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]

```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

### `__array_wrap__`(*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0])))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

### `__bool__`()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
→ Use a.any() or a.all()
```

### `__call__`()

“Evaluate” the *Variable* and return its value

```
>>> a = Variable(value=3)
>>> print(a())
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b()
7
```

### `__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(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*, \*\**kwds*)**\_\_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.allequal((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, '1')))
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 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]))

```

**property divergence**

the divergence of *self*,  $\vec{u}$ ,

$$\nabla \cdot \vec{u} \approx \frac{\sum_f (\vec{u} \cdot \hat{n})_f A_f}{V_P}$$

**Returns**

**divergence** – one rank lower than *self*

**Return type**

*fipy.variables.cellVariable.CellVariable*

**Examples**

```

>>> from fipy.meshes import Grid2D
>>> from fipy.variables.cellVariable import CellVariable
>>> mesh = Grid2D(nx=3, ny=2)
>>> from builtins import range
>>> var = CellVariable(mesh=mesh, value=list(range(3*2)))
>>> print(var.faceGrad.divergence)
[ 4.  3.  2. -2. -3. -4.]

```

**dot**(*other*, *opShape=None*, *operatorClass=None*)

Return the mesh-element-by-mesh-element (cell-by-cell, face-by-face, etc.) scalar product

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**getsctype**(*default=None*)

Returns the Numpy *sctype* of the underlying array.

```

>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↳array((1., 1.)))
True

```

**inBaseUnits**()

Return the value of the *Variable* with all units reduced to their base SI elements.

```

>>> e = Variable(value="2.7 Hartree*Nav")
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1

```

**inUnitsOf(\*units)**

Returns one or more *Variable* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *Variable*.

```
>>> freeze = Variable('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *Variable* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = Variable(value=314159., unit='s')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h',
↳ 'min', 's'),
...                                     ['3.0 d', '15.0 h',
↳ '15.0 min', '59.0 s'])],
...                                     True))
1
```

**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

$$extoother \cdot extself$$

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)
```

(continues on next page)

(continued from previous page)

```

>>> 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.allclose(var.shape, (6,)))
True
>>> print(numerix.allclose(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allclose(var.grad.shape, (2, 6)))
True
>>> print(numerix.allclose(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.allclose((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
```

## 22.10.35 fipy.variables.surfactantVariable

### Classes

*SurfactantVariable*(\*args, \*\*kws)

The *SurfactantVariable* maintains a conserved volumetric concentration on cells adjacent to, but in front of, the interface.

**class** fipy.variables.surfactantVariable.**SurfactantVariable**(\*args, \*\*kws)

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.allequal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]
```

`__array__(t=None)`

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

`__array_wrap__(arr, context=None)`

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

`__bool__()`

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↳ Use a.any() or a.all()
```

`__call__` (*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
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

(continues on next page)

(continued from previous page)

```

>>> 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, \*\*kwds*)**\_\_nonzero\_\_**(*C*)

```

>>> 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**property faceGrad**

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

**property faceGradAverage**

Deprecated since version 3.3: use `grad.arithmeticFaceValue` instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

**property faceValue**

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```
>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

**property gaussGrad**

Return  $\frac{1}{V_p} \sum_f \vec{n}_f \phi_f A_f$  as a rank-1 *CellVariable* (first-order gradient).

**getscstype(*default=None*)**

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getscstype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getscstype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1)).getscstype() == numerix.NUMERIX.obj2sctype(numerix.
↪array((1., 1.)))
True
```

**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')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h
↳', 'min', 's'),
...                                     ['3.0 d', '15.0 h',
↳ '15.0 min', '59.0 s'])],
...                                     True))
1

```

**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

$$extoother \cdot extself$$

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.allclose(var.shape, (6,)))
True
>>> print(numerix.allclose(var.arithmeticFaceValue.shape, (17,)))
True
>>> print(numerix.allclose(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
```

### 22.10.36 fipy.variables.test

Test numeric implementation of the mesh

### 22.10.37 fipy.variables.unaryOperatorVariable

### 22.10.38 fipy.variables.uniformNoiseVariable

#### Classes

---

<code>UniformNoiseVariable(*args, **kwargs)</code>	Represents a uniform distribution of random numbers.
--	--

---

**class** `fipy.variables.uniformNoiseVariable.UniformNoiseVariable(*args, **kwargs)`

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))
```

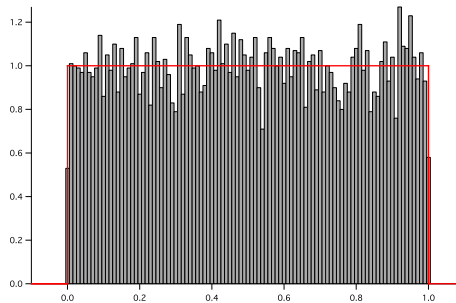
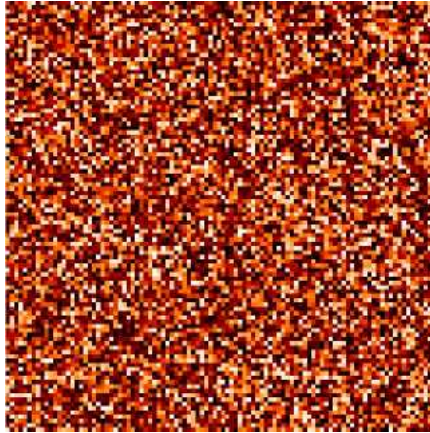
and histogram the noise

```
>>> 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)
```

```
>>> from builtins import range
>>> for i in range(10):
...     noise.scramble()
...     if __name__ == '__main__':
...         viewer.plot()
...         histoplot.plot()
```





### Parameters

- **mesh** (*Mesh*) – The mesh on which to define the noise.
- **minimum** (*float*) – The minimum (not-inclusive) value of the distribution.
- **maximum** (*float*) – The maximum (not-inclusive) value of the distribution.

### `__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.allequal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]
```

**\_\_array\_\_** (*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```
>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]
```

A dimensional *Variable* will convert to the numeric value in its base units

```
>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])
```

**\_\_array\_wrap\_\_** (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```
>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>
```

```
>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>
```

**\_\_bool\_\_** ()

```
>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↪ Use a.any() or a.all()
```

**\_\_call\_\_** (*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, \*\*kwds)****\_\_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(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)
```

(continues on next page)

(continued from previous page)

```
>>> 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)
```

(continues on next page)



(continued from previous page)

```

>>> 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

$$extself \cdot extother$$

Both *self* and *other* can be of arbitrary rank, and *other* does not need to be a *MeshVariable*.

**property faceGrad**

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using differencing for the normal direction(second-order gradient).

**property faceGradAverage**

Deprecated since version 3.3: use `grad.arithmeticFaceValue` instead

Return  $\nabla\phi$  as a rank-1 *FaceVariable* using averaging for the normal direction(second-order gradient)

**property faceValue**

Returns a *FaceVariable* whose value corresponds to the arithmetic interpolation of the adjacent cells:

$$\phi_f = (\phi_1 - \phi_2) \frac{d_{f2}}{d_{12}} + \phi_2$$

```

>>> from fipy.meshes import Grid1D
>>> from fipy import numerix
>>> mesh = Grid1D(dx = (1., 1.))
>>> L = 1
>>> R = 2
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (0.5 / 1.) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True

```

```

>>> mesh = Grid1D(dx = (2., 4.))
>>> var = CellVariable(mesh = mesh, value = (L, R))

```

(continues on next page)

(continued from previous page)

```
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (1.0 / 3.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

```
>>> mesh = Grid1D(dx = (10., 100.))
>>> var = CellVariable(mesh = mesh, value = (L, R))
>>> faceValue = var.arithmeticFaceValue[mesh.interiorFaces.value]
>>> answer = (R - L) * (5.0 / 55.0) + L
>>> print(numerix.allclose(faceValue, answer, atol = 1e-10, rtol = 1e-10))
True
```

**property gaussGrad**

Return  $\frac{1}{V_P} \sum_f \vec{n}_f A_f$  as a rank-1 *CellVariable* (first-order gradient).

**getsctype** (*default=None*)

Returns the Numpy *sctype* of the underlying array.

```
>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↳array((1., 1.)))
True
```

**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')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h',
↳ 'min', 's'),
...
↳ '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

$$extoother \cdot extself$$

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,))
>>> 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)

```

(continues on next page)

(continued from previous page)

```
>>> 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
```

## 22.10.39 fipy.variables.variable

### Classes

<i>Variable</i> (*args, **kwds)	Lazily evaluated quantity with units.
---------------------------------	---------------------------------------

**class** fipy.variables.variable.**Variable**(\*args, \*\*kwds)Bases: `object`

Lazily evaluated quantity with units.

Using a *Variable* in a mathematical expression will create an automatic dependency *Variable*, e.g.,

```
>>> a = Variable(value=3)
>>> b = 4 * a
>>> b
(Variable(value=array(3)) * 4)
>>> b()
12
```

Changes to the value of a *Variable* will automatically trigger changes in any dependent *Variable* objects

```
>>> a.setValue(5)
>>> b
(Variable(value=array(5)) * 4)
>>> print(b())
20
```

Create a *Variable*.

```
>>> Variable(value=3)
Variable(value=array(3))
>>> Variable(value=3, unit="m")
Variable(value=PhysicalField(3,'m'))
>>> Variable(value=3, unit="m", array=numerix.zeros((3, 2), '1'))
Variable(value=PhysicalField(array([[3, 3],
[3, 3],
[3, 3]]), 'm'))
```

### Parameters

- **value** (int or float or *array\_like*) –
- **unit** (*str* or *PhysicalUnit*) – The physical units of the variable
- **array** (*ndarray*, *optional*) – The storage array for the *Variable*
- **name** (*str*) – The user-readable name of the *Variable*
- **cached** (*bool*) – whether to cache or always recalculate the value

### `__abs__()`

Following test it to fix a bug with C inline string using *abs()* instead of *fabs()*

```
>>> print(abs(Variable(2.3) - Variable(1.2)))
1.1
```

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)
```

(continues on next page)



(continued from previous page)

```

>>> b = CellVariable(value=(0, 1, 0, 1), mesh=mesh)
>>> print(numerix.allequal((a == 0) & (b == 1), [False, True, False, False]))
True
>>> print(a & b)
[0 0 0 1]

```

**\_\_array\_\_** (*t=None*)

Attempt to convert the *Variable* to a numerix *array* object

```

>>> v = Variable(value=[2, 3])
>>> print(numerix.array(v))
[2 3]

```

A dimensional *Variable* will convert to the numeric value in its base units

```

>>> v = Variable(value=[2, 3], unit="mm")
>>> numerix.array(v)
array([ 0.002,  0.003])

```

**\_\_array\_wrap\_\_** (*arr, context=None*)

Required to prevent numpy not calling the reverse binary operations. Both the following tests are examples ufuncs.

```

>>> print(type(numerix.array([1.0, 2.0]) * Variable([1.0, 2.0])))
<class 'fipy.variables.binaryOperatorVariable...binOp'>

```

```

>>> from scipy.special import gamma as Gamma
>>> print(type(Gamma(Variable([1.0, 2.0]))))
<class 'fipy.variables.unaryOperatorVariable...unOp'>

```

**\_\_bool\_\_** ()

```

>>> print(bool(Variable(value=0)))
0
>>> print(bool(Variable(value=(0, 0, 1, 1))))
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one element is ambiguous.
↳ Use a.any() or a.all()

```

**\_\_call\_\_** ()

“Evaluate” the *Variable* and return its value

```

>>> a = Variable(value=3)
>>> print(a())
3
>>> b = a + 4
>>> b
(Variable(value=array(3)) + 4)
>>> b()
7

```

`__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
```

(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(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.allequal((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, '1')))
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]))

```

### getsctype(*default=None*)

Returns the Numpy *sctype* of the underlying array.

```

>>> Variable(1).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1))
True
>>> Variable(1.).getsctype() == numerix.NUMERIX.obj2sctype(numerix.array(1.))
True
>>> Variable((1, 1.)).getsctype() == numerix.NUMERIX.obj2sctype(numerix.
↳array((1., 1.)))
True

```

### inBaseUnits()

Return the value of the *Variable* with all units reduced to their base SI elements.

```
>>> e = Variable(value="2.7 Hartree*Nav")
>>> print(e.inBaseUnits().allclose("7088849.01085 kg*m**2/s**2/mol"))
1
```

**inUnitsOf(\*units)**

Returns one or more *Variable* objects that express the same physical quantity in different units. The units are specified by strings containing their names. The units must be compatible with the unit of the object. If one unit is specified, the return value is a single *Variable*.

```
>>> freeze = Variable('0 degC')
>>> print(freeze.inUnitsOf('degF').allclose("32.0 degF"))
1
```

If several units are specified, the return value is a tuple of *Variable* instances with with one element per unit such that the sum of all quantities in the tuple equals the the original quantity and all the values except for the last one are integers. This is used to convert to irregular unit systems like hour/minute/second. The original object will not be changed.

```
>>> t = Variable(value=314159., unit='s')
>>> from builtins import zip
>>> print(numerix.allclose([e.allclose(v) for (e, v) in zip(t.inUnitsOf('d', 'h',
↪', 'min', 's'),
...                                     ['3.0 d', '15.0 h',
↪'15.0 min', '59.0 s'])],
...                               True))
1
```

**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
```

## 22.11 fipy.viewers

Tools for displaying the values of *Variable* objects

### Functions

<i>Viewer</i> (vars[, title, limits, FIPY_VIEWER])	Generic function for creating a <i>Viewer</i> .
--	---

### 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`

### 22.11.1 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 *Variables* against different *Matplotlib Axes*

```
>>> from matplotlib import pyplot as plt
>>> from fiPy import *
```

```
>>> plt.ion()
>>> fig = plt.figure()
```

```
>>> ax1 = plt.subplot((221))
>>> ax2 = plt.subplot((223))
>>> ax3 = plt.subplot((224))
```

```
>>> k = Variable(name="k", value=0.)
```

```
>>> mesh1 = Grid1D(nx=100)
>>> x, = mesh1.cellCenters
>>> xVar = CellVariable(mesh=mesh1, name="x", value=x)
>>> viewer1 = MatplotlibViewer(vars=(numerix.sin(0.1 * k * xVar), numerix.cos(0.1 *
↵ k * xVar / numerix.pi)),
```

(continues on next page)

(continued from previous page)

```

...         limits={'xmin': 10, 'xmax': 90},
...         datamin=-0.9, datamax=2.0,
...         title="Grid1D test",
...         axes=ax1,
...         legend=None)

```

```

>>> mesh2 = Grid2D(nx=50, ny=100, dx=0.1, dy=0.01)
>>> x, y = mesh2.cellCenters
>>> xyVar = CellVariable(mesh=mesh2, name="x y", value=x * y)
>>> viewer2 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Grid2D test",
...                             axes=ax2,
...                             colorbar=None)

```

```

>>> mesh3 = (Grid2D(nx=5, ny=10, dx=0.1, dy=0.1)
...           + (Tri2D(nx=5, ny=5, dx=0.1, dy=0.1)
...              + ((0.5,), (0.2,))))
>>> x, y = mesh3.cellCenters
>>> xyVar = CellVariable(mesh=mesh3, name="x y", value=x * y)
>>> viewer3 = MatplotlibViewer(vars=numerix.sin(k * xyVar),
...                             limits={'ymin': 0.1, 'ymax': 0.9},
...                             datamin=-0.9, datamax=2.0,
...                             title="Irregular 2D test",
...                             axes=ax3,
...                             cmap = plt.cm.OrRd)

```

```

>>> viewer = MultiViewer(viewers=(viewer1, viewer2, viewer3))
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()

```

```

>>> viewer._promptForOpinion()

```

### Parameters

- **vars** (*CellVariable* or *list*) – the *Variable* objects to display.
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **limits** (*dict*) – a (deprecated) alternative to limit keyword arguments
- **xmin** (*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. abstractMatplotlib2DViewer</code>	
<code>fipy.viewers.matplotlibViewer. abstractMatplotlibViewer</code>	
<code>fipy.viewers.matplotlibViewer. matplotlib1DViewer</code>	
<code>fipy.viewers.matplotlibViewer. matplotlib2DContourViewer</code>	
<code>fipy.viewers.matplotlibViewer. matplotlib2DGridContourViewer</code>	
<code>fipy.viewers.matplotlibViewer. matplotlib2DGridViewer</code>	
<code>fipy.viewers.matplotlibViewer. matplotlib2DViewer</code>	
<code>fipy.viewers.matplotlibViewer. matplotlibSparseMatrixViewer</code>	
<code>fipy.viewers.matplotlibViewer. matplotlibStreamViewer</code>	
<code>fipy.viewers.matplotlibViewer. matplotlibVectorViewer</code>	
<code>fipy.viewers.matplotlibViewer.test</code>	Test numeric implementation of the mesh

## fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer

### Classes

<code>AbstractMatplotlib2DViewer(vars[, title, ...])</code>	Base class for plotting 2D <i>MeshVariable</i> objects with Matplotlib.
---	---

```
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**

<i>AbstractMatplotlibViewer</i> (vars[, title, ...])	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

<code>Matplotlib2DContourViewer(vars[, title, ...])</code>	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={},
    cmap=None,
    colorbar='vertical',
    axes=None,
    levels=None,
    figure=None,
    aspect=None,
    **kwargs)
```

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")
>>> from builtins import range
>>> 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, ti-
                                                                    tle=None,
                                                                    lim-
                                                                    its={},
                                                                    cmap=None,
                                                                    color-
                                                                    bar='vertical',
                                                                    axes=None,
                                                                    fi-
                                                                    gaspect='auto',
                                                                    **kwlim-
                                                                    its)
```

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")
>>> from builtins import range
>>> 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")
>>> from builtins import range
>>> 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**

---

<code>MatplotlibSparseMatrixViewer</code> ([title])	Displays <code>_SparseMatrix</code> objects using <code>Matplotlib</code> .
---	---

---

**class** `fipy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer.MatplotlibSparseMatrixViewer` (*title*='SparseMatrix')

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 <code>Matplotlib</code>
---	---

---

```

class fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer(vars, ti-
                                                                    tle=None,
                                                                    log=False,
                                                                    lim-
                                                                    its={},
                                                                    axes=None,
                                                                    fi-
                                                                    gaspect='auto',
                                                                    den-
                                                                    sity=1,
                                                                    linewidth=None,
                                                                    color=None,
                                                                    cmap=None,
                                                                    norm=None,
                                                                    arrow-
                                                                    size=1,
                                                                    arrowstyle='-
                                                                    |>',
                                                                    min-
                                                                    length=0.1,
                                                                    **kwlim-
                                                                    its)

```

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")
>>> from builtins import range
>>> 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")
...
>>> from builtins import range
>>> 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**

<code>MatplotlibVectorViewer</code> (vars[, title, scale, ...])	Displays a vector plot of a 2D rank-1 <i>MeshVariable</i> using <i>Matplotlib</i>
---	---

```
class fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer(vars, title=None, scale=None, sparsity=None, log=False, limits={}, axes=None, figsize='auto', **kwargs)
```

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")
>>> from builtins import range
>>> 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")
...
>>> from builtins import range
>>> 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

## 22.11.2 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)
>>> 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,
```

(continues on next page)

(continued from previous page)

```

...         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")
>>> from builtins import range
>>> 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")
>>> from builtins import range
>>> 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")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)

```

(continues on next page)

```
... 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(vars[, title, daemon_file, fps])</code>	The <code>MayaviClient</code> uses the <code>Mayavi</code> 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")
>>> from builtins import range
>>> 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")
>>> from builtins import range
>>> 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")
>>> from builtins import range
>>> for kval in range(10):
...     k.setValue(kval)
...     viewer.plot()
>>> viewer._promptForOpinion()

```

Create a *MayaviClient*.

#### Parameters

- **vars** (*CellVariable* or *list*) – *CellVariable* objects to plot
- **title** (*str*, *optional*) – displayed at the top of the *Viewer* window
- **xmin** (*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

`main([argv])`

Simple helper to start up the mayavi application.

### Classes

`MayaviDaemon(*args, **kwargs)`

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

## 22.11.3 `fipy.viewers.multiViewer`

### Classes

<code>MultiViewer</code> (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 `~fipy.viewers.viewer.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.

## 22.11.4 fipy.viewers.test

Test implementation of the viewers

## 22.11.5 fipy.viewers.testinteractive

Interactively test the viewers

## 22.11.6 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          -3.333333333333333
0.15  0.15  2          10           5
0.05  0.45 -2          35          -3.333333333333333
0.15  0.45  5          35           5
```

#### Parameters

**filename** (*str*) – If not *None*, the name of a file to save the image into.

**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.

## 22.11.7 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

<b>Attention:</b> 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.

## 22.11.8 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(vars[, title, limits])</code>	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**

<i>VTKFaceViewer</i> (vars[, title, limits])	Renders <i>MeshVariable</i> 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 23

## 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`

## 23.1 examples.benchmarking

### Modules

*examples.benchmarking.benchmarker*

*examples.benchmarking.size*

*examples.benchmarking.steps*

*examples.benchmarking.utils*

*examples.benchmarking.versions*

### 23.1.1 examples.benchmarking.benchmarker

### 23.1.2 examples.benchmarking.size

### 23.1.3 examples.benchmarking.steps

### 23.1.4 examples.benchmarking.utils

### 23.1.5 examples.benchmarking.versions

## 23.2 examples.cahnHilliard

### Modules

*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).

*examples.cahnHilliard.mesh2DCoupled*

Solve the Cahn-Hilliard problem in two dimensions.

*examples.cahnHilliard.mesh3D*

Solves the Cahn-Hilliard problem in a 3D cube

*examples.cahnHilliard.sphere*

Solves the Cahn-Hilliard problem on the surface of a sphere.

*examples.cahnHilliard.sphereDaemon*

*examples.cahnHilliard.tanh1D*

This example solves the Cahn-Hilliard equation given by,

*examples.cahnHilliard.test*

### 23.2.1 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  $\phi$  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  $\phi$  between 0 and 1. The gradient energy term  $\epsilon^2 \nabla^2 \phi$ , on the other hand, penalizes sharp changes of  $\phi$ . These two competing effects result in the segregation of  $\phi$  into domains of 0 and 1, separated by abrupt, but smooth, transitions. The parameters  $a$  and  $\epsilon$  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
>>> 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  $D * a^{**2} * (1 - 6 * \text{phi} * (1 - \text{phi}))$  onto the faces, where the diffusive flux is calculated, but we obtain somewhat more accurate results by performing a linear interpolation from  $\text{phi}$  at cell centers to  $\text{PHI}$  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.

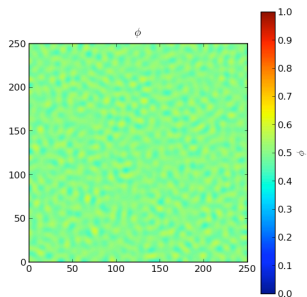
```
>>> PHI = phi.arithmeticFaceValue
>>> D = a = epsilon = 1.
>>> eq = (TransientTerm()
...      == DiffusionTerm(coeff=D * a**2 * (1 - 6 * PHI * (1 - PHI)))
...      - DiffusionTerm(coeff=(D, epsilon**2)))
```

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.

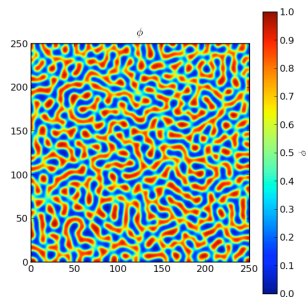
```
>>> dexp = -5
>>> elapsed = 0.
>>> if __name__ == "__main__":
...     duration = 1000.
... else:
...     duration = 1000.
```

```
>>> while elapsed < duration:
...     dt = min(100, numerix.exp(dexp))
...     elapsed += dt
...     dexp += 0.01
...     eq.solve(phi, dt=dt, solver=LinearLUSolver())
...     if __name__ == "__main__":
...         viewer.plot()
...     elif (max(phi.globalValue) > 0.7) and (min(phi.globalValue) < 0.3) and elapsed >=
↳ 10.:
...         break
```

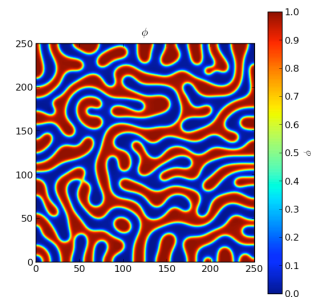
```
>>> print((max(phi.globalValue) > 0.7) and (min(phi.globalValue) < 0.3))
True
```



(a)  $t = 30$



(b)  $t = 100$



(c)  $t = 1000$

## 23.2.2 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, *PySparse* requires enormous amounts of memory, *Trilinos* cannot solve the coupled form, and *PETSc* cannot solve the vector form.

The spinodal decomposition phenomenon is a spontaneous separation of an initially homogeneous mixture into two distinct regions of different properties (spin-up/spin-down, component A/component B). It is a “barrierless” phase separation process, such that under the right thermodynamic conditions, any fluctuation, no matter how small, will tend to grow. This is in contrast to nucleation, where a fluctuation must exceed some critical magnitude before it will survive and grow. Spinodal decomposition can be described by the “Cahn-Hilliard” equation (also known as “conserved Ginsberg-Landau” or “model B” of Hohenberg & Halperin)

$$\frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \left( \frac{\partial f}{\partial \phi} - \epsilon^2 \nabla^2 \phi \right).$$

where  $\phi$  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  $\phi$  between 0 and 1. The gradient energy term  $\epsilon^2 \nabla^2 \phi$ , on the other hand, penalizes sharp changes of  $\phi$ . These two competing effects result in the segregation of  $\phi$  into domains of 0 and 1, separated by abrupt, but smooth, transitions. The parameters  $a$  and  $\epsilon$  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  $\psi$ ,  $\frac{\partial f}{\partial \phi}$ , 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
```

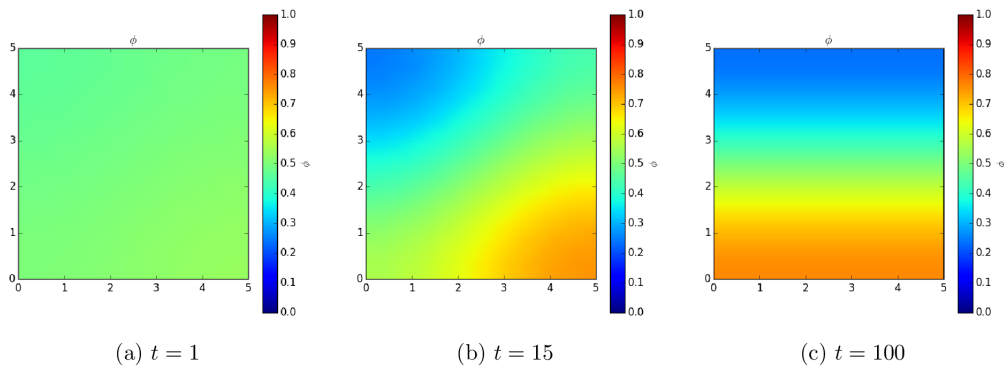
Because the evolution of a spinodal microstructure slows with time, we use exponentially increasing time steps to keep the simulation “interesting”. The *FiPy* user always has direct control over the evolution of their problem.

```
>>> 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  $\phi$  and  $\psi$  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
```

### 23.2.3 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:

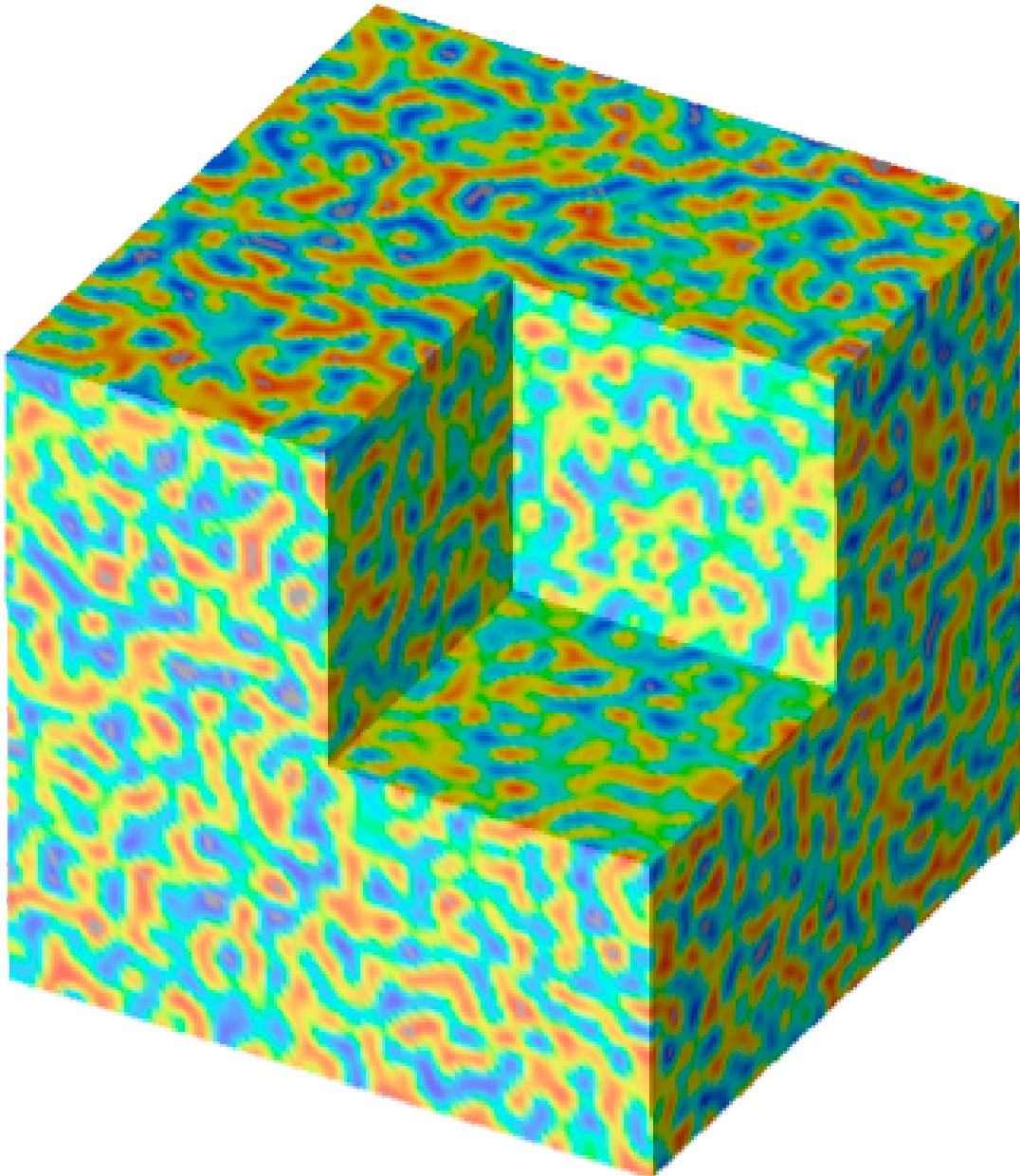
$$\frac{\partial \phi}{\partial t} = \nabla \cdot D a^2 [1 - 6\phi(1 - \phi)] \nabla \phi - \nabla \cdot D \nabla \epsilon^2 \nabla^2 \phi.$$

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()
```



### 23.2.4 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](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, zmax=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  $D * a^{**2} * (1 - 6 * \text{phi} * (1 - \text{phi}))$  onto the faces, where the diffusive flux is calculated, but we obtain somewhat more accurate results by performing a linear interpolation from  $\text{phi}$  at cell centers to  $\text{PHI}$  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)))

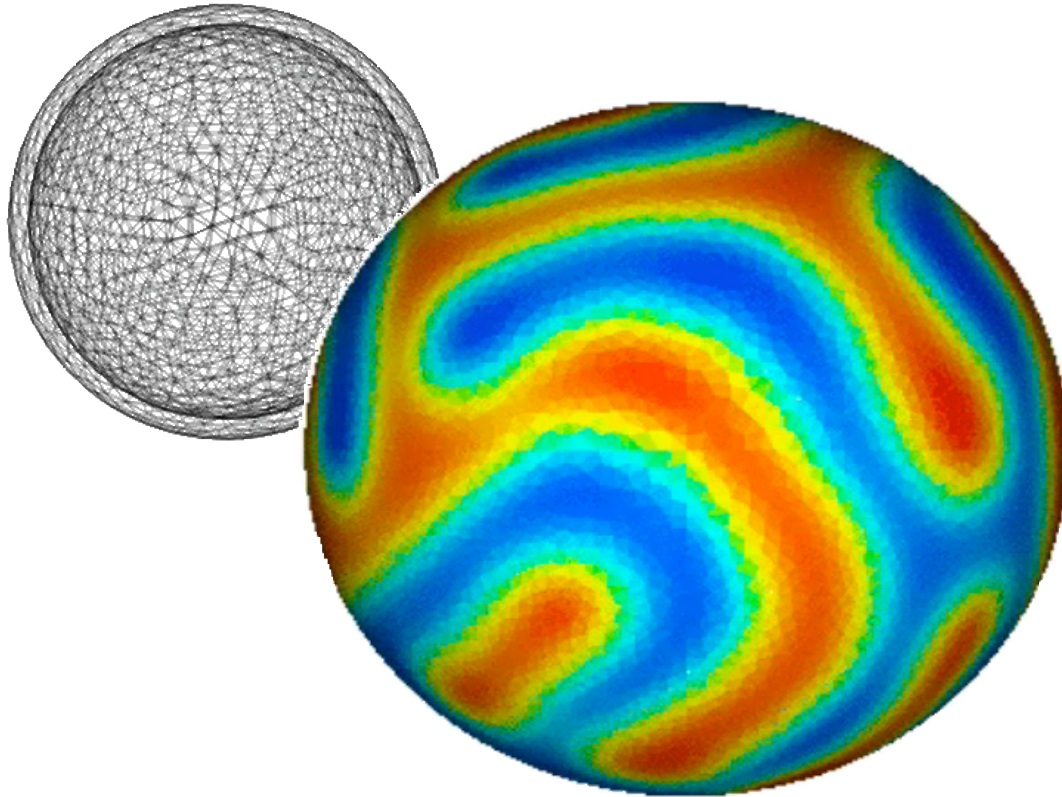
```

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()

```



### 23.2.5 examples.cahnHilliard.sphereDaemon

### 23.2.6 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

$$\left. \begin{array}{l} \phi = \frac{1}{2} \\ \frac{\partial^2 \phi}{\partial x^2} = 0 \end{array} \right\} \text{ on } x = 0$$

and

$$\left. \begin{array}{l} \phi = 1 \\ \frac{\partial^2 \phi}{\partial x^2} = 0 \end{array} \right\} \text{ on } x = L$$

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
```

```
>>> import fipy.solvers.solver
>>> if fipy.solvers.solver_suite in ['pysparse', 'pyamgx']:
...     solver = LinearLUSolver(tolerance=1e-15, iterations=100)
... else:
...     solver = DefaultSolver()
```

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
>>> from builtins import range
>>> 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
```

### 23.2.7 examples.cahnHilliard.test

## 23.3 examples.chemotaxis

### Modules

---

<i>examples.chemotaxis.input</i>	Input file for chemotaxis modeling.
<i>examples.chemotaxis.input2D</i>	Input file for chemotaxis modeling.
<i>examples.chemotaxis.parameters</i>	Input file for parameters
<i>examples.chemotaxis.test</i>	

---



### 23.3.1 examples.chemotaxis.input

Input file for chemotaxis modeling.

Here are some test cases for the model.

```
>>> from __future__ import division
```

```
>>> from builtins import input
>>> from builtins import range
>>> 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")
```

### 23.3.2 examples.chemotaxis.input2D

Input file for chemotaxis modeling.

Here are some test cases for the model.

```
>>> from __future__ import division
```

```
>>> from builtins import input
>>> from builtins import range
>>> 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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
>>> 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")

```

### 23.3.3 examples.chemotaxis.parameters

Input file for parameters

### 23.3.4 examples.chemotaxis.test

## 23.4 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>	

### 23.4.1 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 <a href="http://www.gre.ac.uk/~physica/phy2.12/theory/node173.htm">http://www.gre.ac.uk/~physica/phy2.12/theory/node173.htm</a>

### 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.

```
>>> from builtins import range
>>> 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
```

## 23.4.2 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 <a href="#">examples.convection.exponential1D.mesh1D</a> 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))
```

```
>>> eq.solve(var = var,
...          solver=DefaultAsymmetricSolver(iterations=10000))
```

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))
1
```

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

### 23.4.3 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()
```

## 23.4.4 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 <code>examples.convection.exponential1D.mesh1D</code> 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-15, 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()
```

## 23.4.5 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 <a href="#">examples.convection.exponential1D.mesh1D</a> 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 <a href="#">examples.convection.exponential1D.mesh1D</a> 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

$$\phi = \exp \frac{u}{D} (r_1 - r) \left( \frac{\text{ei} \frac{ur_0}{D} - \text{ei} \frac{ur}{D}}{\text{ei} \frac{ur_0}{D} - \text{ei} \frac{ur_1}{D}} \right)$$

or

```
>>> axis = 0
>>> try:
...     from scipy.special import expi
...     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=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()
```

### 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))
```

```
>>> eq.solve(var = var)
```

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))
1
```

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

### 23.4.6 examples.convection.peclet

This example tests diffusion-convection for increasing Péclet numbers. This test case has been introduced because *LinearCGSSolver* was not working with Péclet numbers over 1. *LinearLUSolver* is now the default for *ConvectionTerm*. For  $nx = 1000$  the *LinearGMRESSolver* does not work.

```
>>> 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
>>> from builtins import str
>>> 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]
```

### 23.4.7 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]
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
```

(continues on next page)



(continued from previous page)

```
>>> 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]
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
```

(continues on next page)

(continued from previous page)

```
>>> analyticalArray = CC / DD
>>> print(var.allclose(analyticalArray, rtol = 1e-2, atol = 1e-2))
1
```

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

### 23.4.8 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
```

### 23.4.9 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  $\phi$  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")
```

## 23.4.10 examples.convection.test

## 23.5 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 $nx = 10$ .

### 23.5.1 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  $\Gamma'$  such that

$$R = \begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix}$$

and

$$\Gamma = R \Gamma' R^T$$

In the case of a point source at (0, 0) a reference solution is given by,

$$\phi(X, Y, t) = Q \frac{\exp\left(-\frac{1}{4t} \left(\frac{X^2}{\Gamma'_{00}} + \frac{Y^2}{\Gamma'_{11}}\right)\right)}{4\pi t \sqrt{\Gamma'_{00}\Gamma'_{11}}}$$

where  $(X, Y)^T = R(x, y)^T$  and  $Q$  is the initial mass.

```
>>> from fipy import CellVariable, Gmsh2D, Viewer, TransientTerm, DiffusionTermCorrection
>>> from fipy.tools import serialComm, numerix
```

Import a mesh previously created using *Gmsh*.

```
>>> import os
>>> mesh = Gmsh2D(os.path.splitext(__file__)[0] + '.msh', communicator=serialComm)
```

Set the center-most cell to have a value.

```
>>> var = CellVariable(mesh=mesh, hasOld=1)
>>> x, y = mesh.cellCenters
>>> var[numerix.argmin(x**2 + y**2)] = 1.
```

Choose an orientation for the anisotropy.

```
>>> theta = numerix.pi / 4.
>>> rotationMatrix = numerix.array(((numerix.cos(theta), numerix.sin(theta)), \
...                                 (-numerix.sin(theta), numerix.cos(theta))))
>>> gamma_prime = numerix.array(((0.2, 0.), (0., 1.)))
>>> DOT = numerix.NUMERIX.dot
>>> gamma = DOT(DOT(rotationMatrix, gamma_prime), numerix.transpose(rotationMatrix))
```

Make the equation, viewer and solve.

```
>>> eqn = TransientTerm() == DiffusionTermCorrection((gamma,))
```

```
>>> if __name__ == '__main__':
...     viewer = Viewer(var, datamin=0.0, datamax=0.001)
```

```
>>> mass = float(var.cellVolumeAverage * numerix.sum(mesh.cellVolumes))
>>> time = 0
>>> dt=0.00025
```

```
>>> from builtins import range
>>> for i in range(20):
...     var.updateOld()
...     res = 1.
...
...     while res > 1e-2:
...         res = eqn.sweep(var, dt=dt)
...
...     if __name__ == '__main__':
...         viewer.plot()
...     time += dt
```

Compare with the analytical solution (within 5% accuracy).

```
>>> X, Y = numerix.dot(mesh.cellCenters, CellVariable(mesh=mesh, rank=2,
↳value=rotationMatrix))
>>> solution = mass * numerix.exp(-(X**2 / gamma_prime[0][0] + Y**2 / gamma_prime[1][1])
↳/ (4 * time)) / (4 * numerix.pi * time * numerix.sqrt(gamma_prime[0][0] * gamma_
↳prime[1][1]))
>>> print(max(abs((var - solution) / max(solution))) < 0.08)
True
```

## 23.5.2 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.)
```

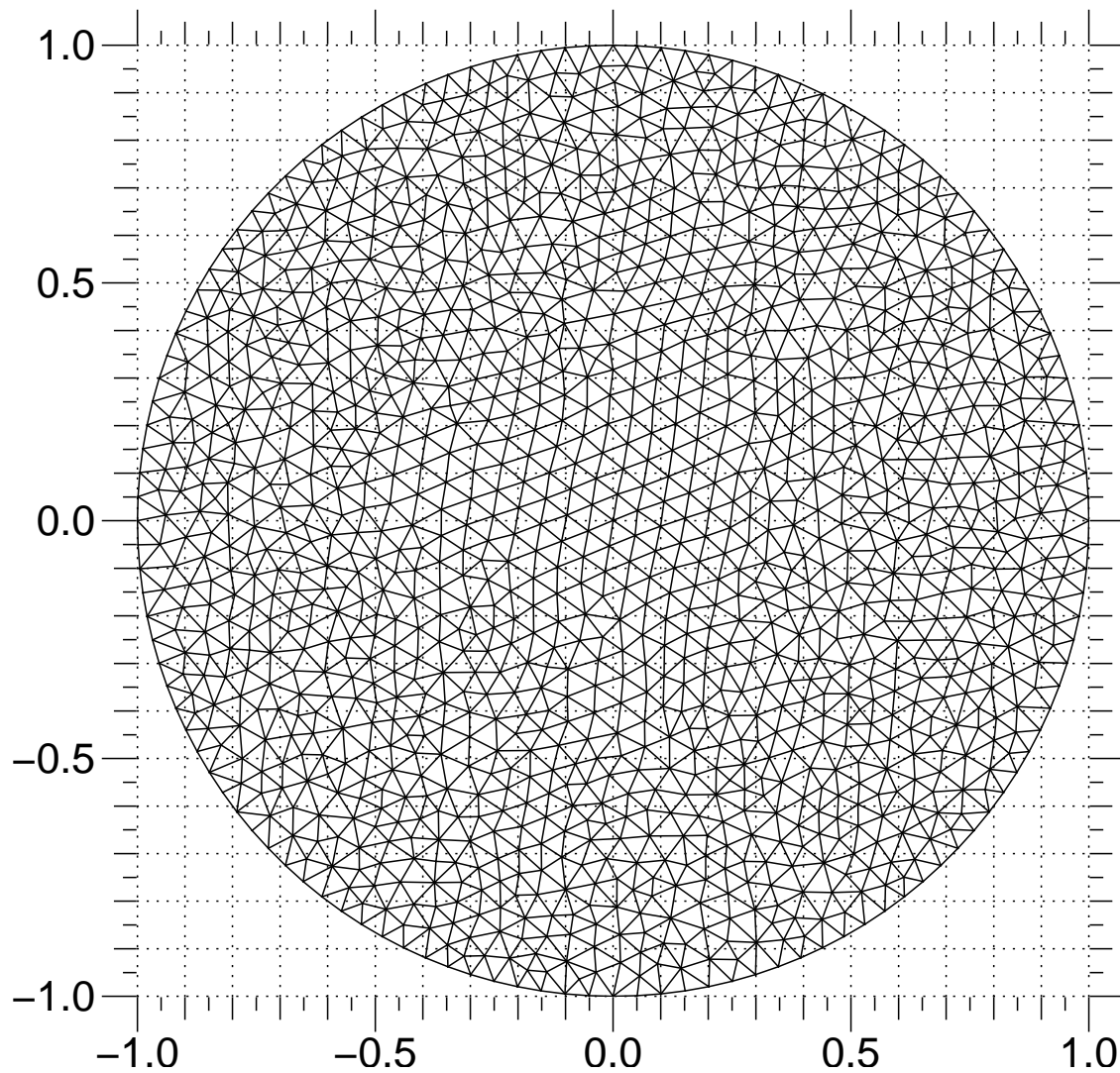
We can now create a Viewer to see the mesh

```
>>> 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)
```

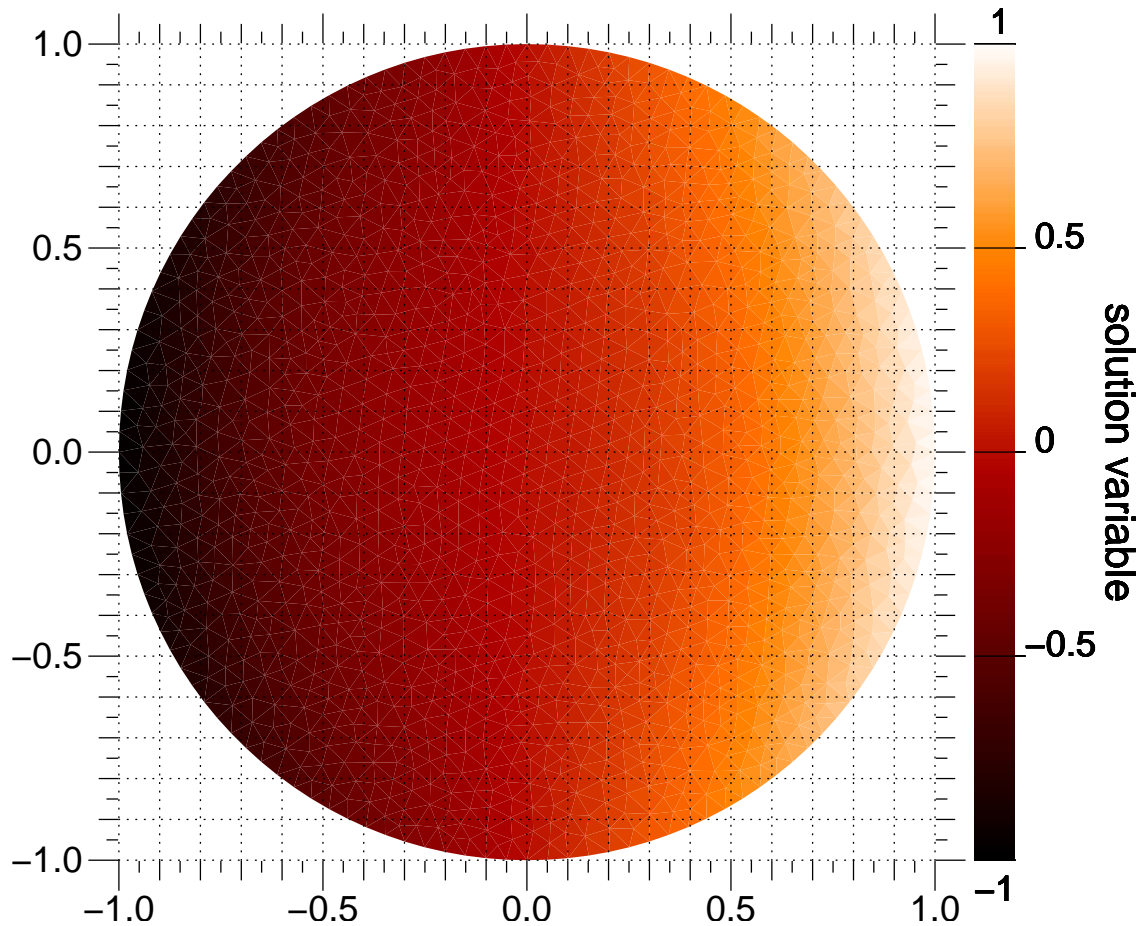
We first step through the transient problem



```

>>> timeStepDuration = 10 * 0.9 * cellSize**2 / (2 * D)
>>> steps = 10
>>> from builtins import range
>>> 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	0.229687917881182	0.0755414402612554	0.964844363287199	-0.000757854476106331
0.0442864953037566	0.773936613923853	0.79191893162384	0.0375859836421991	-0.205560697612547
0.0246775505084069	0.723540342405813	0.771959648896982	0.020853932412869	-0.182589694334729

(continues on next page)

(continued from previous page)

```

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.
↳ 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.

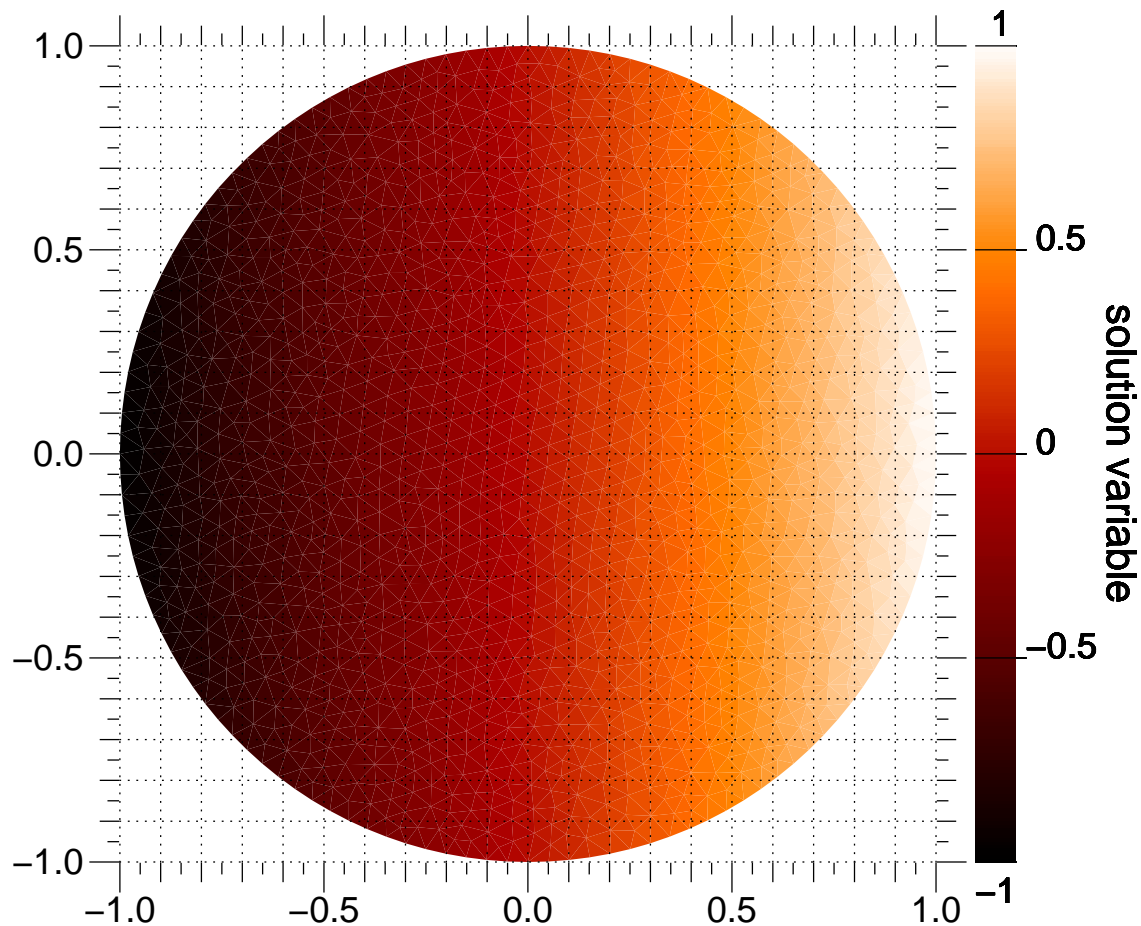
```
>>> DiffusionTerm(coeff=D).solve(var=phi)
```

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...")
```



### 23.5.3 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
```

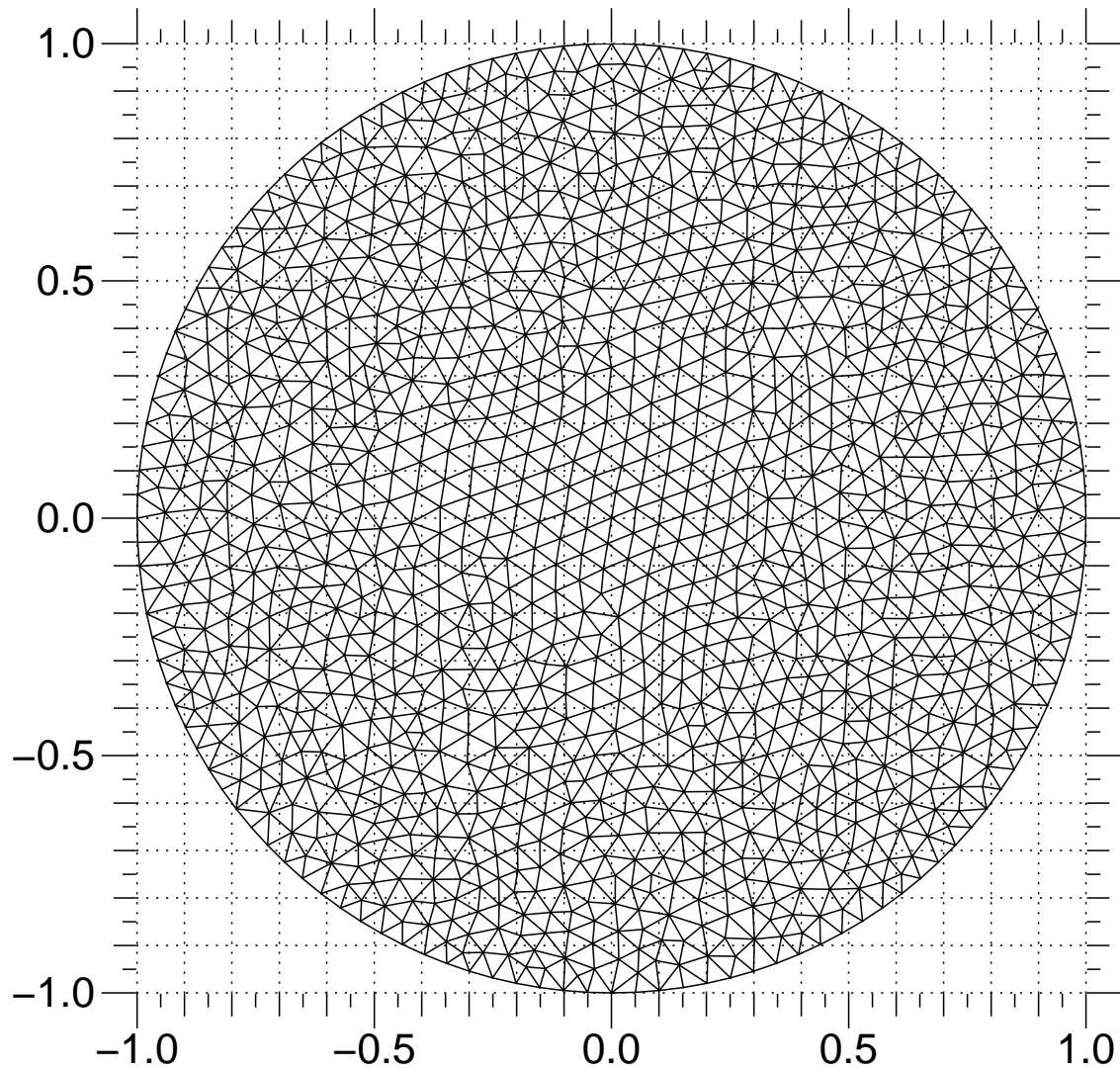
```
>>> mesh = Gmsh2D('''
...     cellSize = %(cellSize)g;
...     radius = %(radius)g;
...     Point(1) = {0, 0, 0, cellSize};
...     Point(2) = {-radius, 0, 0, cellSize};
...     Point(3) = {0, radius, 0, cellSize};
...     Point(4) = {radius, 0, 0, cellSize};
...     Point(5) = {0, -radius, 0, cellSize};
...     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)
```

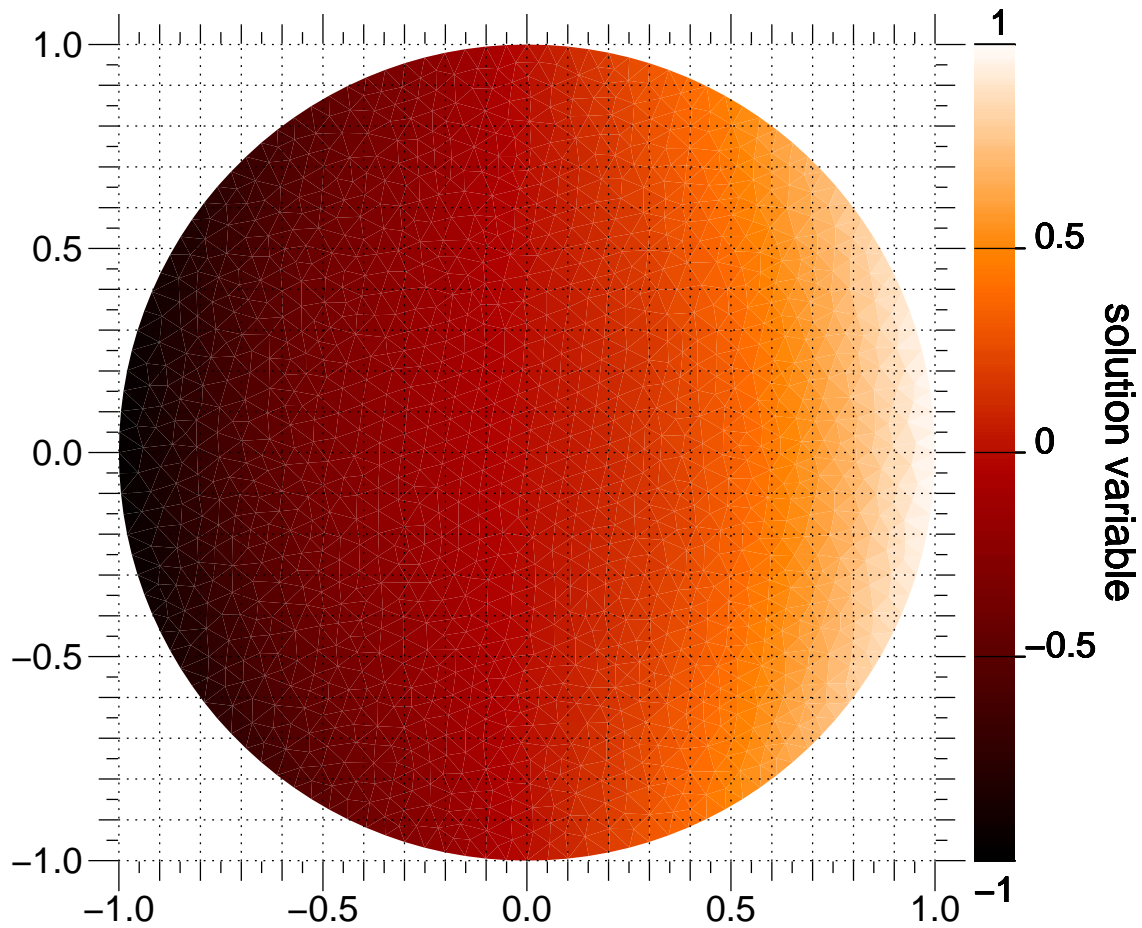
We first step through the transient problem

```
>>> timeStepDuration = 10 * 0.9 * cellSize**2 / (2 * D)
>>> steps = 10
>>> from builtins import range
>>> for step in range(steps):
...     eq.solve(var=phi,
...               dt=timeStepDuration)
...     if viewer is not None:
```

(continues on next page)

(continued from previous page)

```
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	0.0375859836421991	-0.	
0.0442864953037566	0.79191893162384	0.020853932412869	-0.	
0.773936613923853	-0.205560697612547	0.203035857140125	-0.	
0.0246775505084069	0.771959648896982	0.0401235242511506	-0.	
0.723540342405813	-0.182589694334729	-0.00412895434496877	-0.	
0.223345558247991	-0.807931073108895	-0.00412895434496877	-0.	
0.777466238738658	0.0401235242511506	-0.00412895434496877	-0.	
-0.00726763301939488	-0.775978916110686	-0.00412895434496877	-0.	
0.650055516507232	-0.183112882869288	-0.012771874945585	-0.	
-0.0220279064527904	-0.187563765977912	-0.012771874945585	-0.	
0.35707168379437	-0.056072788439713			

(continues on next page)

(continued from previous page)

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.

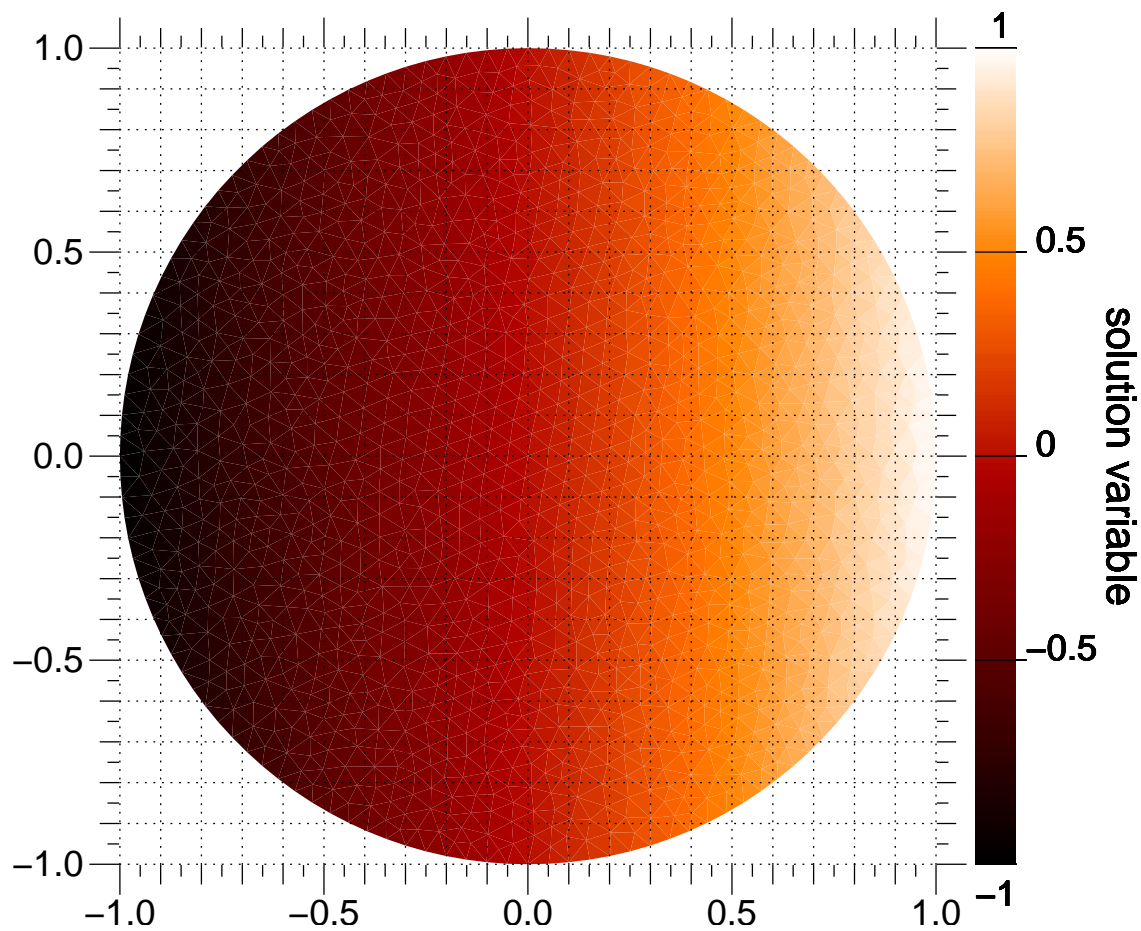
```
>>> DiffusionTerm(coeff=D).solve(var=phi)
```

The values at the elements should be equal to their  $x$  coordinate

```
>>> print(phi.allclose(x, atol = 0.035))
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...")
```





### 23.5.4 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))
```

```

>>> from builtins import range
>>> for t in range(100):
...     v0.updateOld()
...     v1.updateOld()
...     res0 = res1 = 1e100
...     while max(res0, res1) > 0.1:
...         res0 = eq0.sweep(var=v0, dt=1e-5)
...         res1 = eq1.sweep(var=v1, dt=1e-5)
...     if t % 10 == 0:
...         vi.plot()

```

The uncoupled method still works, but it can be advantageous to solve the two equations simultaneously. In this case, by coupling the equations, we can eliminate the explicit sources and dramatically increase the time steps:

```

>>> v0.value = 0.5
>>> v1.value = 0.5

```

```

>>> eqn0 = TransientTerm(var=v0) == DiffusionTerm(0.01, var=v0) - DiffusionTerm(1,
↳var=v1)
>>> eqn1 = TransientTerm(var=v1) == DiffusionTerm(1, var=v0) + DiffusionTerm(0.01,
↳var=v1)

```

```

>>> eqn = eqn0 & eqn1

```

```

>>> from builtins import range
>>> for t in range(1):
...     v0.updateOld()
...     v1.updateOld()
...     eqn.solve(dt=1.e-3)
...     vi.plot()

```

It is also possible to pose the same equations in vector form:

```

>>> v = CellVariable(mesh=m, hasOld=True, value=[[0.5], [0.5]], elementshape=(2,))

```

```

>>> v.constrain([[0], [1]], m.facesLeft)
>>> v.constrain([[1], [0]], m.facesRight)

```

```

>>> eqn = TransientTerm([[1, 0],
...                       [0, 1]]) == DiffusionTerm([[0.01, -1],
...                                                   [1, 0.01]])

```

```

>>> vi = Viewer((v[0], v[1]))

```

```

>>> from builtins import range
>>> for t in range(1):
...     v.updateOld()
...     eqn.solve(var=v, dt=1.e-3)
...     vi.plot()

```

Whether you pose your problem in coupled or vector form should be dictated by the underlying physics. If  $v_0$  and  $v_1$  represent the concentrations of two conserved species, then it is natural to write two separate governing equations

and to couple them. If they represent two components of a vector field, then the vector formulation is obviously more natural. FiPy will solve the same matrix system either way.

### 23.5.5 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  $\phi$ ,

```
>>> potential = CellVariable(mesh=mesh, name='potential', value=0.)
```

the permittivity  $\epsilon$ ,

```
>>> permittivity = 1
```

the concentration  $C_j$  of the  $j^{\text{th}}$  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  $\rho$ ,

```
>>> 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

$$\psi(x) = \frac{x^2}{2} - 2x$$

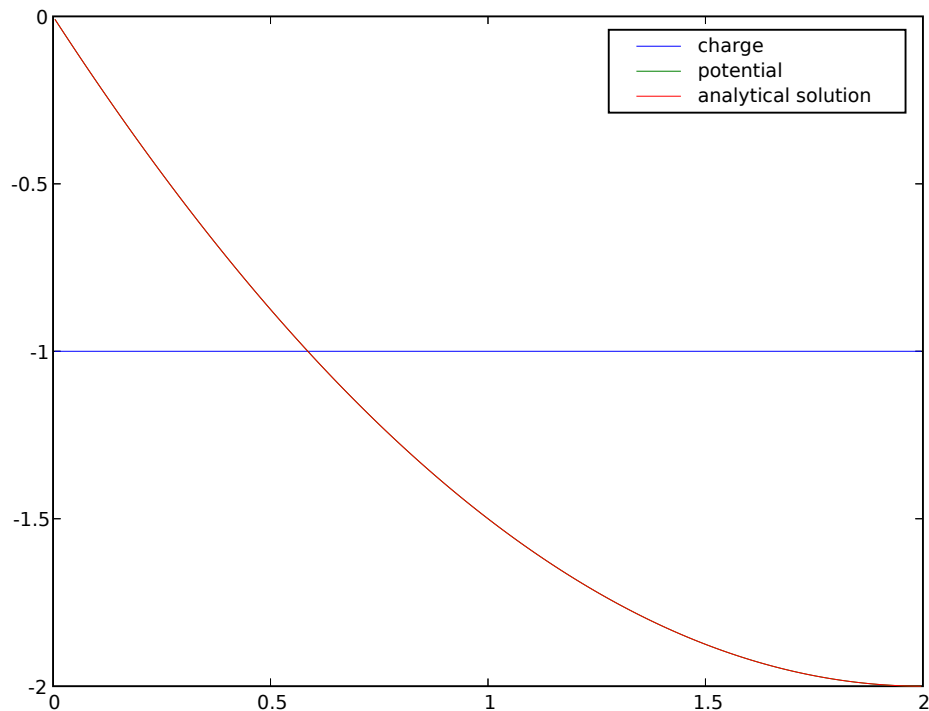
```
>>> x = mesh.cellCenters[0]
>>> analytical = CellVariable(mesh=mesh, name="analytical solution",
...                           value=(x**2)/2 - 2*x)
```

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

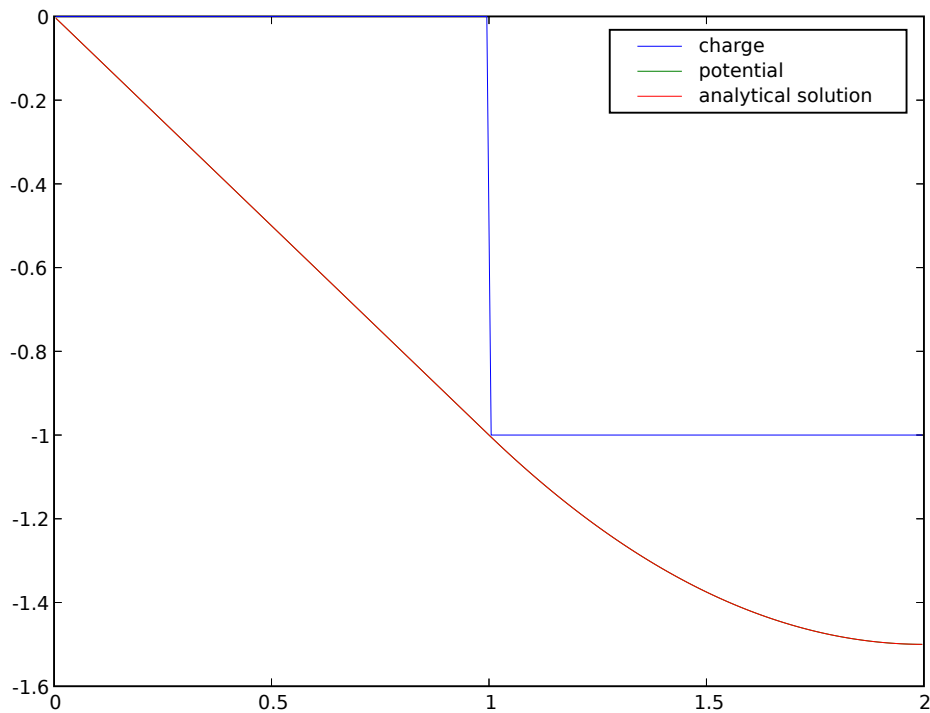
$$\psi(x) = \begin{cases} -x & \text{for } x \leq L/2, \\ \frac{(x-1)^2}{2} - x & \text{for } x > L/2. \end{cases}$$

```
>>> 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

$$\psi(x) = \begin{cases} \frac{x^2}{2} - x & \text{for } x \leq L/2, \\ -\frac{1}{2} & \text{for } x > L/2. \end{cases}$$

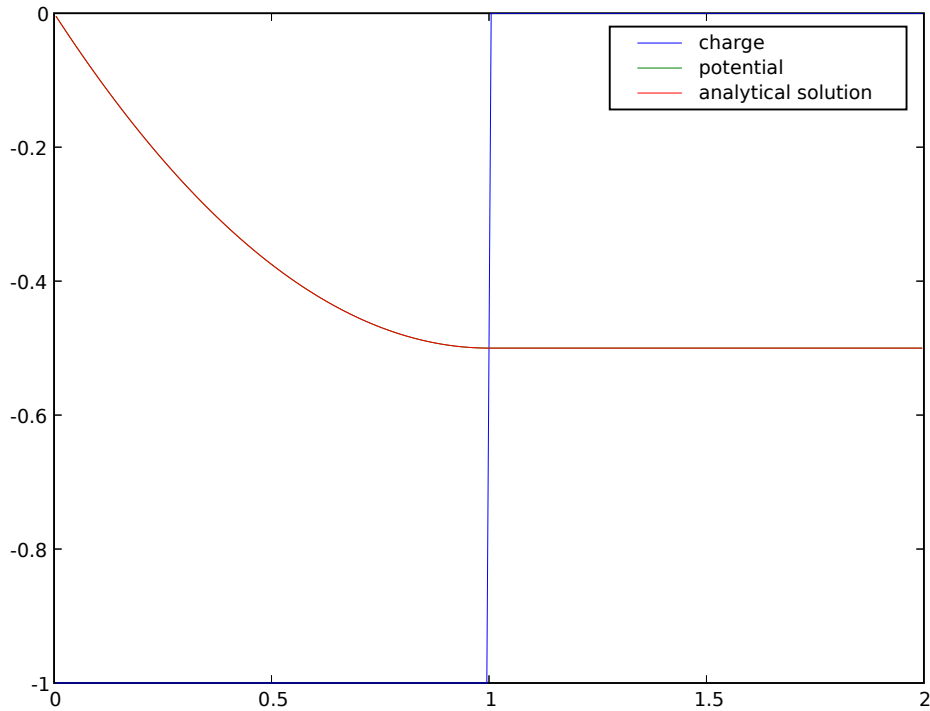
We again verify that the correct equilibrium is attained

```
>>> 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()
```



### 23.5.6 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
>>> from builtins import range
>>> 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
```

(continues on next page)



(continued from previous page)

```

>>> epsi = x / numerix.sqrt(t * diffusionCoeff)
>>> from scipy.special import erf
>>> analyticalArray = erf(epsi/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  $\phi = 1 - \operatorname{erf}((L - x)/2\sqrt{Dt})$ , 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)
```

```
>>> from builtins import range
>>> 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.
>>> valueRight = 1.
>>> timeStepDuration = 0.02
>>> steps = 10
```

A loop is required to execute the necessary time steps:

```
>>> from builtins import range
>>> 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
```

### 23.5.7 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,

$$\frac{\partial \phi}{\partial t} = D \nabla^2 \phi. \quad (23.1)$$

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. (23.1), you most likely attempted “explicit finite differencing” with code something like:

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

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

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

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

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

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

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

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

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

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

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

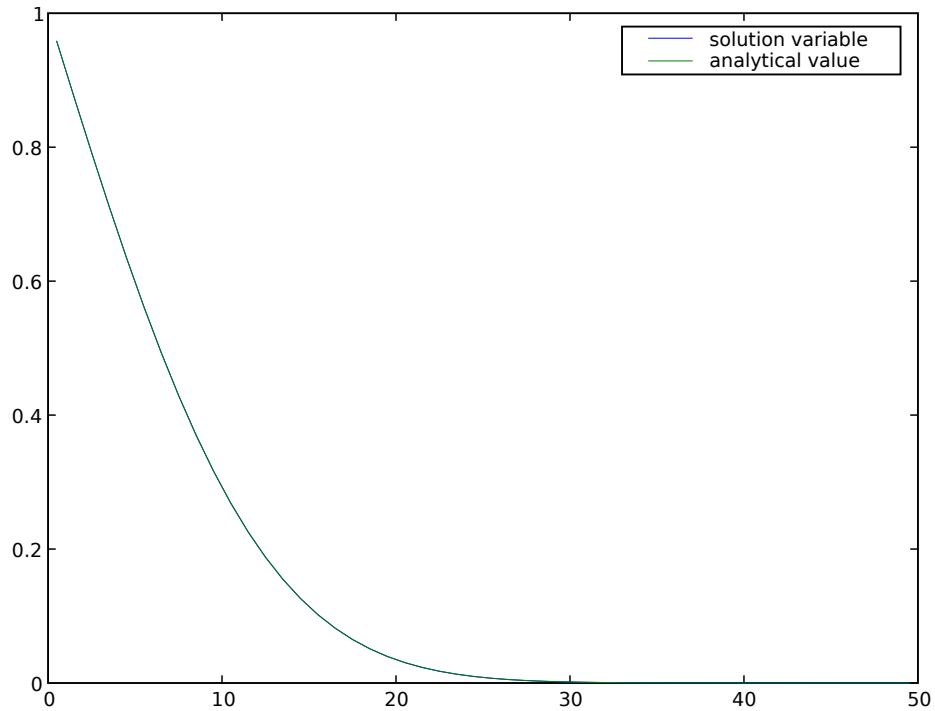
```
>>> try:
...     from scipy.special import erf
...     phiAnalytical.setValue(1 - erf(x / (2 * numerix.sqrt(D * t))))
... except ImportError:
...     print("The SciPy library is not available to test the solution to \
... the transient diffusion equation")
```

We then solve the equation by repeatedly looping in time:

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

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

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



Although explicit finite differences are easy to program, we have just seen that this 1D transient diffusion problem is limited to taking rather small time steps. If, instead, we represent Eq. (23.1) as:

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

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

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

reset the problem

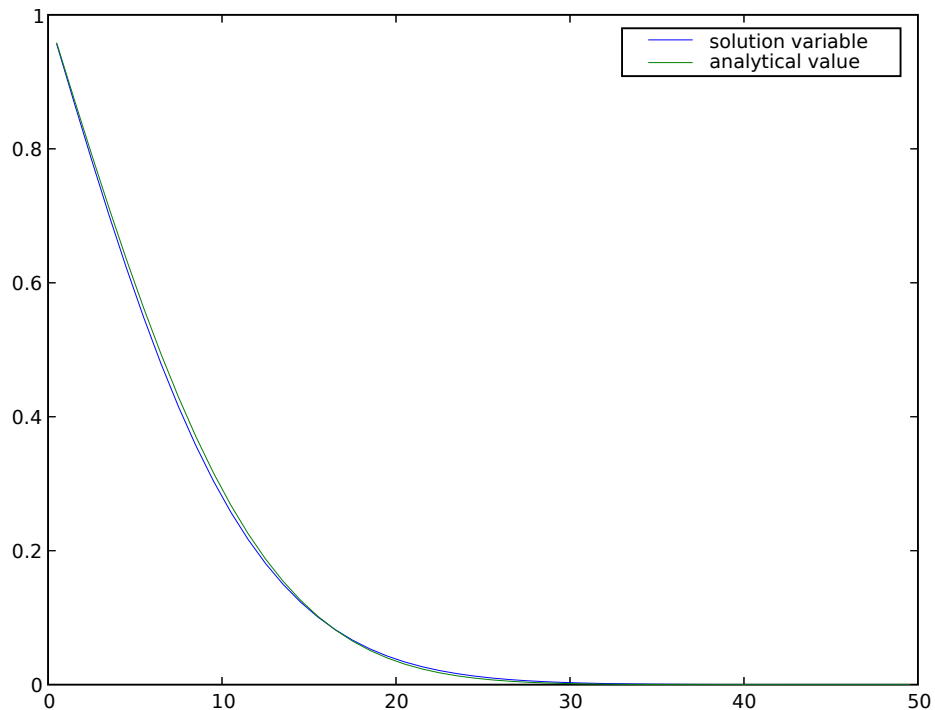
```
>>> phi.setValue(valueRight)
```

and rerun with much larger time steps

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

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

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



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

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

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

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

We again reset the problem

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

and apply the Crank-Nicholson scheme until the end, when we apply one step of the fully implicit scheme to drive down the error (see, *e.g.*, section 19.2 of [24]).

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

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

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

---

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

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

is represented in *FiPy* by

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

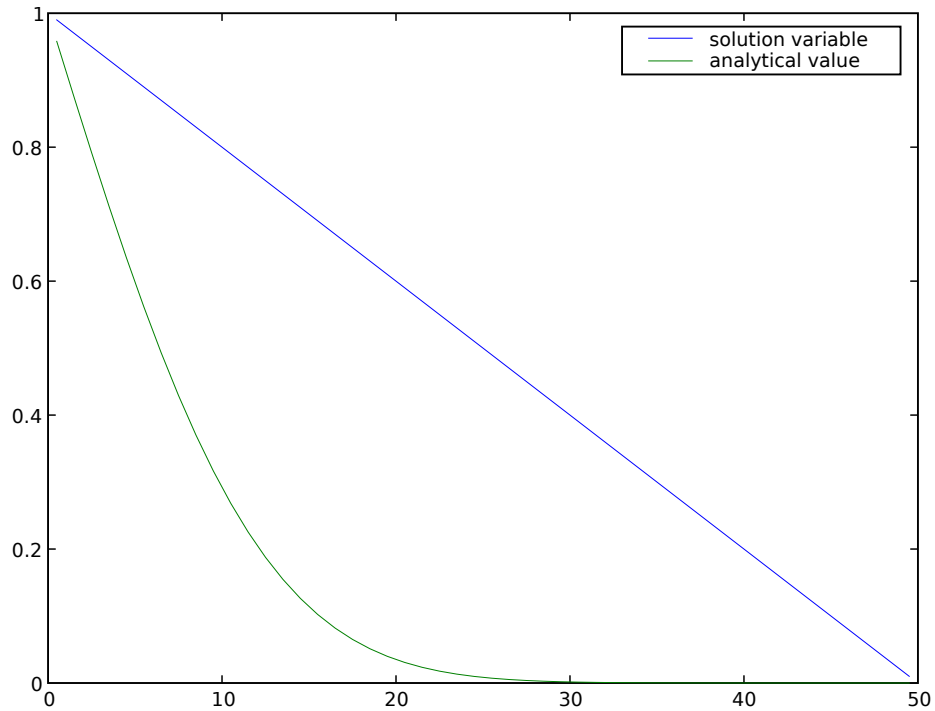
```
>>> if __name__ == '__main__':
...     viewer.plot()
```

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

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

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





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

For example, to have

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

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

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

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

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

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

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

```
>>> dt = .1
>>> while time() < 15:
...     time.setValue(time() + dt)
```

(continues on next page)

(continued from previous page)

```

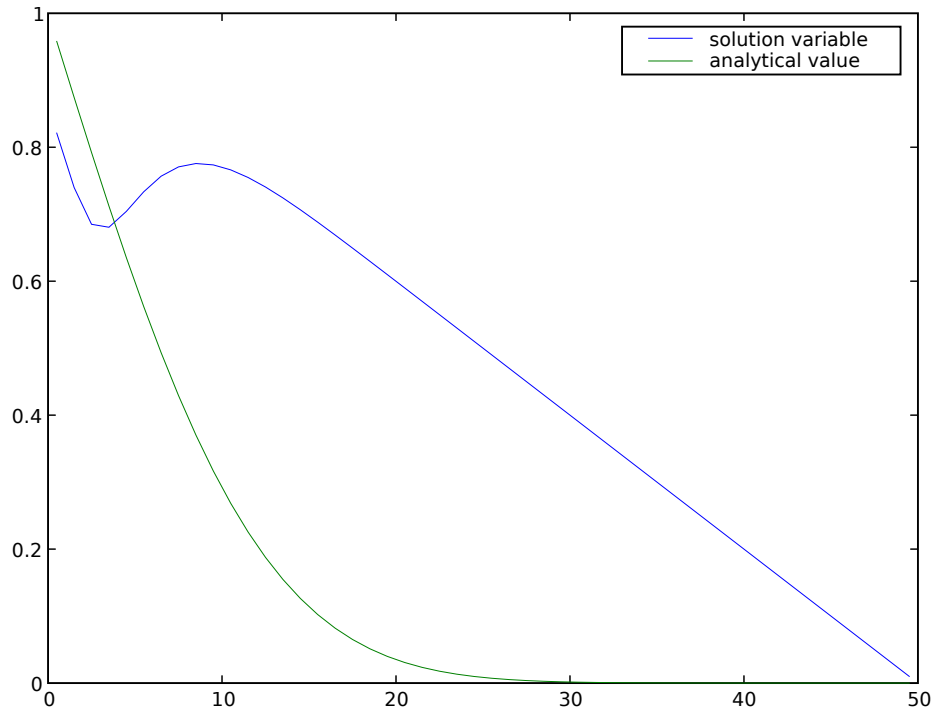
...     eqI.solve(var=phi, dt=dt)
...     if __name__ == '__main__':
...         viewer.plot()

```

```

>>> 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

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

The analytical solution is simply

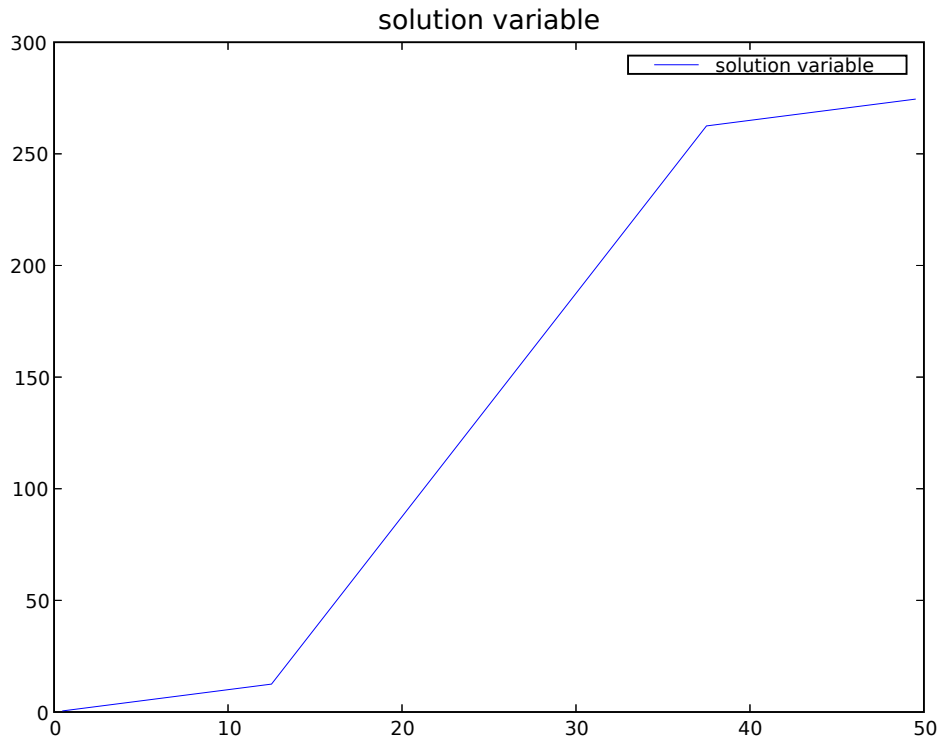
$$\phi = \begin{cases} x & \text{for } 0 < x < L/4, \\ 10x - 9L/4 & \text{for } L/4 \leq x < 3L/4, \\ x + 18L/4 & \text{for } 3L/4 \leq x < L, \end{cases}$$

or

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

And finally, we can plot the result

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



Note that for problems involving heat transfer and other similar conservation equations, it is important to ensure that we begin with the correct form of the equation. For example, for heat transfer with  $\phi$  representing the temperature,

$$\frac{\partial}{\partial t} (\rho \hat{C}_p \phi) = \nabla \cdot [k \nabla \phi].$$

With constant and uniform density  $\rho$ , heat capacity  $\hat{C}_p$  and thermal conductivity  $k$ , this is often written like Eq. (23.1), 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)
>>> eqT.solve(var=phiT)

```

Comparing to the correct analytical solution,  $\phi = x$

```

>>> x = mesh.cellCenters[0]
>>> phiAnalytical.setValue(x)
>>> print(phiT.allclose(phiAnalytical, atol = 1e-8, rtol = 1e-8))
1

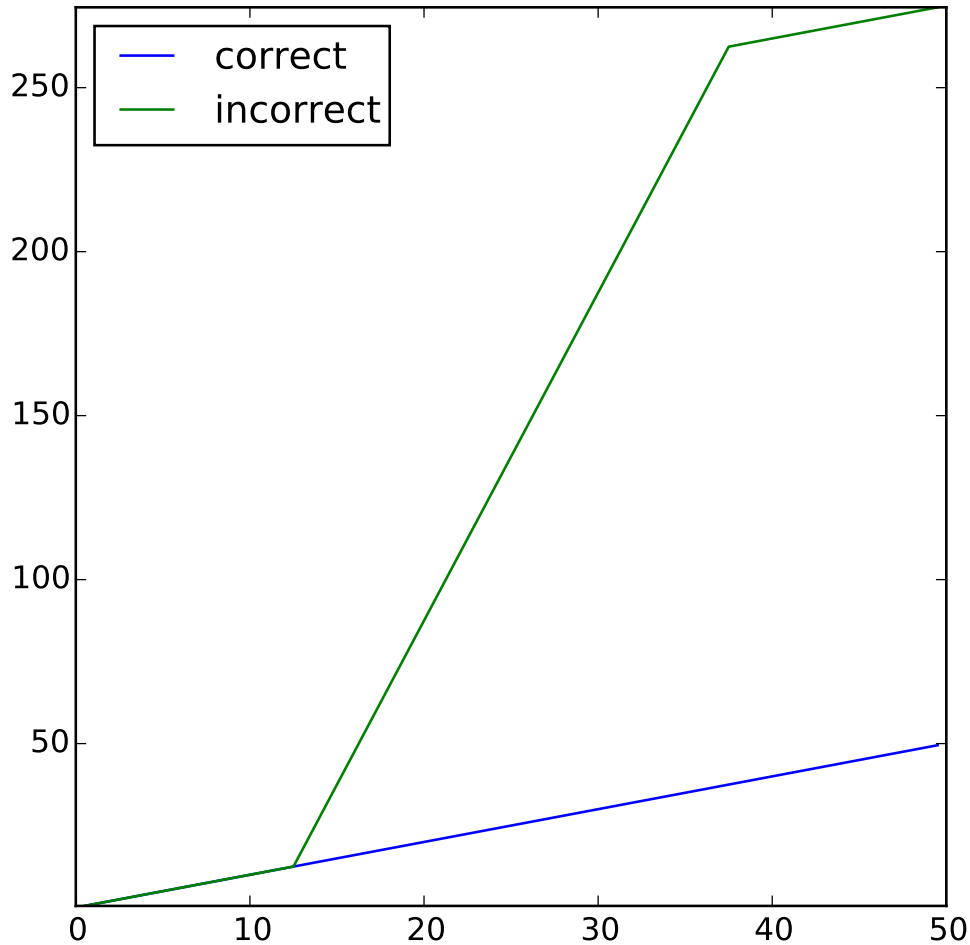
```

and finally, plot

```

>>> from fipy import input
>>> if __name__ == '__main__':
...     Viewer(vars=(phiT, phiF)).plot()
...     input("Non-uniform thermal conductivity. Press <return> to proceed...")

```



Often, the diffusivity is not only non-uniform, but also depends on the value of the variable, such that

$$\frac{\partial \phi}{\partial t} = \nabla \cdot [D(\phi) \nabla \phi]. \quad (23.2)$$

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. (23.2) as

```

>>> D0 = 1.
>>> eq = TransientTerm() == DiffusionTerm(coeff=D0 * (1 - phi[0]))

```

---

**Note:** Because of the non-linearity, the Crank-Nicholson scheme does not work for this problem.

---

We apply the same boundary conditions that we used for the uniform diffusivity cases

```

>>> phi[0].constrain(valueRight, mesh.facesRight)
>>> phi[0].constrain(valueLeft, mesh.facesLeft)

```

Although this problem does not have an exact transient solution, it can be solved in steady-state, with

$$\phi(x) = 1 - \sqrt{\frac{x}{L}}$$

```

>>> x = mesh.cellCenters[0]
>>> phiAnalytical.setValue(1. - numerix.sqrt(x/L))

```

We create a viewer to compare the different numbers of sweeps with the analytical solution from before.

```

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

```

As described above, an inner “sweep” loop is generally required for the solution of non-linear or multiple equation sets. Often a conditional is required to exit this “sweep” loop given some convergence criteria. Instead of using the `solve()` method equation, when sweeping, it is often useful to call `sweep()` instead. The `sweep()` method behaves the same way as `solve()`, but returns the residual that can then be used as part of the exit condition.

We now repeatedly run the problem with increasing numbers of sweeps.

```

>>> from fipy import input
>>> from builtins import range
>>> for sweeps in range(1, 5):
...     phi[0].setValue(valueRight)
...     for step in range(steps):
...         # only move forward in time once per time step
...         phi[0].updateOld()
...
...         # but "sweep" many times per time step
...         for sweep in range(sweeps):
...             res = eq.sweep(var=phi[0],
...                             dt=timeStepDuration)
...             if __name__ == '__main__':
...                 viewer.plot()
...
...         # copy the final result into the appropriate display variable
...         phi[sweeps].setValue(phi[0])
...         if __name__ == '__main__':
...             viewer.plot()
...             input("Implicit variable diffusivity. %d sweep(s). \
... Residual = %f. Press <return> to proceed..." % (sweeps, (abs(res))))

```

As can be seen, sweeping does not dramatically change the result, but the “residual” of the equation (a measure of how accurately it has been solved) drops about an order of magnitude with each additional sweep.

**Attention:** Choosing an optimal balance between the number of time steps, the number of sweeps, the number of solver iterations, and the solver tolerance is more art than science and will require some experimentation on your part for each new problem.

Finally, we can increase the number of steps to approach equilibrium, or we can just solve for it directly

```

>>> eq = DiffusionTerm(coeff=D0 * (1 - phi[0]))

```

```

>>> phi[0].setValue(valueRight)
>>> res = 1e+10
>>> while res > 1e-6:
...     res = eq.sweep(var=phi[0],
...                     dt=timeStepDuration)

```

```

>>> print(phi[0].allclose(phiAnalytical, atol = 1e-1))
1

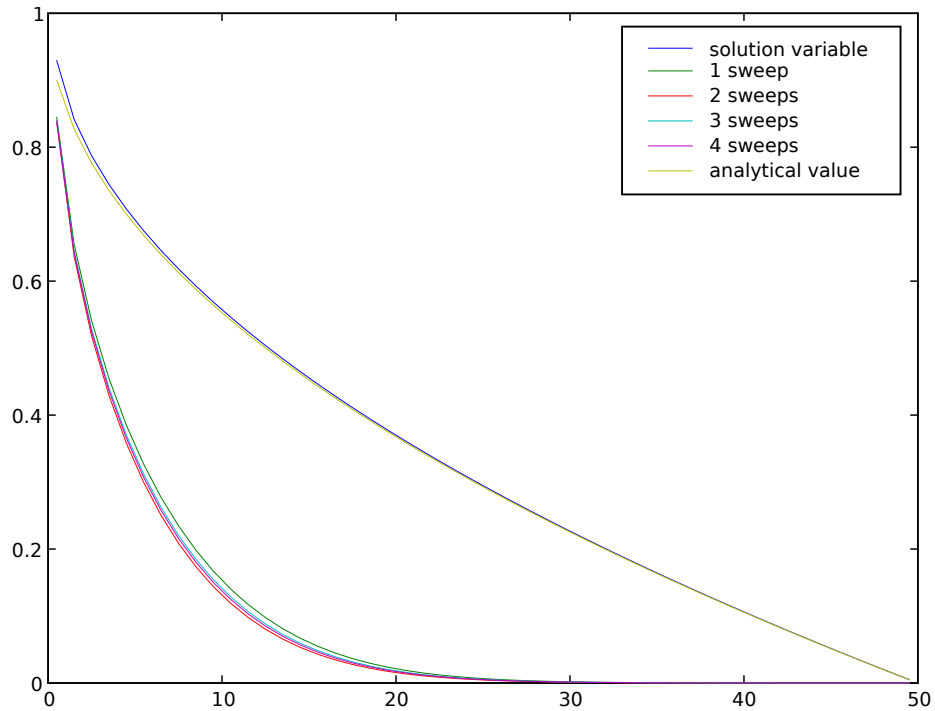
```

```

>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Implicit variable diffusivity - steady-state. \
... Press <return> to proceed...")

```

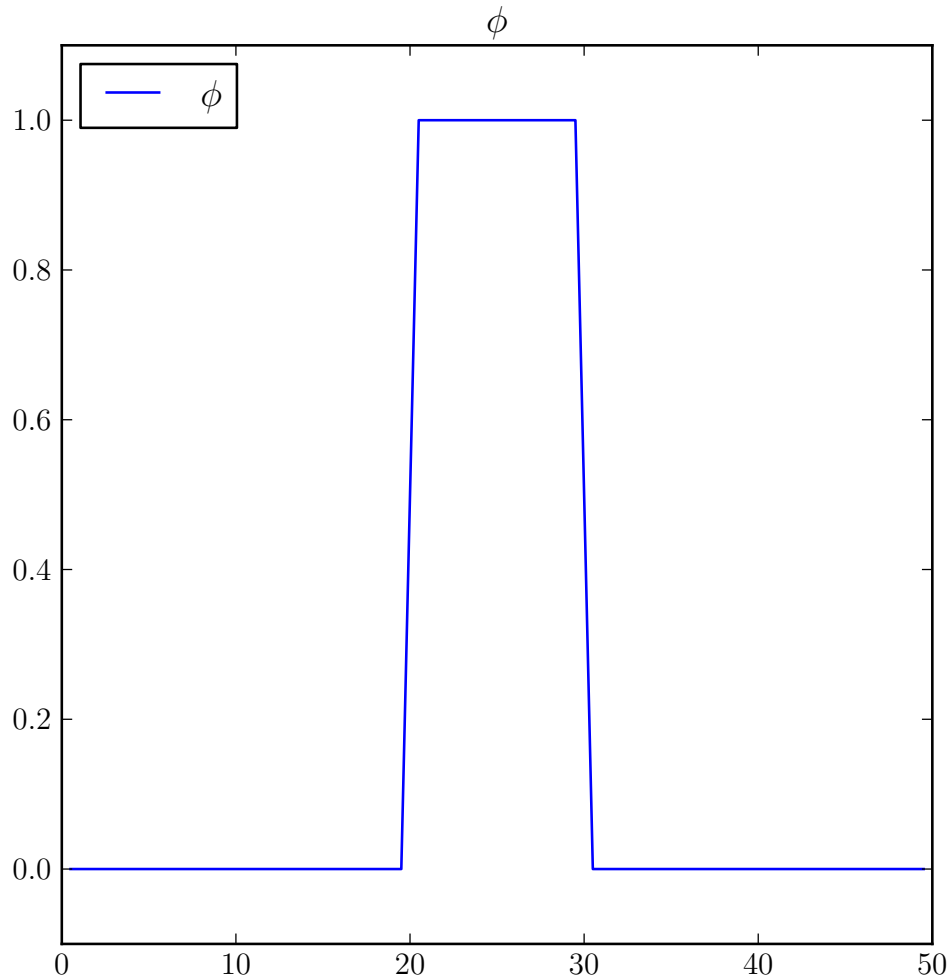




Fully implicit solutions are not without their pitfalls, particularly in steady state. Consider a localized block of material diffusing in a closed box.

```
>>> phi = CellVariable(mesh=mesh, name=r"$\phi$")
```

```
>>> phi.value = 0.
>>> phi.setValue(1., where=(x > L/2. - L/10.) & (x < L/2. + L/10.))
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=phi, datamin=-0.1, datamax=1.1)
```



We assign no explicit boundary conditions, leaving the default no-flux boundary conditions, and solve

$$\partial\phi/\partial t = \nabla \cdot (D\nabla\phi)$$

```
>>> D = 1.
>>> eq = TransientTerm() == DiffusionTerm(D)
```

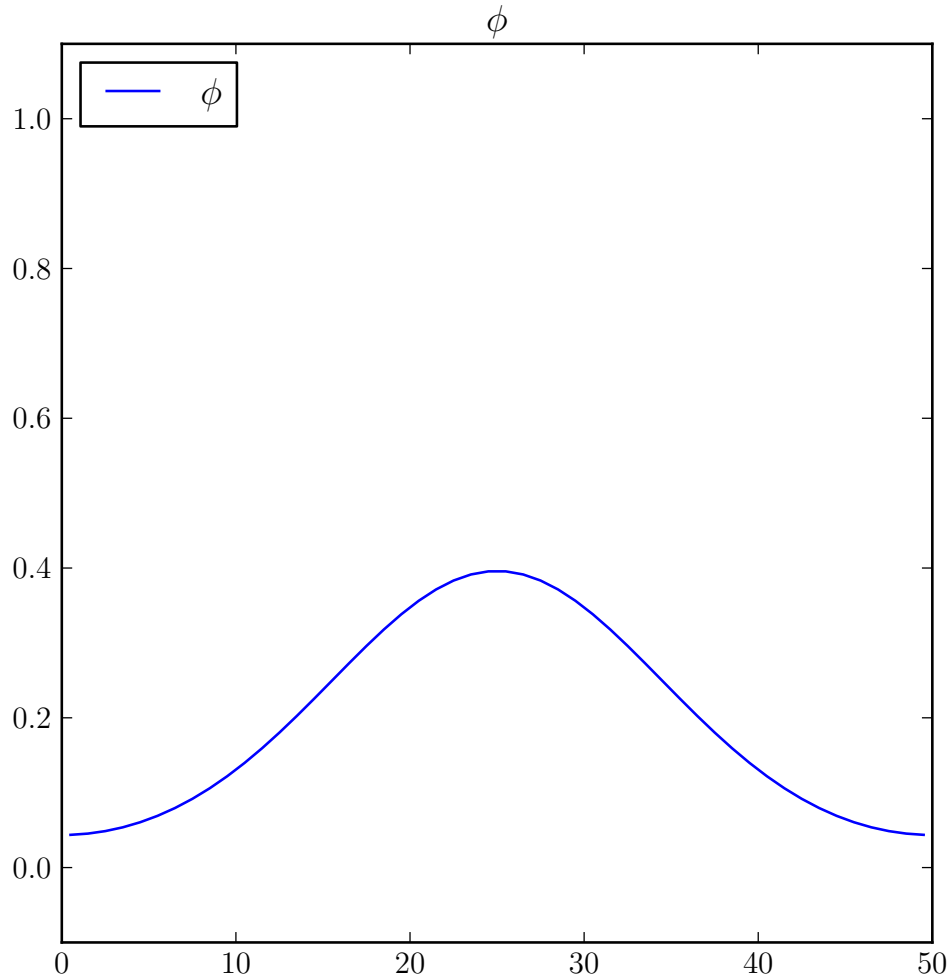
```
>>> dt = 10. * dx**2 / (2 * D)
>>> steps = 200
```

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

(continues on next page)

(continued from previous page)

```
>>> if __name__ == '__main__':
...     input("No-flux - transient. \
...     Press <return> to proceed...")
```



and see that  $\phi$  dissipates to the expected average value of 0.2 with reasonable accuracy.

```
>>> print( numerix.allclose(phi, 0.2, atol=1e-5)
True
```

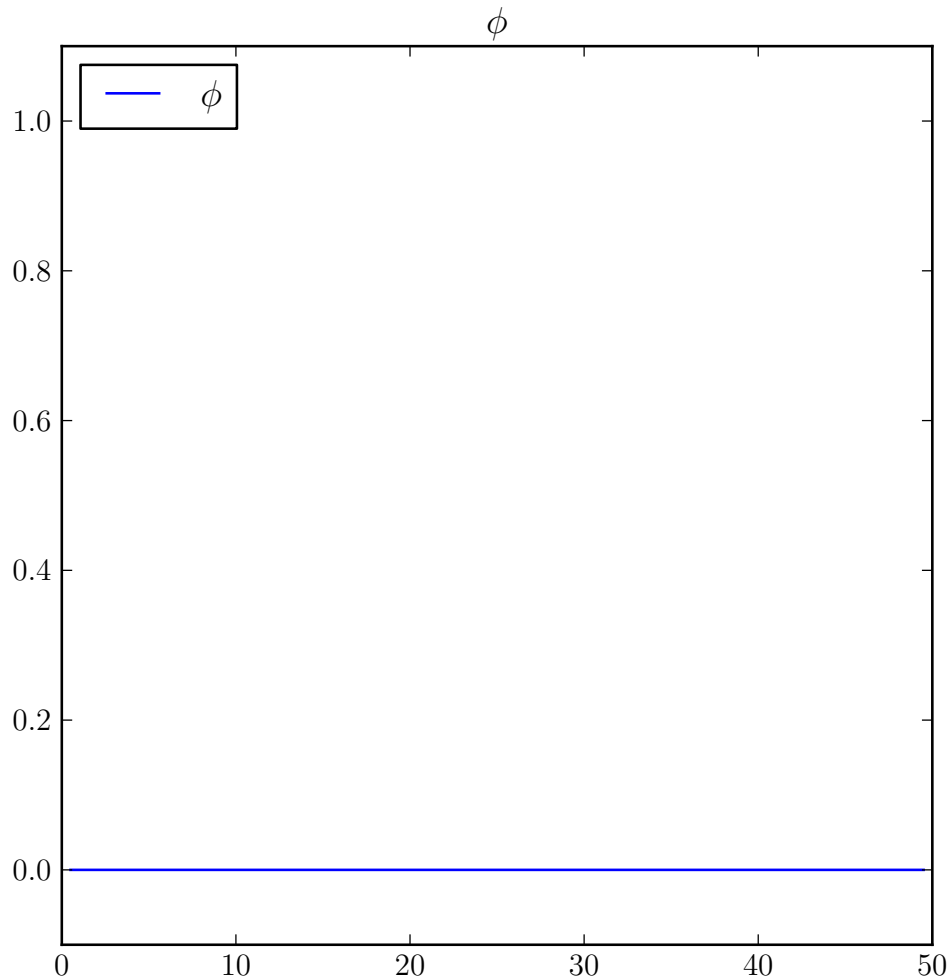
If we reset the initial condition

```
>>> phi.value = 0.
>>> phi.setValue(1., where=(x > L/2. - L/10.) & (x < L/2. + L/10.))
>>> if __name__ == '__main__':
...     viewer.plot()
```

and solve the steady-state problem

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

$$\begin{array}{cccccccc}
 \frac{D}{\Delta x^2} & -\frac{D}{\Delta x^2} & & & & & & \\
 & \ddots & \ddots & \ddots & & & & \\
 & & -\frac{D}{\Delta x^2} & \frac{2D}{\Delta x^2} & -\frac{D}{\Delta x^2} & & & \\
 & & & -\frac{D}{\Delta x^2} & \frac{2D}{\Delta x^2} & -\frac{D}{\Delta x^2} & & \\
 & & & & -\frac{D}{\Delta x^2} & \frac{2D}{\Delta x^2} & -\frac{D}{\Delta x^2} & \\
 & & & & & \ddots & \ddots & \ddots \\
 & & & & & & -\frac{D}{\Delta x^2} & \frac{D}{\Delta x^2}
 \end{array}
 \begin{array}{c}
 \phi_0^{\text{new}} \\
 \vdots \\
 \phi_{j-1}^{\text{new}} \\
 \phi_j^{\text{new}} \\
 \phi_{j+1}^{\text{new}} \\
 \vdots \\
 \phi_{N-1}^{\text{new}}
 \end{array}
 =
 \begin{array}{c}
 0 \\
 \vdots \\
 0 \\
 0 \\
 0 \\
 \vdots \\
 0
 \end{array}$$

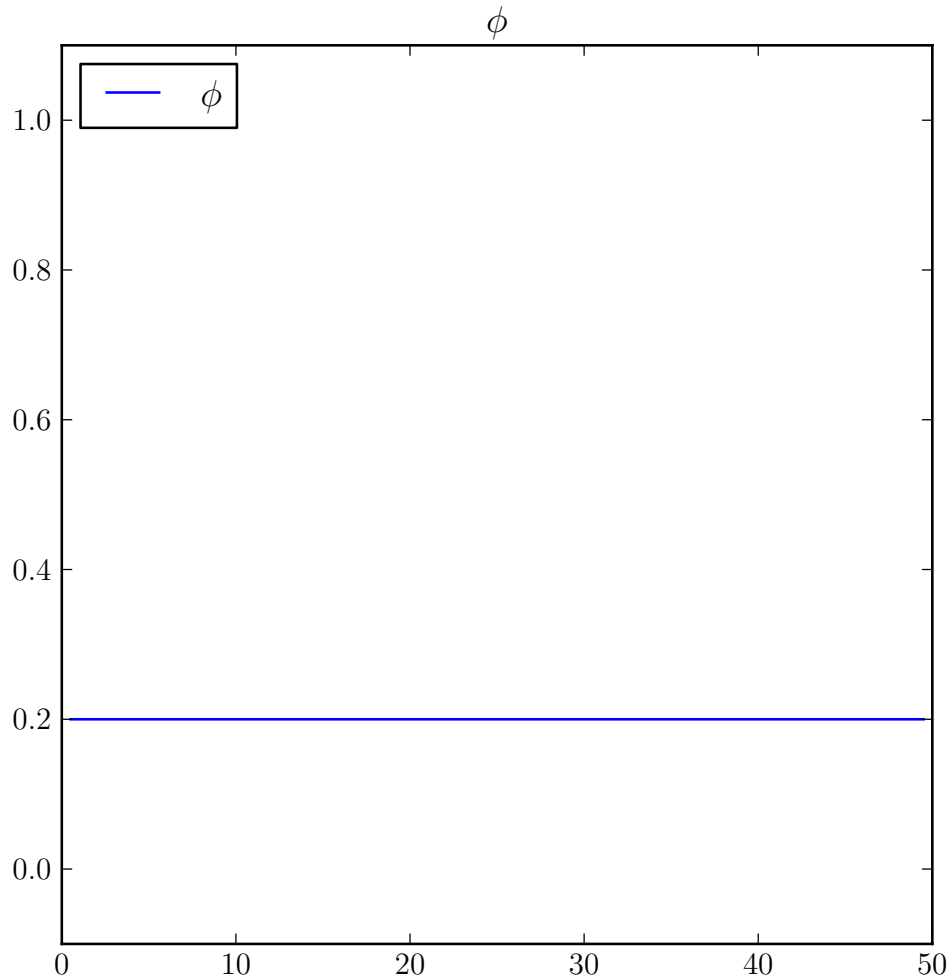
the initial condition  $\phi^{\text{old}}$  no longer appears and  $\phi = 0$  is a perfectly legitimate solution to this matrix equation.

The solution is to run the transient problem and to take one enormous time step

```
>>> phi.value = 0.
>>> phi.setValue(1., where=(x > L/2. - L/10.) & (x < L/2. + L/10.))
>>> if __name__ == '__main__':
...     viewer.plot()
```

```
>>> (TransientTerm() == DiffusionTerm(D)).solve(var=phi, dt=1e6*dt)
>>> if __name__ == '__main__':
...     viewer.plot()
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("No-flux - steady-state. \
... Press <return> to proceed...")
```

```
>>> print( numerix.allclose(phi, 0.2, atol=1e-5))
True
```



---

If this example had been written primarily as a script, instead of as documentation, we would delete every line that does not begin with either “>>>” or “. . .”, and then delete those prefixes from the remaining lines, leaving:

```
## This script was derived from
## 'examples/diffusion/mesh1D.py'

nx = 50
dx = 1.
mesh = Grid1D(nx = nx, dx = dx)
phi = CellVariable(name="solution variable",
                  mesh=mesh,
                  value=0)
```

```
eq = DiffusionTerm(coeff=D0 * (1 - phi[0]))
phi[0].setValue(valueRight)
res = 1e+10
```

(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:

```
$ python setup.py copy_script --From .../example.py --To myExample.py
```

at the command line.

Most of the *FiPy* examples will be a mixture of plain scripts and doctest documentation/tests.

### 23.5.8 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)
```

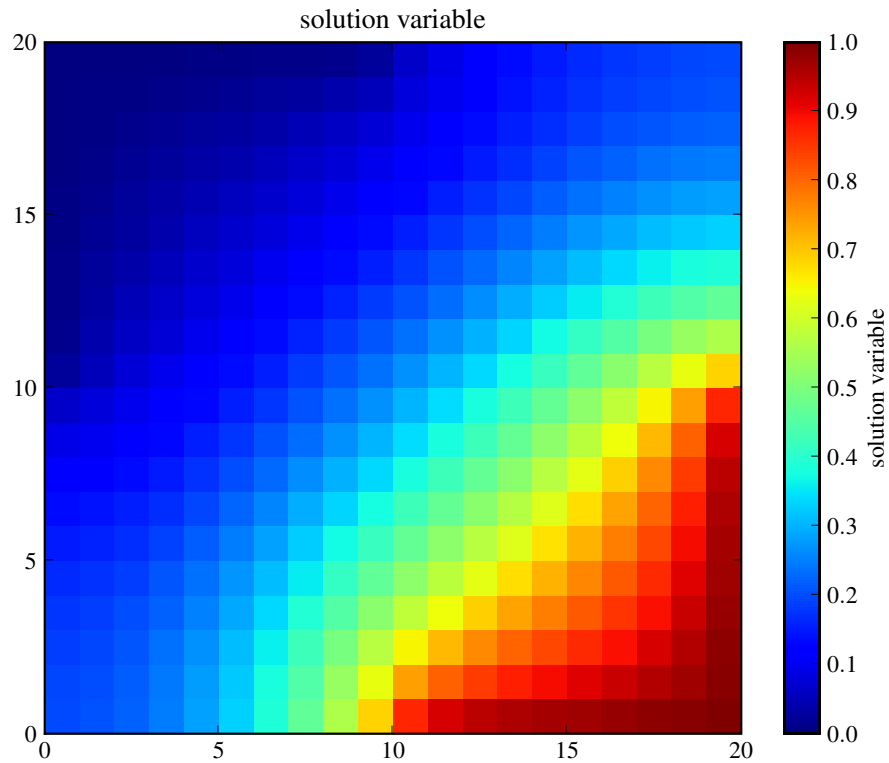
We create a viewer to see the results

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





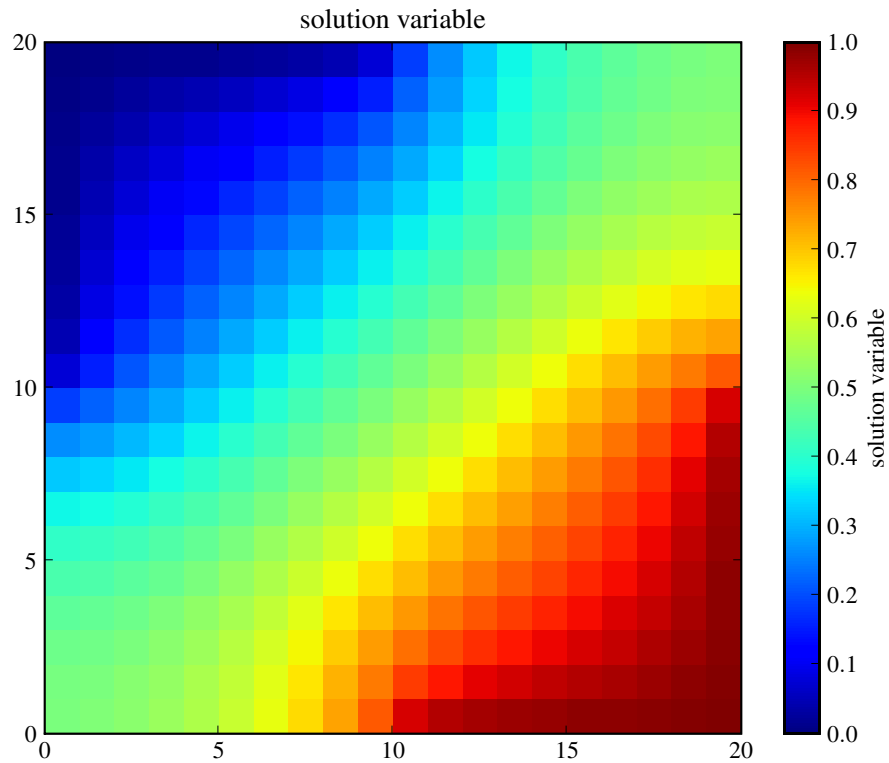
We can test the value of the bottom-right corner cell.

```
>>> print( numerix.allclose(phi((L,), (0,)), valueBottomRight, atol = 1e-2))
1
```

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

We can also solve the steady-state problem directly

```
>>> DiffusionTerm().solve(var=phi)
>>> if __name__ == '__main__':
...     viewer.plot()
```



and test the value of the bottom-right corner cell.

```
>>> print(numerix.allclose(phi((L,), (0,)), valueBottomRight, atol = 1e-2))
1
```

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

### 23.5.9 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)))
```

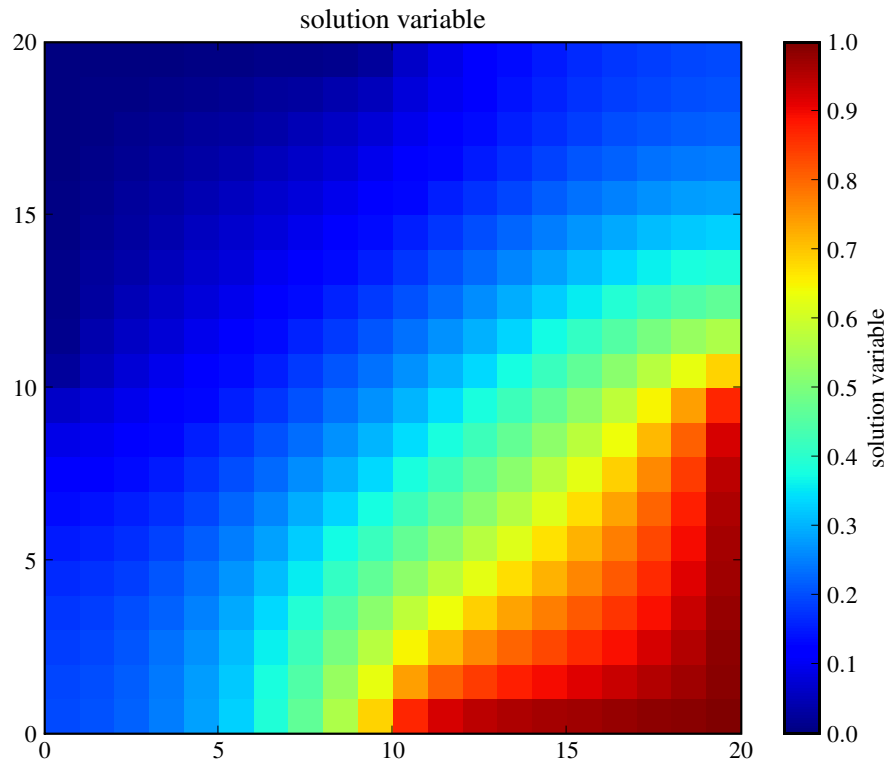
```
>>> phi.constrain(valueTopLeft, facesTopLeft)
>>> phi.constrain(valueBottomRight, facesBottomRight)
```

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



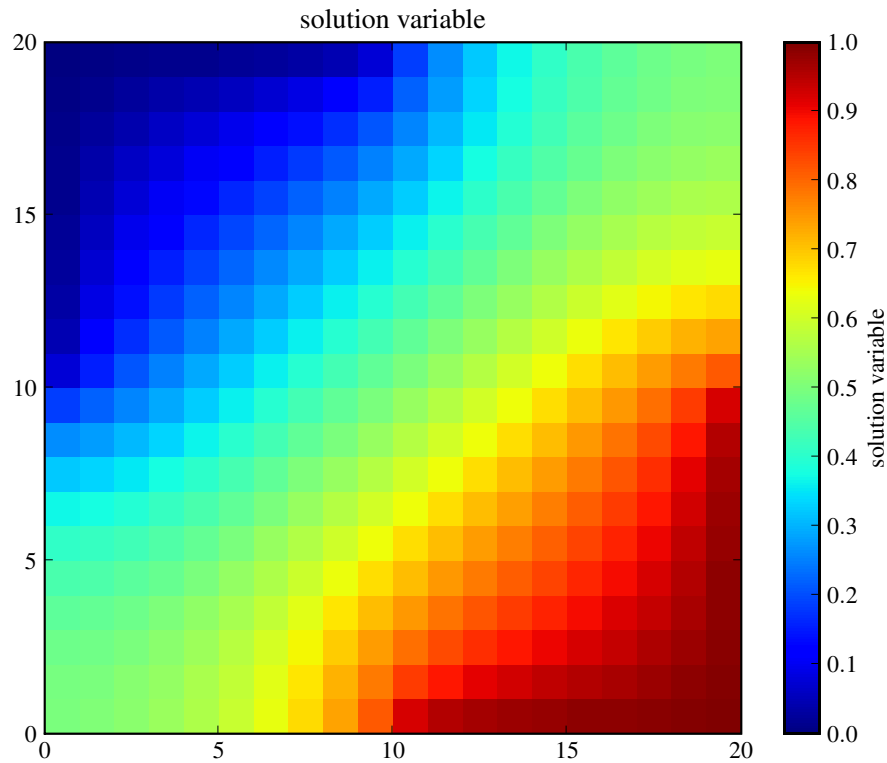
We can test the value of the bottom-right corner cell.

```
>>> 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...")
```

### 23.5.10 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, GeneralSolver
```

```
>>> nx = 500
>>> dx = L / nx
>>> mesh = Grid1D(dx=dx, nx=nx)
```

and initialize the solution variable to 0

```
>>> var = CellVariable(mesh=mesh, name='solution variable')
```

For this problem, we impose the boundary conditions:

$$\begin{aligned} \phi &= \alpha_1 & \text{at } x = 0 \\ \frac{\partial \phi}{\partial x} &= \alpha_2 & \text{at } x = L \\ \frac{\partial^2 \phi}{\partial x^2} &= \alpha_3 & \text{at } x = 0 \\ \frac{\partial^3 \phi}{\partial x^3} &= \alpha_4 & \text{at } x = L. \end{aligned}$$

or

```
>>> alpha1 = 2.
>>> alpha2 = 1.
>>> alpha3 = 4.
>>> alpha4 = -3.
```

```
>>> BCs = (NthOrderBoundaryCondition(faces=mesh.facesLeft, value=alpha3, order=2),
...       NthOrderBoundaryCondition(faces=mesh.facesRight, value=alpha4, order=3))
>>> var.faceGrad.constrain([alpha2], mesh.facesRight)
>>> var.constrain(alpha1, mesh.facesLeft)
```

We initialize the steady-state equation

```
>>> eq = DiffusionTerm(coeff=(1, 1)) == 0
```

```
>>> import fipy.solvers.solver
>>> if fipy.solvers.solver_suite == 'petsc':
...     solver = GeneralSolver(precon='lu')
... else:
...     solver = GeneralSolver()
```

We perform one implicit timestep to achieve steady state

```
>>> eq.solve(var=var,
...         boundaryConditions=BCs,
...         solver=solver)
```

The analytical solution is:

$$\phi = \frac{\alpha_4}{6}x^3 + \frac{\alpha_3}{2}x^2 + \left(\alpha_2 - \frac{\alpha_4}{2}L^2 - \alpha_3L\right)x + \alpha_1$$

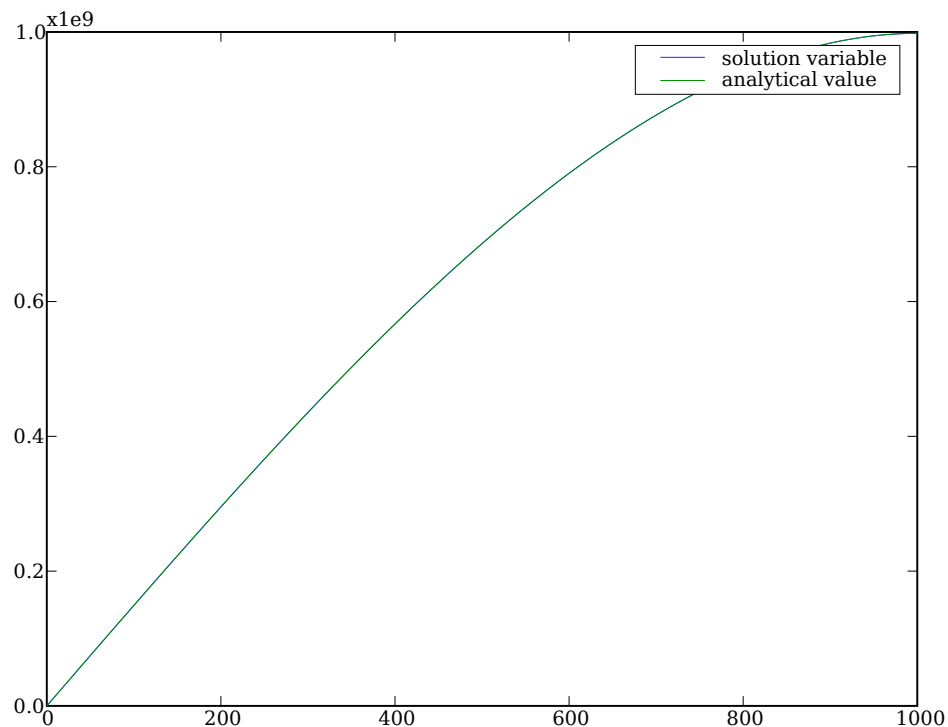
or

```
>>> analytical = CellVariable(mesh=mesh, name='analytical value')
>>> x = mesh.cellCenters[0]
>>> analytical.setValue(alpha4 / 6. * x**3 + alpha3 / 2. * x**2 + \
...                    (alpha2 - alpha4 / 2. * L**2 - alpha3 * L) * x + alpha1)
```

```
>>> print(var.allclose(analytical, rtol=1e-4))
1
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=(var, analytical))
...     viewer.plot()
```



**examples.diffusion.nthOrder.input4thOrder\_line**

```
>>> eq.solve(var,  
...         boundaryConditions = BCs,  
...         solver = solver)
```

Using the Pysparse 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****23.5.11 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**

---

*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.

*examples.diffusion.steadyState.mesh1D.  
tri2Dinput*

---

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()
>>> eq.solve(var=var, dt=1.)
```

```
>>> 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))
1
```

### 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.

```
>>> DiffusionTerm().solve(var)
```

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))
1
```

## examples.diffusion.steadyState.mesh20x20

### Modules

<code>examples.diffusion.steadyState.mesh20x20.gmshinput</code>	This input file again solves a 1D diffusion problem as in <code>./examples/diffusion/steadyState/mesh1D/input.py</code> .
<code>examples.diffusion.steadyState.mesh20x20.isotropy</code>	This input file solves a steady-state 1D diffusion problem as in <code>./examples/diffusion/mesh1D.py</code> .
<code>examples.diffusion.steadyState.mesh20x20.modifiedMeshInput</code>	This input file again solves a 1D diffusion problem as in <code>./examples/diffusion/steadyState/mesh1D/input.py</code> .
<code>examples.diffusion.steadyState.mesh20x20.orthoerror</code>	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.
<code>examples.diffusion.steadyState.mesh20x20.tri2Dinput</code>	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()
...
...     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.

```
>>> DiffusionTerm().solve(var)
```

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))
1
```

### examples.diffusion.steadyState.mesh50x50

#### Modules

<code>examples.diffusion.steadyState.mesh50x50.input</code>	This input file again solves a 1D diffusion problem as in <code>examples.diffusion.mesh1D</code> .
<code>examples.diffusion.steadyState.mesh50x50.tri2Dinput</code>	This input file again solves a 1D diffusion problem as in <code>./examples/diffusion/steadyState/mesh1D/input.py</code> .

### 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:

```
>>> DiffusionTerm().solve(var)
>>> Lx = nx * dx
>>> x = mesh.cellCenters[0]
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> print(var.allclose(analyticalArray, rtol = 1e-9))
1
```

### 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:

```
>>> DiffusionTerm().solve(var)
>>> Lx = nx * dx
>>> x = mesh.cellCenters[0]
>>> analyticalArray = valueLeft + (valueRight - valueLeft) * x / Lx
>>> print(var.allclose(analyticalArray, atol = 1e-7))
1
```

## 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*.

```
>>> DiffusionTerm().solve(var)
>>> DiffusionTerm().solve(var2)
>>> a = numerix.array(var.globalValue)
>>> b = numerix.array(var2.globalValue)
>>> c = numerix.ravel(numerix.array((b, b, b)))
>>> print(numerix.allclose(a, c))
1
```

## 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

## 23.5.12 examples.diffusion.test

Run all the test cases in examples/diffusion/

## 23.5.13 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, atol = 1e-8, rtol = 1e-8))
1
```

## 23.6 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  $\xi$

$$\underbrace{\frac{1}{M_\xi}}_{\text{transient}} \underbrace{\frac{\partial \xi}{\partial t}}_{\text{diffusion}} = \underbrace{\kappa_\xi \nabla^2 \xi}_{\text{diffusion}} - \underbrace{\sum_{j=1}^n C_j [p'(\xi) \Delta \mu_j^\circ + g'(\xi) W_j]}_{\text{source}} + \underbrace{\frac{\epsilon'(\xi)}{2} (\nabla \phi)^2}_{\text{dielectric}} \quad (23.3)$$

a set of diffusion equations for the concentrations  $C_j$ , for the substitutional elements  $j = 2, \dots, n-1$

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = \underbrace{D_j \nabla^2 C_j}_{\text{diffusion}} + \underbrace{D_j \nabla \cdot \frac{C_j}{1 - \sum_{\substack{k=2 \\ k \neq j}}^{n-1}} C_k}}_{\text{convection}} \left\{ \underbrace{\sum_{\substack{i=2 \\ i \neq j}}^{n-1} \nabla C_i}_{\text{counter diffusion}} + \underbrace{C_n [p'(\xi) \Delta \mu_{jn}^\circ + g'(\xi) W_{jn}]}_{\text{phase transformation}} \nabla \xi + \underbrace{C_n z_{jn} \nabla \phi}_{\text{electromigration}} \right\} \quad (23.4)$$

a diffusion equation for the concentration  $C_{e^-}$  of electrons

$$\frac{\partial C_{e^-}}{\partial t} = D_{e^-} \nabla^2 C_{e^-} + \underbrace{D_{e^-} \nabla \cdot C_{e^-}}_{\text{convection}} \left\{ \underbrace{[p'(\xi) \Delta \mu_{e^-}^\circ + g'(\xi) W_{e^-}]}_{\text{phase transformation}} \nabla \xi + \underbrace{z_{e^-} \nabla \phi}_{\text{electromigration}} \right\} \quad (23.5)$$

and Poisson's equation for the electrostatic potential  $\phi$

$$\underbrace{\nabla \cdot (\epsilon \nabla \phi)}_{\text{diffusion}} + \underbrace{\sum_{j=1}^n z_j C_j}_{\text{source}} = 0 \quad (23.6)$$

$M_\xi$  is the phase field mobility,  $\kappa_\xi$  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>	

### 23.6.1 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  $j^{\text{th}}$  species,  $t$  is time,  $D_{jj}$  is the self-diffusion coefficient of the  $j^{\text{th}}$  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
...
...     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].set_value(0.3)
>>> substitutionals[0].set_value(0.6, where=x > L / 2)
>>> substitutionals[1].set_value(0.6)
>>> substitutionals[1].set_value(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
...
...     convectionCoeff = CkSum.faceGrad \
...         * (Cj.diffusivity / (1. - CkFaceSum))
...
...     Cj.equation = (TransientTerm()
...         == DiffusionTerm(coeff=Cj.diffusivity)
...         + PowerLawConvectionTerm(coeff=convectionCoeff))
...     Cj.solver = DefaultAsymmetricSolver(precon=None, iterations=3200)
```

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

```
>>> from builtins import range
>>> for i in range(40):
...     for Cj in substitutionals:
...         Cj.equation.solve(var=Cj,
...             dt=10000.,
...             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\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",
```

(continues on next page)

(continued from previous page)

```

...         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()
```

```
>>> from builtins import range
>>> 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)

```

(continues on next page)

(continued from previous page)

```
...     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) \
```

(continues on next page)

(continued from previous page)

```

...     * 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

```

>>> from builtins import range
>>> 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



```
>>> from builtins import range
>>> 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
```

## 23.6.2 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("1.800000006366754e-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
...         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 = 'SO4',
...                       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',
```

(continues on next page)

(continued from previous page)

```

...     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
...     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
```

(continues on next page)

(continued from previous page)

```

...     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)
...     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",

```

(continues on next page)

(continued from previous page)

```

↪"residual"))
>>> residual = 0.
>>> from builtins import range
>>> from builtins import str
>>> 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
...
...             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:

```

(continues on next page)

(continued from previous page)

```

...     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))
>>> 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

```

### 23.6.3 examples.elphf.phase

A simple 1D example to test the setup of the phase field equation.

We rearrange Eq. (23.3) to

$$\frac{1}{M_\xi} \frac{\partial \xi}{\partial t} = \kappa_\xi \nabla^2 \xi + \frac{e'(\xi)}{2} (\nabla \phi)^2 - [p'(\xi) \Delta \mu_n^\circ + g'(\xi) W_n] - \sum_{j=2}^{n-1} C_j [p'(\xi) \Delta \mu_{jn}^\circ + g'(\xi) W_{jn}] - C_{e^-} [p'(\xi) \Delta \mu_{e^-}^\circ + g'(\xi) W_{e^-}]$$

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  $\xi$  is the phase field,  $t$  is time,  $M_\xi$  is the phase field mobility,  $\kappa_\xi$  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

```
>>> from builtins import range
>>> for step in range(10):
...     phase.equation.solve(var = phase, dt=1.)
```

Since we have only a single component  $n$ , with  $\Delta\mu_n^o = 0$ , and the electrostatic potential is uniform, Eq. (23.3) 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

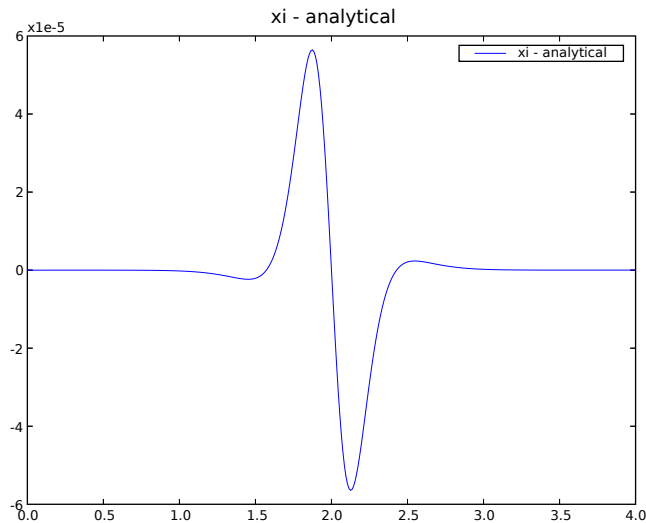
```
>>> x = mesh.cellCenters[0]
```

```
>>> 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()
```



### 23.6.4 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*

```

>>> def makeEquations(phase, substitutionals, interstitials):
...     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
...
...     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()

```

(continues on next page)

(continued from previous page)

```
... == DiffusionTerm(coeff=Cj.diffusivity)
... + PowerLawConvectionTerm(coeff=convectionCoeff))
```

```
>>> 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

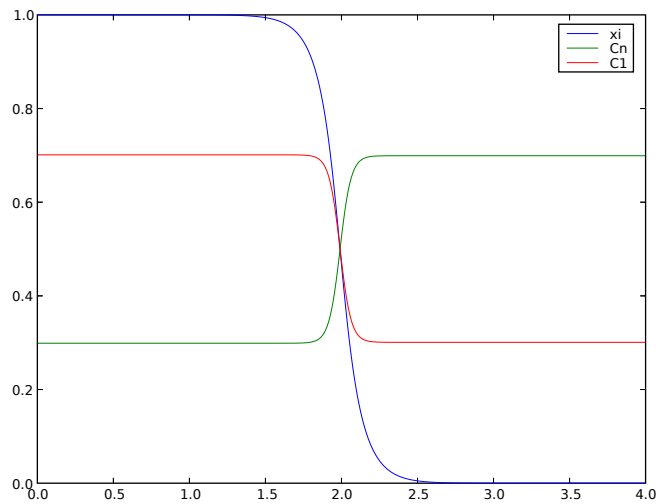
```
>>> substitutionals[0].setValue(0.5)
```

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
>>> from builtins import range
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
...     phase.equation.solve(var = phase, dt = dt)
...     for field in substitutionals + interstitials:
...         field.equation.solve(var = field,
...                               dt = dt)
...     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  $C_3 = 0.15$ .

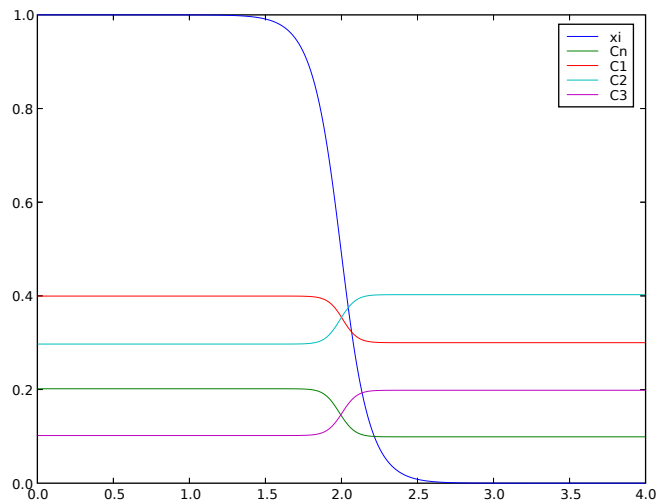
```
>>> substitutionals[0].setValue(0.35)
>>> substitutionals[1].setValue(0.35)
>>> substitutionals[2].setValue(0.15)
```

We make new equations

```
>>> makeEquations(phase, substitutionals, interstitials)
```

and again iterate to equilibrium

```
>>> dt = 10000
>>> from builtins import range
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
...     phase.equation.solve(var = phase, dt = dt)
...     for field in substitutionals + interstitials:
...         field.equation.solve(var = field,
...                               dt = dt)
...     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)
↪3)
1
>>> numerix.allclose(substitutionals[1](((0., L),)), (0.3, 0.4), rtol = 3e-3, atol = 3e-3)
↪3)
1
>>> numerix.allclose(substitutionals[2](((0., L),)), (0.1, 0.2), rtol = 3e-3, atol = 3e-3)
↪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  $C_3 = 0.15$ .

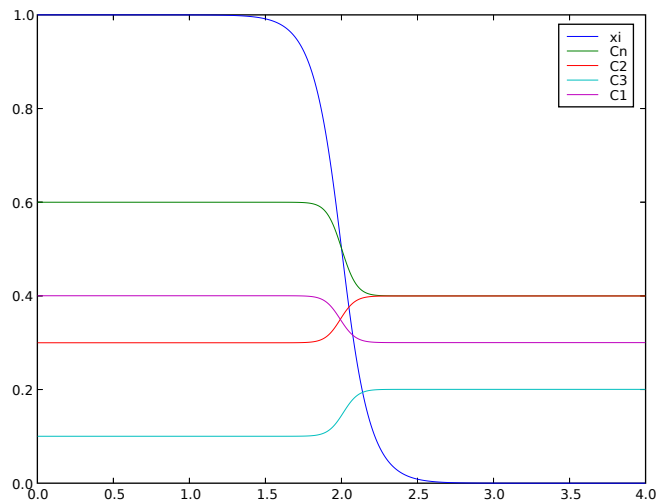
```
>>> substitutionals[0].setValue(0.35)
>>> substitutionals[1].setValue(0.15)
```

We make new equations

```
>>> makeEquations(phase, substitutionals, interstitials)
```

and again iterate to equilibrium

```
>>> dt = 10000
>>> from builtins import range
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
...     phase.equation.solve(var = phase, dt = dt)
...     for field in substitutionals + interstitials:
...         field.equation.solve(var = field,
...                               dt = dt)
...     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
```

### 23.6.5 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  $\phi$  is the electrostatic potential,  $\epsilon$  is the permittivity,  $\rho$  is the charge density,  $C_j$  is the concentration of the  $j^{\text{th}}$  component, and  $z_j$  is the valence of the  $j^{\text{th}}$  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()
```

## 23.6.6 examples.elphf.test

## 23.7 examples.flow

### Modules

*examples.flow.stokesCavity*  
*examples.flow.test*

Solve the Navier-Stokes equation in the viscous limit.

### 23.7.1 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  $\mu$  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()`.

```
>>> from builtins import range
>>> for sweep in range(sweeps):
...     ...
```

(continues on next page)

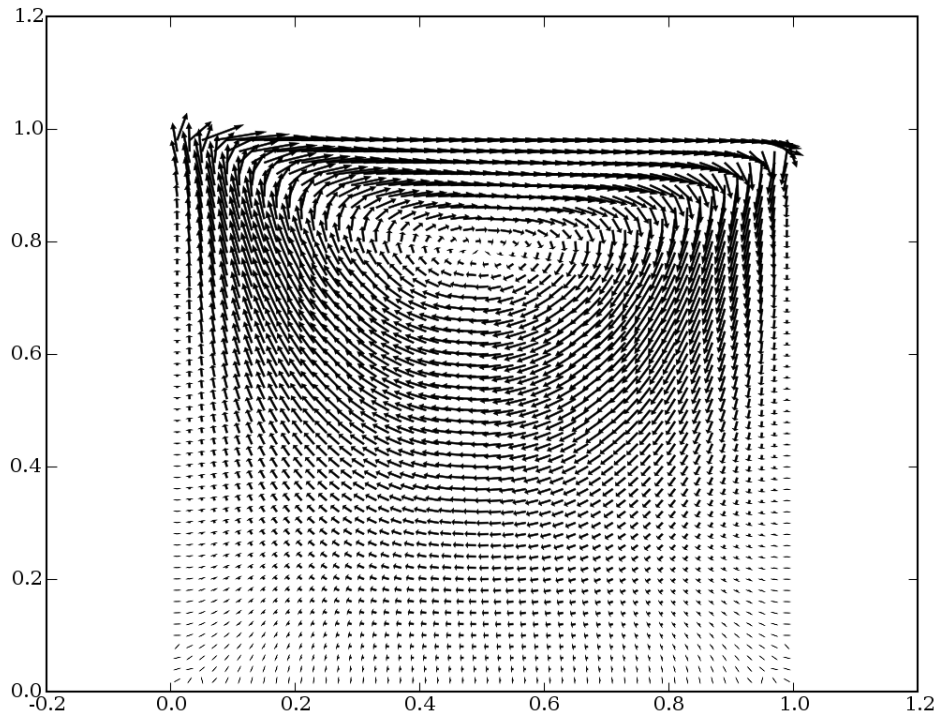


(continued from previous page)

```

...     ## 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)
...     rhs = pressureCorrectionEq.RHSvector
...
...     ## update the pressure using the corrected value
...     pressure.setValue(pressure + pressureRelaxation * pressureCorrection )
...     ## update the velocity using the corrected pressure
...     xVelocity.setValue(xVelocity - pressureCorrection.grad[0] / \
...                         ap * mesh.cellVolumes)
...     yVelocity.setValue(yVelocity - pressureCorrection.grad[1] / \
...                         ap * mesh.cellVolumes)
...
...     if __name__ == '__main__':
...         if sweep%10 == 0:
...             print('sweep:', sweep, ', x residual:', xres, \
...                   ', y residual', yres, \
...                   ', p residual:', pres, \
...                   ', continuity:', max(abs(rhs)))
...
...         viewer.plot()

```



Test values in the last cell.

```
>>> print(numerix.allclose(pressure.globalValue[... , -1], 162.790867927))
1
>>> print(numerix.allclose(xVelocity.globalValue[... , -1], 0.265072740929))
1
>>> print(numerix.allclose(yVelocity.globalValue[... , -1], -0.150290488304))
1
```

### 23.7.2 examples.flow.test

## 23.8 examples.levelSet

### Modules

*examples.levelSet.advection*

*examples.levelSet.distanceFunction*

*examples.levelSet.electroChem*

*examples.levelSet.surfactant*

*examples.levelSet.test*

Run all the test cases in examples/

## 23.8.1 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.

```
>>> from builtins import range
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-radius, datamax=radius)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following commands.

```
>>> from builtins import range
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = numerix.array(mesh.cellCenters[0])
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = initialArray - distanceTravelled
>>> answer = numerix.where(answer < 0., -1001., answer)
>>> solution = numerix.where(answer < 0., -1001., numerix.array(var))
>>> numerix.allclose(answer, solution, atol=4.7e-3)
1
```

If the advection equation is built with the *AdvectionTerm()* the result is more accurate,

```
>>> var.setValue(initialArray)
>>> advEqn = TransientTerm() + AdvectionTerm(velocity)
>>> from builtins import range
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> solution = numerix.where(answer < 0., -1001., numerix.array(var))
>>> numerix.allclose(answer, solution, atol=1.02e-3)
1
```

**examples.levelSet.advection.mesh1D**

Solve the distance function equation in one dimension and then advect it.

This example first solves the distance function equation in one dimension:

$$|\nabla\phi| = 1$$

with  $\phi = 0$  at  $x = L/5$ .

The variable is then advected with,

$$\frac{\partial\phi}{\partial t} + \vec{u} \cdot \nabla\phi = 0$$

The scheme used in the *FirstOrderAdvectionTerm* preserves the *var* as a distance function.

The solution to this problem will be demonstrated in the following script. Firstly, setup the parameters.

```
>>> from fipy import CellVariable, Grid1D, DistanceVariable, TransientTerm, \
↳ FirstOrderAdvectionTerm, AdvectionTerm, Viewer
>>> from fipy.tools import numerix, serialComm
```

```
>>> velocity = 1.
>>> dx = 1.
>>> nx = 10
>>> timeStepDuration = 1.
>>> steps = 2
>>> L = nx * dx
>>> interfacePosition = L / 5.
```

Construct the mesh.

```
>>> mesh = Grid1D(dx=dx, nx=nx, communicator=serialComm)
```

Construct a *distanceVariable* object.

```
>>> var = DistanceVariable(name='level set variable',
...                        mesh=mesh,
...                        value=-1.,
...                        hasOld=1)
>>> var.setValue(1., where=mesh.cellCenters[0] > interfacePosition)
>>> var.calcDistanceFunction()
```

The *advectionEquation* is constructed.

```
>>> advEqn = TransientTerm() + FirstOrderAdvectionTerm(velocity)
```

The problem can then be solved by executing a series of time steps.

```
>>> from builtins import range
>>> if __name__ == '__main__':
...     viewer = Viewer(vars=var, datamin=-10., datamax=10.)
...     viewer.plot()
...     for step in range(steps):
...         var.updateOld()
...         advEqn.solve(var, dt=timeStepDuration)
...         viewer.plot()
```

The result can be tested with the following code:

```
>>> from builtins import range
>>> for step in range(steps):
...     var.updateOld()
...     advEqn.solve(var, dt=timeStepDuration)
>>> x = mesh.cellCenters[0]
>>> distanceTravelled = timeStepDuration * steps * velocity
>>> answer = x - interfacePosition - timeStepDuration * steps * velocity
>>> 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()
```

```
>>> from builtins import range
>>> 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))
>>> print(var.allclose(answer, atol = 1e-1))
1
```

## 23.8.2 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,
...     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`

### 23.8.3 `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.
>>> from builtins import range
>>> 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,

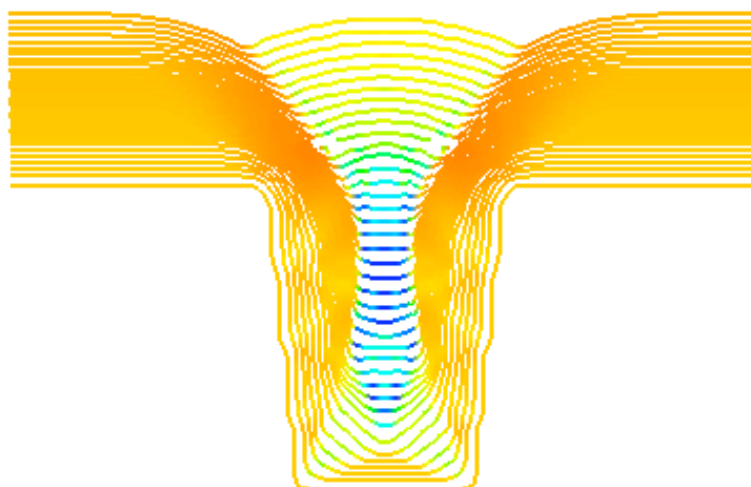
$$i = \frac{c_m}{c_m^\infty} (b_0 + b_1 \theta).$$

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:

```
$ python setup.py copy_script --From examples/levelSet/electroChem/howToWriteAScript.py -
↳-To 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,  $\phi$ , 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  $\phi$  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,  $\theta$ , variable. This variable influences the deposition rate.

```
>>> catalystVar = SurfactantVariable(
...     name="catalyst variable",
...     value=catalystCoverage,
...     distanceVar=distanceVar)
```

Create the bulk catalyst concentration,  $c_\theta$ , 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  $\Omega$  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,  $c_m^\infty$  is the far-field bulk concentration of metal ions,  $\alpha$  is the transfer coefficient,  $R$  is the gas constant,  $T$  is the temperature and  $\eta$  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  $v_{\text{ext}}$ . 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  $\theta$  is the coverage of catalyst at the interface,  $J$  is the curvature of the interface,  $v$  is the normal velocity of the interface,  $c_{\theta}^i$  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  $\phi$  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  $v_{\text{ext}}$  throughout the whole domain using  $\nabla\phi \cdot \nabla v_{\text{ext}} = 0$ .

```
>>> from builtins import range
>>> 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,

$$\begin{aligned} \dot{\theta}_L &= \kappa v \theta_L + k_L^+ c_L (1 - \theta_L) - k_L^- v \theta_L, \\ \dot{\theta}_A &= \kappa v \theta_A + k_A^+ c_A (1 - \theta_A - \theta_L) - k_L c_L \theta_A - k_A^- \theta_A^{q-1}, \\ \dot{\theta}_S &= 1 - \theta_A - \theta_L \\ \dot{\theta}_V &= 0. \end{aligned}$$

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  $\eta$  given by,

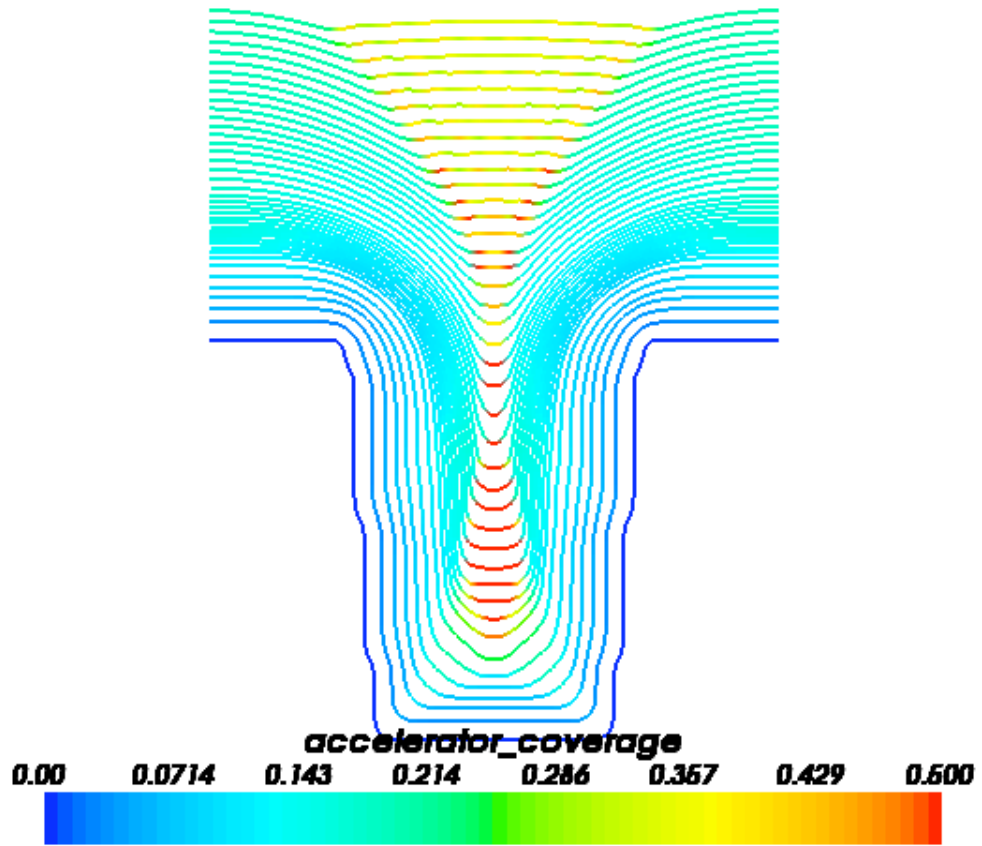
$$\begin{aligned} k_A^+ &= k_{A0}^+ \exp \frac{-\alpha_k F \eta}{RT}, \\ k_A^- &= B_d + \frac{A}{\exp(B_a(\eta + V_d))} + \exp(B_b(\eta + V_d)) \\ q &= m\eta + b. \end{aligned}$$

The following table shows the symbols used in the governing equations and their corresponding arguments for the

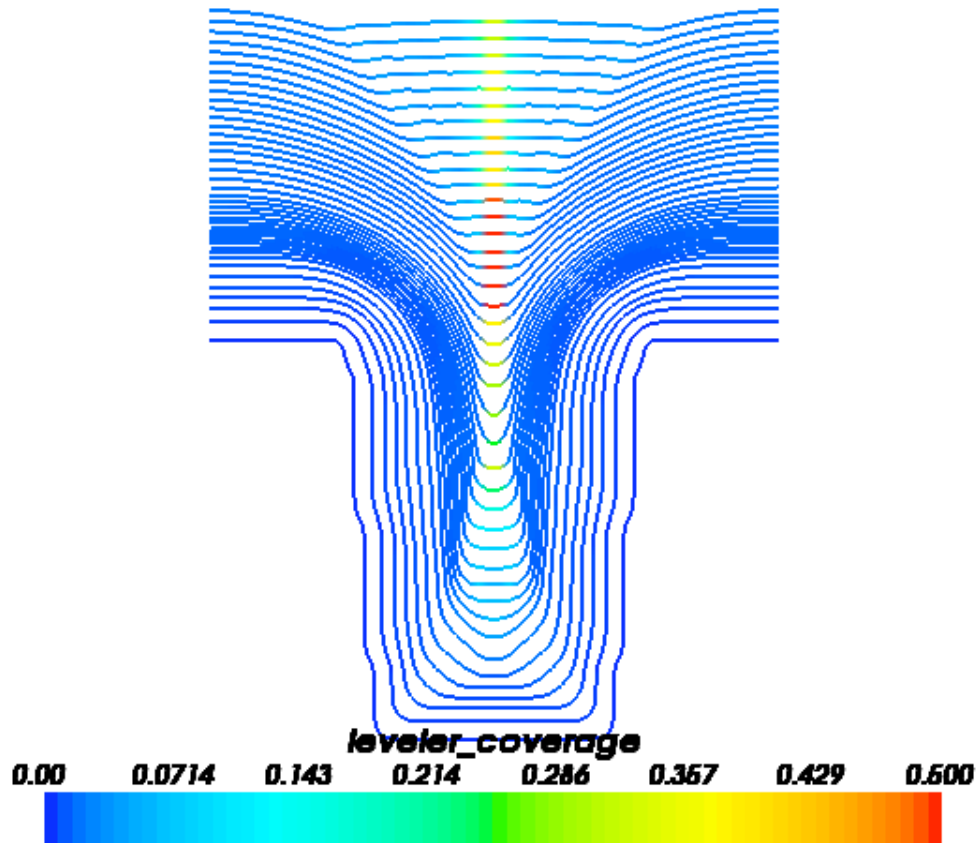
runLeveler() function.

Symbol	Description	Keyword Argument	Value	Unit
Deposition Rate Parameters				
$v$	deposition rate			$\text{m s}^{-1}$
$i_A$	accelerator current density	<code>i0Accelerator</code>		$\text{A m}^{-2}$
$i_L$	leveler current density	<code>i0Leveler</code>		$\text{A m}^{-2}$
$\Omega$	molar volume	<code>molarVolume</code>	$7.1 \times 10^{-6}$	$\text{m}^3 \text{mol}^{-1}$
$n$	ion charge	<code>charge</code>	2	
$F$	Faraday's constant	<code>faradaysConstant</code>	$9.6 \times 10^{-4}$	$\text{C mol}^{-1}$
$i_0$	exchange current density			$\text{A m}^{-2}$
$\alpha_A$	accelerator transfer coefficient	<code>alphaAccelerator</code>	0.4	
$\alpha_S$	leveler transfer coefficient	<code>alphaLeveler</code>	0.5	
$\eta$	overpotential	<code>overpotential</code>	-0.3	V
$R$	gas constant	<code>gasConstant</code>	8.314	$\text{J K mol}^{-1}$
$T$	temperature	<code>temperature</code>	298.0	K
Ion Parameters				
$c_I$	ion concentration	<code>ionConcentration</code>	250.0	$\text{mol m}^{-3}$
$c_I^\infty$	far field ion concentration	<code>ionConcentration</code>	250.0	$\text{mol m}^{-3}$
$D_I$	ion diffusion coefficient	<code>ionDiffusion</code>	$5.6 \times 10^{-10}$	$\text{m}^2 \text{s}^{-1}$
Accelerator Parameters				
$\theta_A$	accelerator coverage	<code>acceleratorCoverage</code>	0.0	
$c_A$	accelerator concentration	<code>acceleratorConcentration</code>	$5.0 \times 10^{-3}$	$\text{mol m}^{-3}$
$c_A^\infty$	far field accelerator concentration	<code>acceleratorConcentration</code>	$5.0 \times 10^{-3}$	$\text{mol m}^{-3}$
$D_A$	catalyst diffusion coefficient	<code>catalystDiffusion</code>	$1.0 \times 10^{-9}$	$\text{m}^2 \text{s}^{-1}$
$\Gamma_A$	accelerator site density	<code>siteDensity</code>	$9.8 \times 10^{-6}$	$\text{mol m}^{-2}$
$k_A^+$	accelerator adsorption			$\text{m}^3 \text{mol}^{-1} \text{s}^{-1}$
$k_{A0}^+$	accelerator adsorption coeff	<code>kAccelerator0</code>	$2.6 \times 10^{-4}$	$\text{m}^3 \text{mol}^{-1} \text{s}^{-1}$
$\alpha_k$	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>	$9.8 \times 10^{-2}$	
$B_d$	experimental parameter	<code>Bd</code>	$8.0 \times 10^{-4}$	
Geometry Parameters				
$D$	trench depth	<code>trenchDepth</code>	$0.5 \times 10^{-6}$	m
$D/W$	trench aspect ratio	<code>aspectRatio</code>	2.0	
$S$	trench spacing	<code>trenchSpacing</code>	$0.6 \times 10^{-6}$	m
$\delta$	boundary layer depth	<code>boundaryLayerDepth</code>	$0.3 \times 10^{-6}$	m
Simulation Control Parameters				
	computational cell size	<code>cellSize</code>	$0.1 \times 10^{-7}$	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)
>>> from builtins import range
>>> 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)
>>> 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 * dx
>>> Ly = nx * dy
>>> 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
```

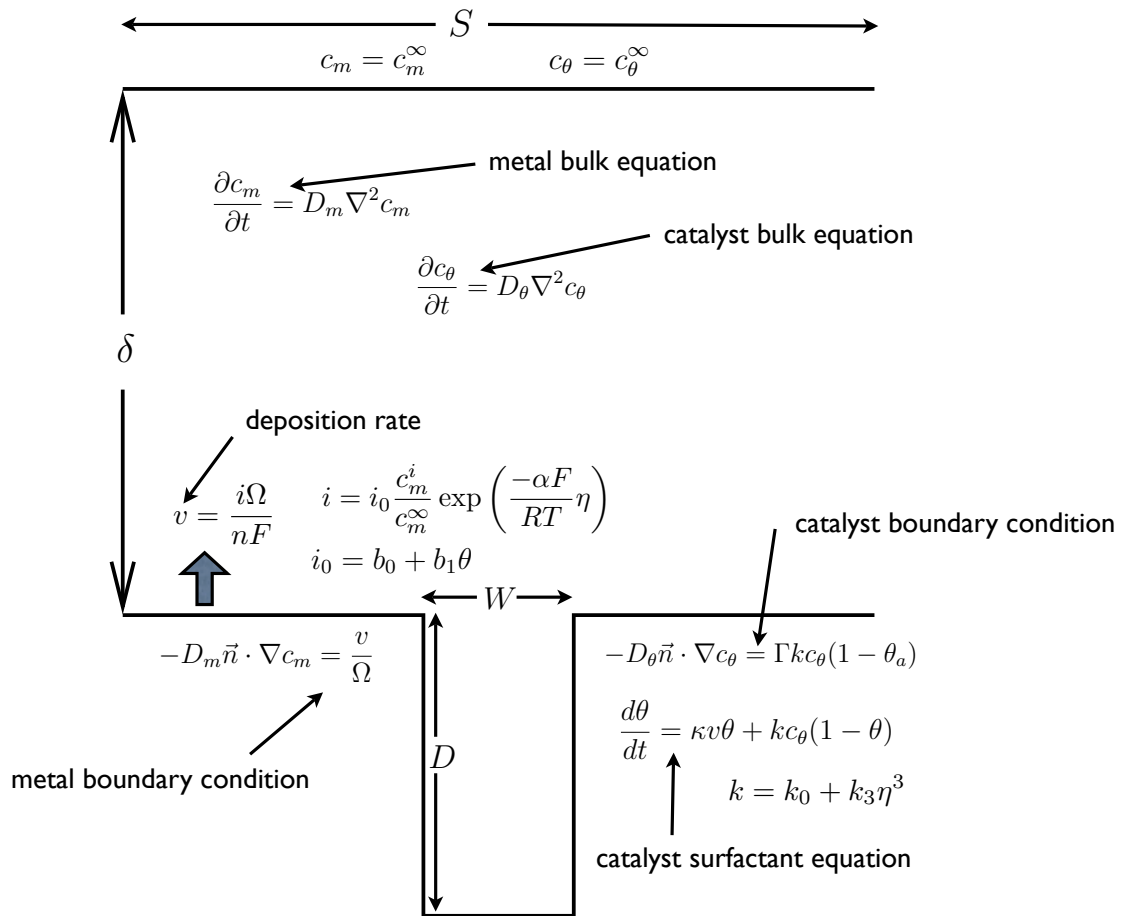
Change the `displayViewers` argument to `True` if you wish to see the results displayed on the screen. Example [examples.levelSet.electroChem.simpleTrenchSystem](#) gives explanation for writing new scripts or modifying existing scripts that are encapsulated by functions.

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.

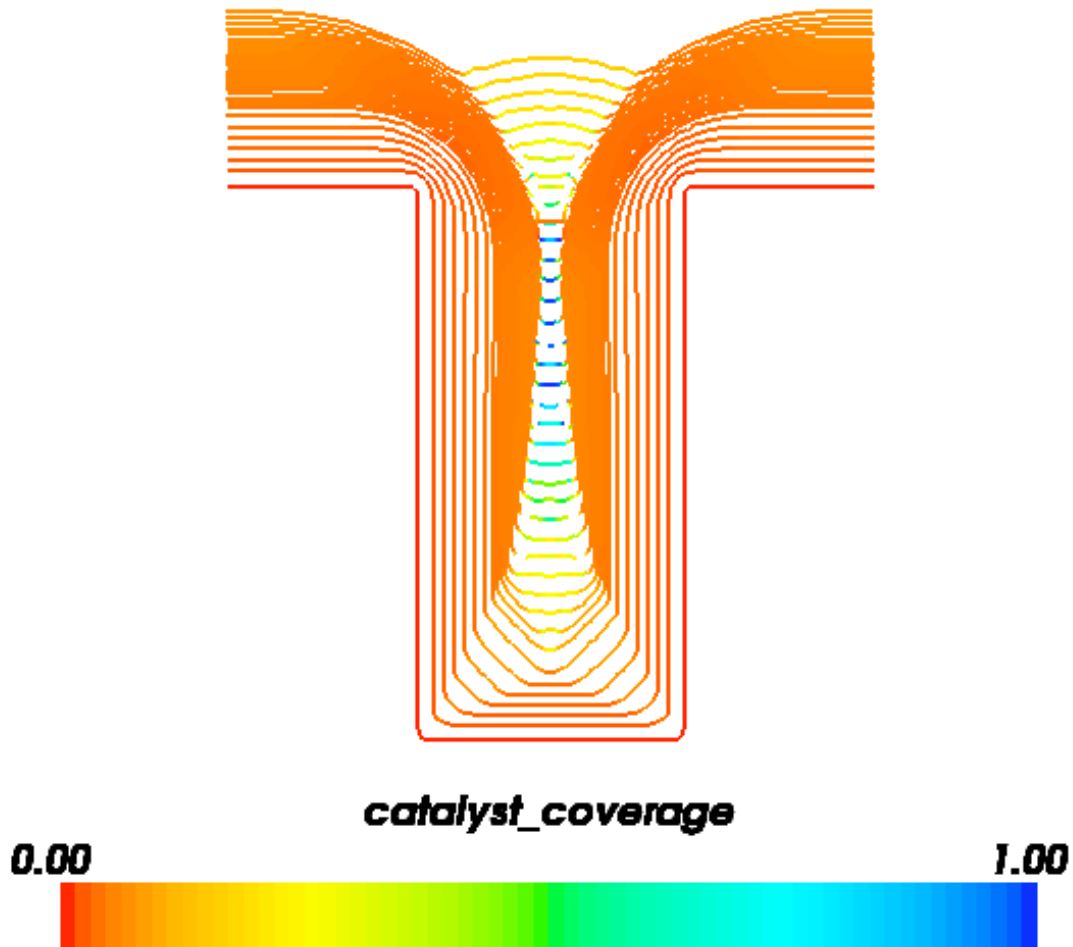




The following table shows the symbols used in the governing equations and their corresponding arguments to the `runSimpleTrenchSystem()` function. The boundary layer depth is intentionally small in this example in order not to complicate the mesh. Further examples will simulate more realistic boundary 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			$\text{m s}^{-1}$
$i$	current density			$\text{A m}^{-2}$
$\Omega$	molar volume	molarVolume	$7.1 \times 10^{-6}$	$\text{m}^3 \text{mol}^{-1}$
$n$	ion charge	charge	2	
$F$	Faraday's constant	faradaysConstant	$9.6 \times 10^{-4}$	$\text{C mol}^{-1}$
$i_0$	exchange current density			$\text{A m}^{-2}$
$\alpha$	transfer coefficient	transferCoefficient	0.5	
$\eta$	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 $\theta^0$	currentDensity0	0.26	$\text{A m}^{-2}$
$b_1$	current density for $\theta$	currentDensity1	45.0	$\text{A m}^{-2}$
Metal Ion Parameters				
$c_m$	metal ion concentration	metalConcentration	250.0	$\text{mol m}^{-3}$
$c_m^\infty$	far field metal ion concentration	metalConcentration	250.0	$\text{mol m}^{-3}$
$D_m$	metal ion diffusion coefficient	metalDiffusion	$5.6 \times 10^{-10}$	$\text{m}^2 \text{s}^{-1}$
Catalyst Parameters				
$\theta$	catalyst surfactant concentration	catalystCoverage	0.0	
$c_\theta$	bulk catalyst concentration	catalystConcentration	$5.0 \times 10^{-3}$	$\text{mol m}^{-3}$
$c_\theta^\infty$	far field catalyst concentration	catalystConcentration	$5.0 \times 10^{-3}$	$\text{mol m}^{-3}$
$D_\theta$	catalyst diffusion coefficient	catalystDiffusion	$1.0 \times 10^{-9}$	$\text{m}^2 \text{s}^{-1}$
$\Gamma$	catalyst site density	siteDensity	$9.8 \times 10^{-6}$	$\text{mol m}^{-2}$
$k$	rate constant			$\text{m}^3 \text{mol}^{-1} \text{s}^{-1}$
$k_0$	rate constant for $\eta^0$	rateConstant0	1.76	$\text{m}^3 \text{mol}^{-1} \text{s}^{-1}$
$k_3$	rate constant for $\eta^3$	rateConstant3	$-245.0 \times 10^{-6}$	$\text{m}^3 \text{mol}^{-1} \text{s}^{-1} \text{V}^{-3}$
Geometry Parameters				
$D$	trench depth	trenchDepth	$0.5 \times 10^{-6}$	m
$D/W$	trench aspect ratio	aspectRatio	2.0	
$S$	trench spacing	trenchSpacing	$0.6 \times 10^{-6}$	m
$\delta$	boundary layer depth	boundaryLayerDepth	$0.3 \times 10^{-6}$	m
Simulation Control Parameters				
	computational cell size	cellSize	$0.1 \times 10^{-7}$	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`

### 23.8.4 `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)
>>> from builtins import range
>>> 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
>>> from builtins import range
>>> 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)
>>> from builtins import range
>>> 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`

## 23.8.5 `examples.levelSet.test`

Run all the test cases in `examples/`

## 23.9 `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 <code>examples/meshing/</code>

### 23.9.1 `examples.meshing.gmshRefinement`

### 23.9.2 `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()
```

### 23.9.3 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
```

### 23.9.4 examples.meshing.test

Run all the test cases in examples/meshing/

## 23.10 examples.parallel

## 23.11 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.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>	

### 23.11.1 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  $\phi$  (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  $\Delta T$  ( $\Delta T = 0$  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  $\beta = \frac{1-\phi^2}{1+\phi^2}$ ,  $\Phi = \tan\left(\frac{N}{2}\psi\right)$ ,  $\psi = \theta + \arctan\left(\frac{\partial \phi / \partial y}{\partial \phi / \partial x}\right)$ ,  $\theta$  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,))
...

```

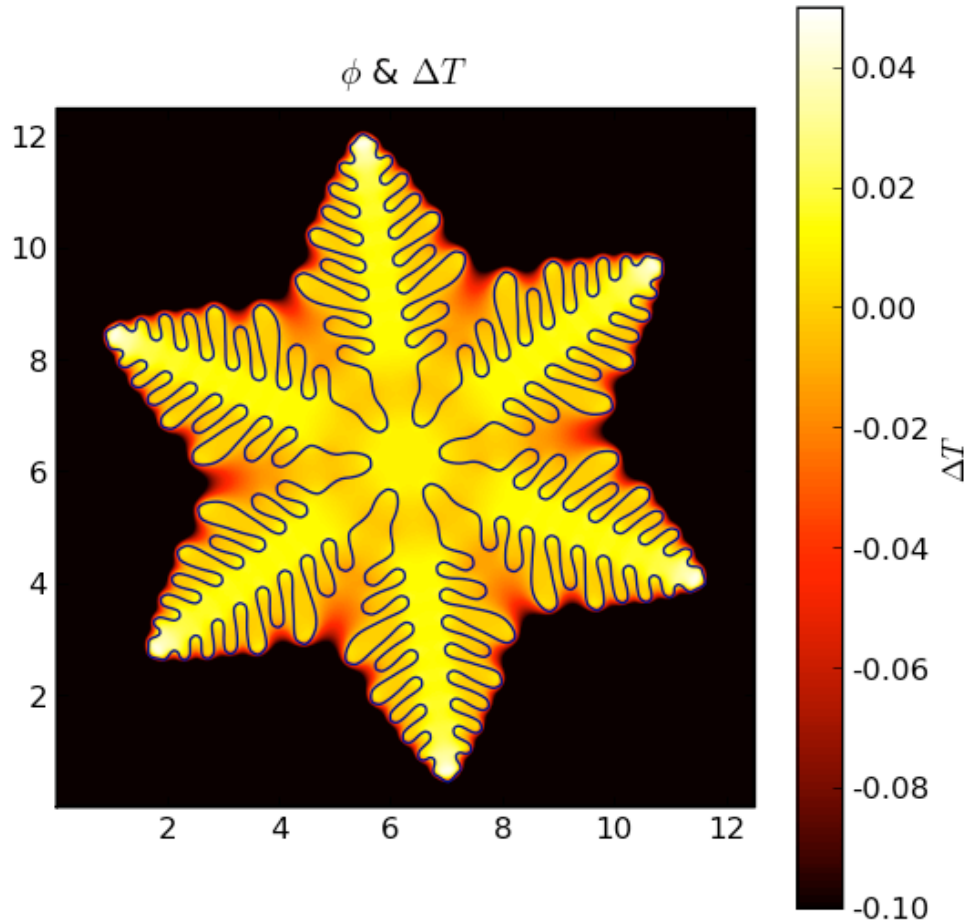
(continues on next page)

(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
>>> from builtins import range
>>> for i in range(steps):
...     phase.updateOld()
...     dT.updateOld()
...     phaseEq.solve(phase, dt=dt)
...     heatEq.solve(dT, dt=dt)
...     if __name__ == "__main__" and (i % 10 == 0):
...         viewer.plot()
```



The non-uniform temperature results from the release of latent heat at the solidifying interface. The dendrite arms grow fastest where the temperature gradient is steepest.

We note that this FiPy simulation is written in about 50 lines of code (excluding the custom viewer), compared with over 800 lines of (fairly lucid) FORTRAN code used for the figures in [13].

### 23.11.2 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,

```
>>> steps = 10
>>> from builtins import range
>>> for i in range(steps):
...     phase.updateOld()
...     temperature.updateOld()
...     phaseEq.solve(phase, dt=timeStepDuration)
...     temperatureEq.solve(temperature, dt=timeStepDuration)
...     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
>>> from future.utils import text_to_native_str
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + text_to_native_str('.gz'))
>>> print(phase.allclose(testData))
1
```

### 23.11.3 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  $\phi$ <sup>1</sup>

<sup>1</sup> 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  $\phi$  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

$$\frac{\partial \phi}{\partial t} = M_\phi \left( \kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \right) \quad (23.7)$$

for phase and

$$\frac{\partial C}{\partial t} = \nabla \cdot \left( M_C \nabla \frac{\partial f}{\partial C} \right) \quad (23.8)$$

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

```
>>> R = 8.314 # J / (mol K)
```

is the gas constant and  $\Omega_S$  and  $\Omega_L$  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

$$f(\phi, C, T) = p(\phi) f^S(C, T) + (1 - p(\phi)) f^L(C, T) + \left[ (1 - C) \frac{W_A}{2} + C \frac{W_B}{2} \right] g(\phi).$$

<sup>2</sup> we are going to want to examine different temperatures in this example, so we declare  $T$  as a *Variable*

When the thermodynamic models are the same in both phases, both approaches should yield the same result.

We choose the first approach and make the simplifying assumptions of an ideal solution and that

$$f_A^L(T) = 0$$

$$f_A^S(T) - f_A^L(T) = \frac{L_A (T - T_M^A)}{T_M^A}$$

and likewise for component  $B$ .

```
>>> LA = 2350. # J / cm**3
>>> LB = 1728. # J / cm**3
>>> TmA = 1728. # K
>>> TmB = 1358. # K
```

```
>>> enthalpyA = LA * (T - TmA) / TmA
>>> enthalpyB = LB * (T - TmB) / TmB
```

This relates the difference between the free energy densities of the pure solid and pure liquid phases to the latent heat  $L_A$  and the pure component melting point  $T_M^A$ , such that

$$f_A(\phi, T) = \frac{L_A (T - T_M^A)}{T_M^A} p(\phi) + \frac{W_A}{2} g(\phi).$$

With these assumptions

$$\frac{\partial f}{\partial \phi} = (1 - C) \frac{\partial f_A}{\partial \phi} + C \frac{\partial f_B}{\partial \phi}$$

$$= \left\{ (1 - C) \frac{L_A (T - T_M^A)}{T_M^A} + C \frac{L_B (T - T_M^B)}{T_M^B} \right\} p'(\phi) + \left\{ (1 - C) \frac{W_A}{2} + C \frac{W_B}{2} \right\} g'(\phi)$$

and

$$\frac{\partial f}{\partial C} = \left[ f_B(\phi, T) + \frac{RT}{V_m} \ln C \right] - \left[ f_A(\phi, T) + \frac{RT}{V_m} \ln(1 - C) \right]$$

$$= [\mu_B(\phi, C, T) - \mu_A(\phi, C, T)] / V_m$$

where  $\mu_A$  and  $\mu_B$  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  $\phi$

```
>>> 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  $\mu_A$  and  $\mu_B$  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)

```

(continues on next page)

(continued from previous page)

```
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. (23.8) to the canonical form of Eq. (12.2), hence we must expand Eq. (23.9) 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. (23.8) 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. (23.8) 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 `fraction = Cavg = 0.5`, a little bit of algebra reveals that the temperature that leaves the phase fraction unchanged is given by

```
>>> T.setValue((LA + LB) * TmA * TmB / (LA * TmB + LB * TmA))
```

In this simple, binary, ideal solution case, we can derive explicit expressions for the solidus and liquidus compositions. In general, this may not be possible or practical. In that event, the root-finding facilities in SciPy can be used.

We'll need a function to return the two conditions for equilibrium

$$0 = \mu_A(1, C_S, T) - \mu_A(0, C_L, T) = \frac{L_A(T - T_M^A)}{T_M^A} V_m + RT \ln(1 - C_S) - RT \ln(1 - C_L)$$

$$0 = \mu_B(1, C_S, T) - \mu_B(0, C_L, T) = \frac{L_B(T - T_M^B)}{T_M^B} V_m + RT \ln C_S - RT \ln C_L$$

```
>>> def equilibrium(C):
...     return [numerix.array(enthalpyA * Vm
...                           + R * T * numerix.log(1 - C[0])
...                           - R * T * numerix.log(1 - C[1])),
...            numerix.array(enthalpyB * Vm
...                           + R * T * numerix.log(C[0])
...                           - R * T * numerix.log(C[1]))]
```

and we'll have much better luck if we also supply the Jacobian

$$\begin{bmatrix} \frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_S} & \frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_L} \\ \frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_S} & \frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_L} \end{bmatrix} = RT \begin{bmatrix} -\frac{1}{1-C_S} & \frac{1}{1-C_L} \\ \frac{1}{C_S} & -\frac{1}{C_L} \end{bmatrix}$$

```
>>> def equilibriumJacobian(C):
...     return R * T * numerix.array([[ -1. / (1 - C[0]), 1. / (1 - C[1])],
...                                   [ 1. / C[0],      -1. / C[1]])
```

```
>>> try:
...     from scipy.optimize import fsolve
...     CsRoot, ClRoot = fsolve(func=equilibrium, x0=[0.5, 0.5],
...                             fprime=equilibriumJacobian)
... except ImportError:
...     ClRoot = CsRoot = 0
...     print("The SciPy library is not available to calculate the solidus and ...
↳ liquidus concentrations")
```

```
>>> print(Cl.allclose(ClRoot))
1
>>> print(Cs.allclose(CsRoot))
1
```

We plot the result against the sharp interface solution

```
>>> sharp = CellVariable(name="sharp", mesh=mesh)
>>> x = mesh.cellCenters[0]
>>> sharp.setValue(Cs, where=x < L * fraction)
>>> sharp.setValue(Cl, where=x >= L * fraction)
```

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

Because the phase field interface will not move, and because we've seen in earlier examples that the diffusion problem is unconditionally stable, we need take only one very large timestep to reach equilibrium

```
>>> dt = 1.e5
```

Because the phase field equation is coupled to the composition through enthalpy and  $W$  and the diffusion equation is coupled to the phase field through `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 and cannot be solved by the default `LinearPCGSolver`. Therefore, we use a `DefaultAsymmetricSolver` for this equation.

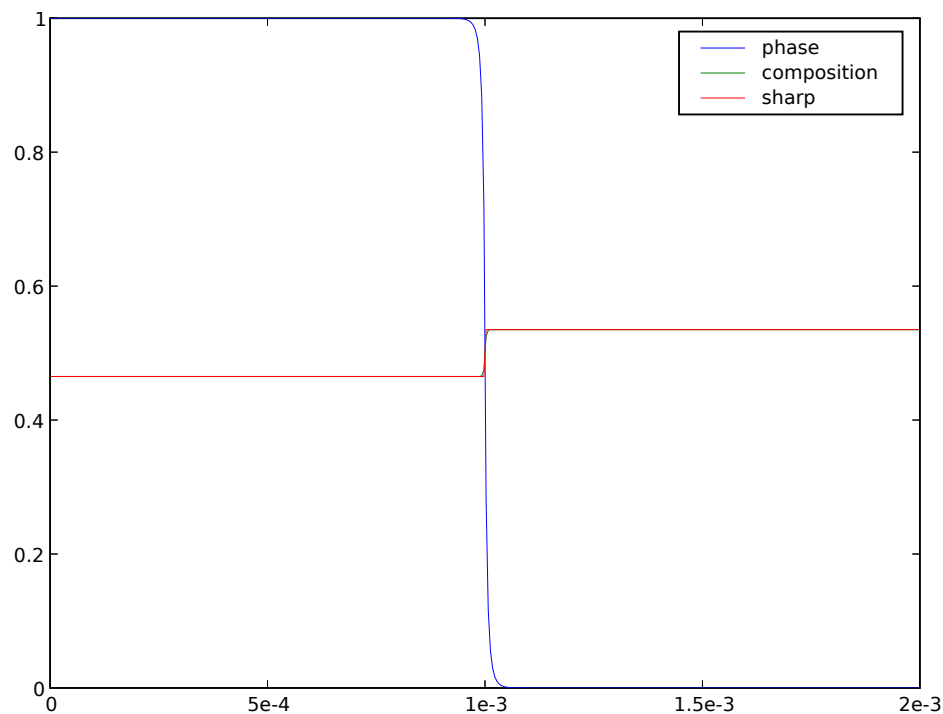
We now use the “`sweep()`” method instead of “`solve()`” because we require the residual.

```
>>> solver = DefaultAsymmetricSolver(tolerance=1e-10)
```

```

>>> phase.updateOld()
>>> C.updateOld()
>>> phaseRes = 1e+10
>>> diffRes = 1e+10
>>> while phaseRes > 1e-3 or diffRes > 1e-3:
...     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. (23.9). If we assume that the phase changes from 1 to 0 in a single grid spacing, that the diffusivity is  $Dl$  at the interface, and that the term due to the difference in barrier heights is negligible:

$$\begin{aligned}\vec{u}_\phi &= \frac{D_\phi}{C} \nabla \phi \\ &\approx \frac{Dl \frac{1}{2} V_m}{RT} \left[ \frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] \frac{1}{\Delta x} \\ &\approx \frac{Dl \frac{1}{2} V_m}{RT} (L_B + L_A) \frac{T_M^A - T_M^B}{T_M^A + T_M^B} \frac{1}{\Delta x} \\ &\approx 0.28 \text{ cm/s}\end{aligned}$$

To get a CFL =  $\vec{u}_\phi \Delta t / \Delta x < 1$ , we need a time step of about  $10^{-5}$  s.

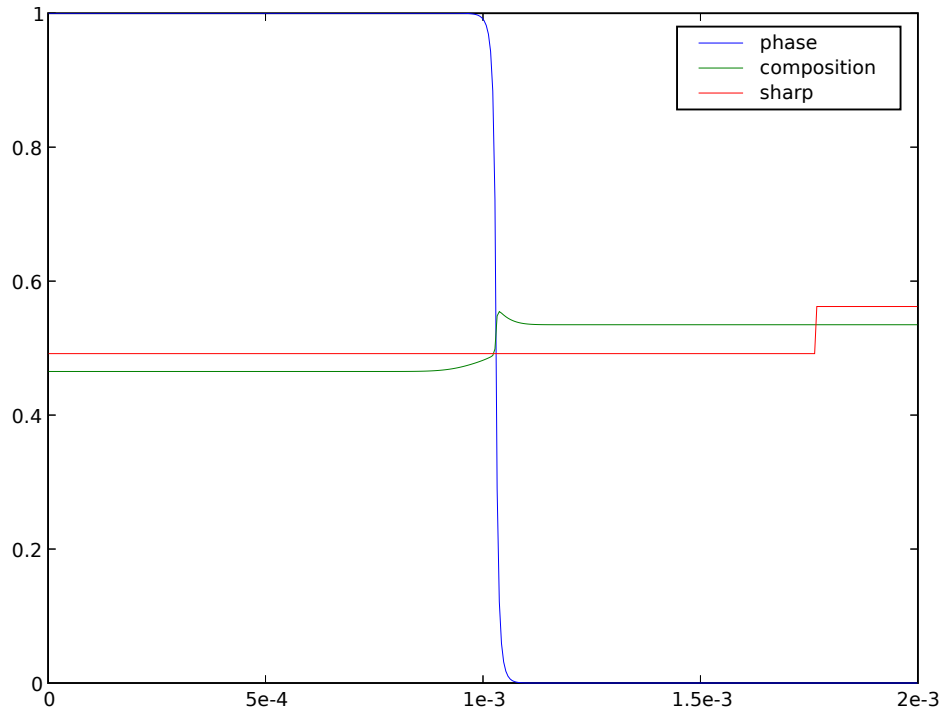
```
>>> dt = 1.e-5
```

```
>>> if __name__ == '__main__':
...     timesteps = 100
... else:
...     timesteps = 10
```

```
>>> from builtins import range
>>> for i in range(timesteps):
...     phase.updateOld()
...     C.updateOld()
...     phaseRes = 1e+10
...     diffRes = 1e+10
...     while phaseRes > 1e-3 or diffRes > 1e-3:
...         phaseRes = phaseEq.sweep(var=phase, dt=dt)
...         diffRes = diffusionEq.sweep(var=C, dt=dt, solver=solver)
...     if __name__ == '__main__':
...         viewer.plot()
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Moving phase field. Press <return> to proceed...")
```





We see that the composition on either side of the interface approach the sharp-interface solidus and liquidus, but it will take a great many more timesteps to reach equilibrium. If we waited sufficiently long, we could again verify the final concentrations and phase fraction against the expected values.

### 23.11.4 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
>>> 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  $\phi$ <sup>1</sup>

<sup>1</sup> We will find that we need to “sweep” this non-linear problem (see e.g. the composition-dependent diffusivity example in *examples*.)

```
>>> 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

$$\frac{\partial \phi}{\partial t} = M_\phi \left( \kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \right) \quad (23.9)$$

for phase and

$$\frac{\partial C}{\partial t} = \nabla \cdot \left( M_C \nabla \frac{\partial f}{\partial C} \right) \quad (23.10)$$

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  $\Omega_S$  and  $\Omega_L$  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

$$f(\phi, C, T) = p(\phi) f^S(C, T) + (1 - p(\phi)) f^L(C, T) + \left[ (1 - C) \frac{W_A}{2} + C \frac{W_B}{2} \right] g(\phi).$$

*diffusion.mesh1D*), so we declare  $\phi$  and  $C$  to retain an “old” value.

<sup>2</sup> we are going to want to examine different temperatures in this example, so we declare  $T$  as a *Variable*

When the thermodynamic models are the same in both phases, both approaches should yield the same result.

We choose the first approach and make the simplifying assumptions of an ideal solution and that

$$f_A^L(T) = 0$$

$$f_A^S(T) - f_A^L(T) = \frac{L_A (T - T_M^A)}{T_M^A}$$

and likewise for component  $B$ .

```
>>> LA = 2350. # J / cm**3
>>> LB = 1728. # J / cm**3
>>> TmA = 1728. # K
>>> TmB = 1358. # K
```

```
>>> enthalpyA = LA * (T - TmA) / TmA
>>> enthalpyB = LB * (T - TmB) / TmB
```

This relates the difference between the free energy densities of the pure solid and pure liquid phases to the latent heat  $L_A$  and the pure component melting point  $T_M^A$ , such that

$$f_A(\phi, T) = \frac{L_A (T - T_M^A)}{T_M^A} p(\phi) + \frac{W_A}{2} g(\phi).$$

With these assumptions

$$\frac{\partial f}{\partial \phi} = (1 - C) \frac{\partial f_A}{\partial \phi} + C \frac{\partial f_B}{\partial \phi}$$

$$= \left\{ (1 - C) \frac{L_A (T - T_M^A)}{T_M^A} + C \frac{L_B (T - T_M^B)}{T_M^B} \right\} p'(\phi) + \left\{ (1 - C) \frac{W_A}{2} + C \frac{W_B}{2} \right\} g'(\phi)$$

and

$$\frac{\partial f}{\partial C} = \left[ f_B(\phi, T) + \frac{RT}{V_m} \ln C \right] - \left[ f_A(\phi, T) + \frac{RT}{V_m} \ln(1 - C) \right]$$

$$= [\mu_B(\phi, C, T) - \mu_A(\phi, C, T)] / V_m$$

where  $\mu_A$  and  $\mu_B$  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  $\phi$

```
>>> 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  $\mu_A$  and  $\mu_B$  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. (23.10) to the canonical form of Eq. (12.2), hence we must expand Eq. (23.11) 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. (23.10) 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. (23.10) 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  $\phi$  with a diffusion coefficient

$$D_\phi(C, \phi) = \frac{D_C C(1 - C)V_m}{RT} \left\{ \left[ \frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] p'(\phi) + \frac{W_B - W_A}{2} g'(\phi) \right\},$$

such that

$$\frac{\partial C}{\partial t} = \nabla \cdot (D_C \nabla C) + \nabla \cdot (D_\phi \nabla \phi)$$

or

```

>>> D1 = Variable(value=1e-5) # cm**2 / s
>>> Ds = Variable(value=1e-9) # cm**2 / s
>>> Dc = (Ds - D1) * phase.arithmeticFaceValue + D1
    
```

```

>>> Dphi = ((Dc * C.harmonicFaceValue * (1 - C.harmonicFaceValue) * Vm / (R * T))
...         * ((enthalpyB - enthalpyA) * pPrime(phase.arithmeticFaceValue)
...         + 0.5 * (WB - WA) * gPrime(phase.arithmeticFaceValue)))
    
```

```
>>> diffusionEq = (TransientTerm(var=C)
...                 == DiffusionTerm(coeff=Dc, var=C)
...                 + DiffusionTerm(coeff=Dphi, var=phase))
```

```
>>> eq = phaseEq & diffusionEq
```

We initialize the phase field to a step function in the middle of the domain

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=mesh.cellCenters[0] > L/2.)
```

and start with a uniform composition field  $C = 1/2$

```
>>> C.setValue(0.5)
```

In equilibrium,  $\mu_A(0, C_L, T) = \mu_A(1, C_S, T)$  and  $\mu_B(0, C_L, T) = \mu_B(1, C_S, T)$  and, for ideal solutions, we can deduce the liquidus and solidus compositions as

$$C_L = \frac{1 - \exp\left(-\frac{L_A(T-T_M^A)}{T_M^A} \frac{V_m}{RT}\right)}{\exp\left(-\frac{L_B(T-T_M^B)}{T_M^B} \frac{V_m}{RT}\right) - \exp\left(-\frac{L_A(T-T_M^A)}{T_M^A} \frac{V_m}{RT}\right)}$$

$$C_S = \exp\left(-\frac{L_B(T-T_M^B)}{T_M^B} \frac{V_m}{RT}\right) C_L$$

```
>>> Cl = ((1. - numerix.exp(-enthalpyA * Vm / (R * T)))
...       / (numerix.exp(-enthalpyB * Vm / (R * T)) - numerix.exp(-enthalpyA * Vm / (R *
... T))))
>>> Cs = numerix.exp(-enthalpyB * Vm / (R * T)) * Cl
```

The phase fraction is predicted by the lever rule

```
>>> Cavg = C.cellVolumeAverage
>>> fraction = (Cl - Cavg) / (Cl - Cs)
```

For the special case of  $\text{fraction} = \text{Cavg} = 0.5$ , a little bit of algebra reveals that the temperature that leaves the phase fraction unchanged is given by

```
>>> T.setValue((LA + LB) * TmA * TmB / (LA * TmB + LB * TmA))
```

In this simple, binary, ideal solution case, we can derive explicit expressions for the solidus and liquidus compositions. In general, this may not be possible or practical. In that event, the root-finding facilities in SciPy can be used.

We'll need a function to return the two conditions for equilibrium

$$0 = \mu_A(1, C_S, T) - \mu_A(0, C_L, T) = \frac{L_A(T - T_M^A)}{T_M^A} V_m + RT \ln(1 - C_S) - RT \ln(1 - C_L)$$

$$0 = \mu_B(1, C_S, T) - \mu_B(0, C_L, T) = \frac{L_B(T - T_M^B)}{T_M^B} V_m + RT \ln C_S - RT \ln C_L$$

```
>>> def equilibrium(C):
...     return [numerix.array(enthalpyA * Vm
...                           + R * T * numerix.log(1 - C[0])
...                           - R * T * numerix.log(1 - C[1])),
...            numerix.array(enthalpyB * Vm
...                           + R * T * numerix.log(C[0])
...                           - R * T * numerix.log(C[1]))]
```

and we'll have much better luck if we also supply the Jacobian

$$\begin{bmatrix} \frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_S} & \frac{\partial(\mu_A^S - \mu_A^L)}{\partial C_L} \\ \frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_S} & \frac{\partial(\mu_B^S - \mu_B^L)}{\partial C_L} \end{bmatrix} = RT \begin{bmatrix} -\frac{1}{1-C_S} & \frac{1}{1-C_L} \\ \frac{1}{C_S} & -\frac{1}{C_L} \end{bmatrix}$$

```
>>> def equilibriumJacobian(C):
...     return R * T * numerix.array([[ -1. / (1 - C[0]), 1. / (1 - C[1])],
...                                  [ 1. / C[0], -1. / C[1]])]
```

```
>>> try:
...     from scipy.optimize import fsolve
...     CsRoot, ClRoot = fsolve(func=equilibrium, x0=[0.5, 0.5],
...                             fprime=equilibriumJacobian)
... except ImportError:
...     ClRoot = CsRoot = 0
...     print("The SciPy library is not available to calculate the solidus and ...
↳ liquidus concentrations")
```

```
>>> print(Cl.allclose(ClRoot))
1
>>> print(Cs.allclose(CsRoot))
1
```

We plot the result against the sharp interface solution

```
>>> sharp = CellVariable(name="sharp", mesh=mesh)
>>> x = mesh.cellCenters[0]
>>> sharp.setValue(Cs, where=x < L * fraction)
>>> sharp.setValue(Cl, where=x >= L * fraction)
```

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

Because the phase field interface will not move, and because we've seen in earlier examples that the diffusion problem is unconditionally stable, we need take only one very large timestep to reach equilibrium

```
>>> dt = 1.e5
```

Because the phase field equation is coupled to the composition through enthalpy and  $\bar{W}$  and the diffusion equation is coupled to the phase field through  $\text{phaseTransformationVelocity}$ , it is necessary sweep this non-linear problem to convergence. We use the “residual” of the equations (a measure of how well they think they have solved the given set of linear equations) as a test for how long to sweep.

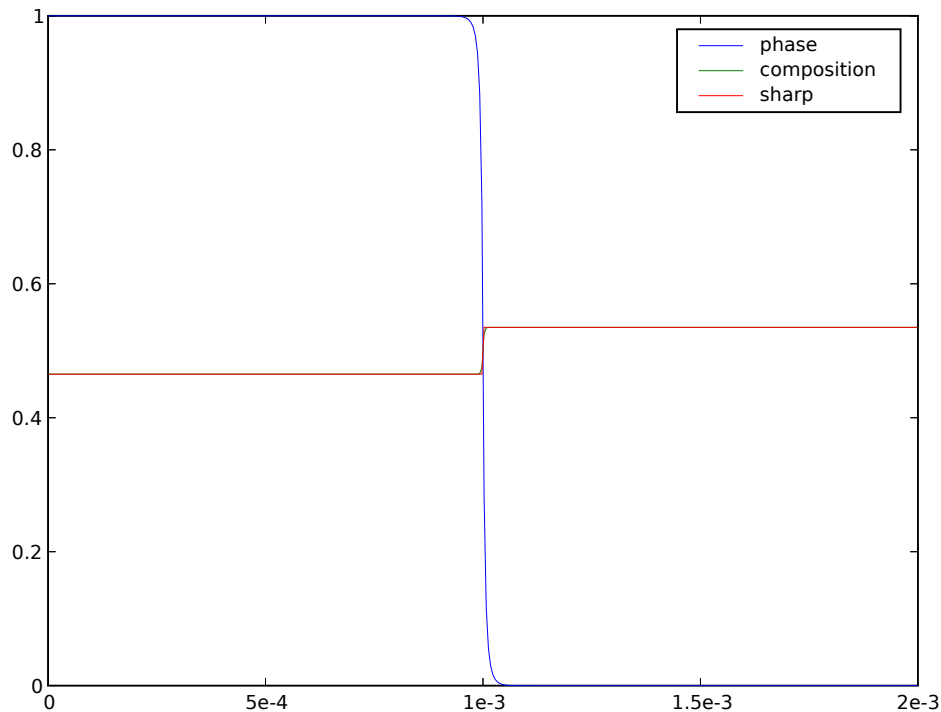
We now use the “`sweep()`” method instead of “`solve()`” because we require the residual.

```
>>> solver = LinearLUSolver(tolerance=1e-10)
```

```
>>> phase.updateOld()
>>> C.updateOld()
>>> res = 1.
>>> initialRes = None
>>> sweep = 0
```

```
>>> while res > 1e-4 and sweep < 20:
...     res = eq.sweep(dt=dt, solver=solver)
...     if initialRes is None:
...         initialRes = res
...     res = res / initialRes
...     sweep += 1
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Stationary phase field. Press <return> to proceed...")
```



We verify that the bulk phases have shifted to the predicted solidus and liquidus compositions

```
>>> X = mesh.faceCenters[0]
>>> print(Cs.allclose(C.faceValue[X.value==0], atol=1e-2))
```

(continues on next page)



(continued from previous page)

```
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. (23.11) (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  $Dl$  at the interface, and that the term due to the difference in barrier heights is negligible:

$$\begin{aligned}\vec{u}_\phi &= \frac{D_\phi}{C} \nabla \phi \\ &\approx \frac{Dl \frac{1}{2} V_m}{RT} \left[ \frac{L_B (T - T_M^B)}{T_M^B} - \frac{L_A (T - T_M^A)}{T_M^A} \right] \frac{1}{\Delta x} \\ &\approx \frac{Dl \frac{1}{2} V_m}{RT} (L_B + L_A) \frac{T_M^A - T_M^B}{T_M^A + T_M^B} \frac{1}{\Delta x} \\ &\approx 0.28 \text{ cm/s}\end{aligned}$$

To get a  $\text{CFL} = \vec{u}_\phi \Delta t / \Delta x < 1$ , we need a time step of about  $10^{-5}$  s.

```
>>> dt = 1.e-5
```

```
>>> if __name__ == '__main__':
...     timesteps = 100
... else:
...     timesteps = 10
```

```
>>> from builtins import range
>>> for i in range(timesteps):
...     phase.updateOld()
```

(continues on next page)

(continued from previous page)

```

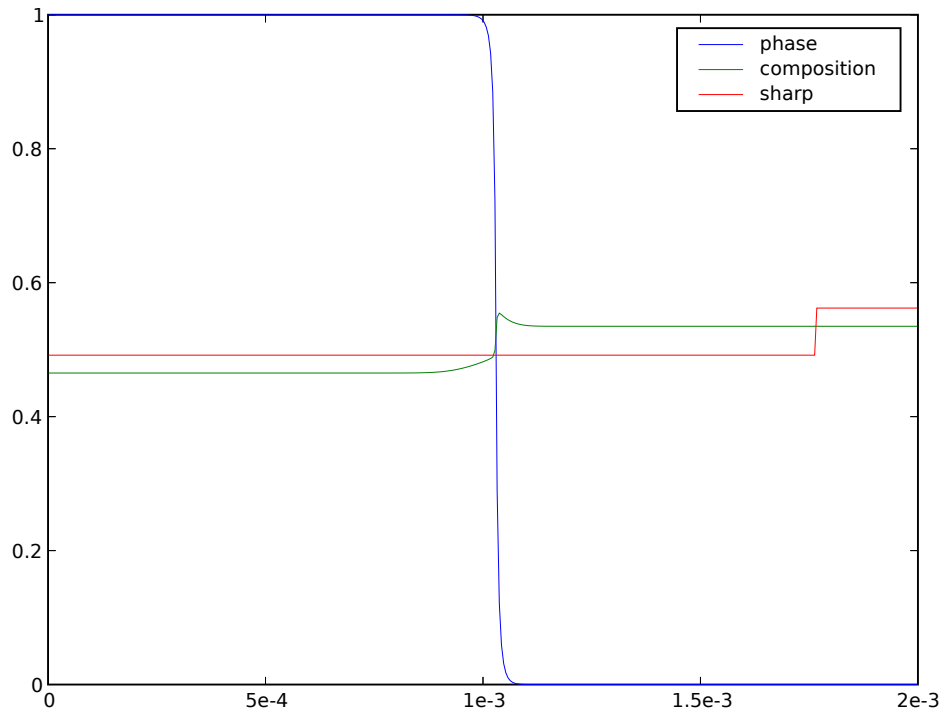
...     C.updateOld()
...     res = 1e+10
...     sweep = 0
...     while res > 1e-3 and sweep < 20:
...         res = eq.sweep(dt=dt, solver=solver)
...         sweep += 1
...     if __name__ == '__main__':
...         viewer.plot()

```

```

>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Moving phase field. Press <return> to proceed...")

```



We see that the composition on either side of the interface approach the sharp-interface solidus and liquidus, but it will take a great many more timesteps to reach equilibrium. If we waited sufficiently long, we could again verify the final concentrations and phase fraction against the expected values.

## 23.11.5 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
```

(continues on next page)

(continued from previous page)

```
>>> mu = 1e3
>>> thetaTransientCoeff = 0.01
>>> gamma= 1e3
>>> epsilon = 0.008
>>> s = 0.01
>>> alpha = 0.015
```

The system is held isothermal at

```
>>> temperature = 10.
```

and is initialized to liquid everywhere

```
>>> phase = CellVariable(name='phase field', mesh=mesh)
```

The orientation is initialized to a uniform value to denote the randomly oriented liquid phase

```
>>> theta = ModularVariable(
...     name='theta',
...     mesh=mesh,
...     value=-numerix.pi + 0.0001,
...     hasOld=1
... )
```

Four different solid circular domains are created at each corner of the domain with appropriate orientations

```
>>> x, y = mesh.cellCenters
>>> for a, b, thetaValue in ((0., 0., 2. * numerix.pi / 3.),
...                          (L, 0., -2. * numerix.pi / 3.),
...                          (0., L, -2. * numerix.pi / 3. + 0.3),
...                          (L, L, 2. * numerix.pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)
```

The phase equation is built in the following way. The source term is linearized in the manner demonstrated in [examples.phase.simple](#) (Kobayashi, semi-implicit). Here we use a function to build the equation, so that it can be reused later.

```
>>> def buildPhaseEquation(phase, theta):
...
...     mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
...     thetaMag = theta.old.grad.mag
...     implicitSource = mPhiVar * (phase - (mPhiVar < 0))
...     implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
...
...     return TransientTerm(phaseTransientCoeff) == \
...         ExplicitDiffusionTerm(alpha**2) \
...         - ImplicitSourceTerm(implicitSource) \
...         + (mPhiVar > 0) * mPhiVar * phase
```

```
>>> phaseEq = buildPhaseEquation(phase, theta)
```

The theta equation is built in the following way. The details for this equation are fairly involved, see J. A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of theta on the circle. The source term requires the evaluation of the face gradient without the modular operators.

```
>>> def buildThetaEquation(phase, theta):
...
...     phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
...     phaseModSq = phaseMod * phaseMod
...     expo = epsilon * beta * theta.grad.mag
...     expo = (expo < 100.) * (expo - 100.) + 100.
...     pFunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
...
...     phaseFace = phase.arithmeticFaceValue
...     phaseSq = phaseFace * phaseFace
...     gradMag = theta.faceGrad.mag
...     eps = 1. / gamma / 10.
...     gradMag += (gradMag < eps) * eps
...     IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
...     diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
...
...     thetaGradDiff = theta.faceGrad - theta.faceGradNoMod
...     sourceCoeff = (diffusionCoeff * thetaGradDiff).divergence
...
...     return TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...         DiffusionTerm(diffusionCoeff) \
...         + sourceCoeff
```

```
>>> thetaEq = buildThetaEquation(phase, theta)
```

If the example is run interactively, we create viewers for the phase and orientation variables. Rather than viewing the raw orientation, which is not meaningful in the liquid phase, we weight the orientation by the phase

```
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase, datamin=0., datamax=1.)
...     thetaProd = -numerix.pi + phase * (theta + numerix.pi)
...     thetaProductViewer = Viewer(vars=thetaProd,
...                                 datamin=-numerix.pi, datamax=numerix.pi)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

The solution will be tested against data that was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `mesh20x20.gz` extracts the data and compares it with the *theta* variable.

```
>>> import os
>>> from future.utils import text_to_native_str
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + text_to_native_str('.gz
→')).flat
```

We step the solution in time, plotting as we go if running interactively,

```
>>> from builtins import range
>>> for i in range(steps):
...     theta.updateOld()
```

(continues on next page)

(continued from previous page)

```

...     thetaEq.solve(theta, dt=timeStepDuration, solver=GeneralSolver(iterations=2000,
↳tolerance=1e-15))
...     phaseEq.solve(phase, dt=timeStepDuration, solver=GeneralSolver(iterations=2000,
↳tolerance=1e-15))
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()

```

The solution is compared against Ryo Kobayashi's test data

```

>>> print(theta.allclose(testData, rtol=1e-7, atol=1e-7))
1

```

The following code shows how to restart a simulation from some saved data. First, reset the variables to their original values.

```

>>> phase.setValue(0)
>>> theta.setValue(-numerix.pi + 0.0001)
>>> x, y = mesh.cellCenters
>>> for a, b, thetaValue in ((0., 0., 2. * numerix.pi / 3.),
...                          (L, 0., -2. * numerix.pi / 3.),
...                          (0., L, -2. * numerix.pi / 3. + 0.3),
...                          (L, L, 2. * numerix.pi / 3.)):
...     segment = (x - a)**2 + (y - b)**2 < (L / 2.)**2
...     phase.setValue(1., where=segment)
...     theta.setValue(thetaValue, where=segment)

```

Step through half the time steps.

```

>>> from builtins import range
>>> for i in range(steps // 2):
...     theta.updateOld()
...     thetaEq.solve(theta, dt=timeStepDuration, solver=GeneralSolver(iterations=2000,
↳tolerance=1e-15))
...     phaseEq.solve(phase, dt=timeStepDuration, solver=GeneralSolver(iterations=2000,
↳tolerance=1e-15))

```

We confirm that the solution has not yet converged to that given by Ryo Kobayashi's FORTRAN code:

```

>>> print(theta.allclose(testData))
0

```

We save the variables to disk.

```

>>> (f, filename) = dump.write({'phase' : phase, 'theta' : theta}, extension = '.gz')

```

and then recall them to test the data pickling mechanism

```

>>> data = dump.read(filename, f)
>>> newPhase = data['phase']
>>> newTheta = data['theta']
>>> newThetaEq = buildThetaEquation(newPhase, newTheta)
>>> newPhaseEq = buildPhaseEquation(newPhase, newTheta)

```

and finish the iterations,

```
>>> from builtins import range
>>> for i in range(steps // 2):
...     newTheta.updateOld()
...     newThetaEq.solve(newTheta, dt=timeStepDuration,
↳ solver=GeneralSolver(iterations=2000, tolerance=1e-15))
...     newPhaseEq.solve(newPhase, dt=timeStepDuration,
↳ solver=GeneralSolver(iterations=2000, tolerance=1e-15))
```

The solution is compared against Ryo Kobayashi's test data

```
>>> print(newTheta.allclose(testData, rtol=1e-7))
1
```

### 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  $0 \leq x \leq L_x/2$  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  $2\pi$ -periodicity, we must use `ModularVariable` instead of a `CellVariable`. A `ModularVariable` confines `theta` to  $-\pi < \theta \leq \pi$  by adding or subtracting  $2\pi$  where necessary and by defining a new subtraction operator between two angles.

```
>>> theta = ModularVariable(
...     name='theta',
...     mesh=mesh,
...     value=1.,
...     hasOld=1
... )
```

The left and right halves of the domain are given different orientations.

```
>>> theta.setValue(0., where=mesh.cellCenters[0] > Lx / 2.)
```

The phase equation is built in the following way.

```
>>> mPhiVar = phase - 0.5 + temperature * phase * (1 - phase)
```

The source term is linearized in the manner demonstrated in [examples.phase.simple](#) (Kobayashi, semi-implicit).

```
>>> thetaMag = theta.old.grad.mag
>>> implicitSource = mPhiVar * (phase - (mPhiVar < 0))
>>> implicitSource += (2 * s + epsilon**2 * thetaMag) * thetaMag
```

The phase equation is constructed.

```
>>> phaseEq = TransientTerm(phaseTransientCoeff) \
... == ExplicitDiffusionTerm(alpha**2) \
...     - ImplicitSourceTerm(implicitSource) \
...     + (mPhiVar > 0) * mPhiVar * phase
```



The theta equation is built in the following way. The details for this equation are fairly involved, see J. A. Warren *et al.*. The main detail is that a source must be added to correct for the discretization of theta on the circle.

```
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> phaseModSq = phaseMod * phaseMod
>>> expo = epsilon * beta * theta.grad.mag
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> pFunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
```

```
>>> phaseFace = phase.arithmeticFaceValue
>>> phaseSq = phaseFace * phaseFace
>>> gradMag = theta.faceGrad.mag
>>> eps = 1. / gamma / 10.
>>> gradMag += (gradMag < eps) * eps
>>> IGamma = (gradMag > 1. / gamma) * (1 / gradMag - gamma) + gamma
>>> diffusionCoeff = phaseSq * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. `theta.faceGradNoMod` evaluates the gradient without modular arithmetic.

```
>>> thetaGradDiff = theta.faceGrad - theta.faceGradNoMod
>>> sourceCoeff = (diffusionCoeff * thetaGradDiff).divergence
```

Finally the theta equation can be constructed.

```
>>> thetaEq = TransientTerm(thetaTransientCoeff * phaseModSq * pFunc) == \
...     DiffusionTerm(diffusionCoeff) \
...     + sourceCoeff
```

If the example is run interactively, we create viewers for the phase and orientation variables.

```
>>> if __name__ == '__main__':
...     phaseViewer = Viewer(vars=phase, datamin=0., datamax=1.)
...     thetaProductViewer = Viewer(vars=theta,
...                                 datamin=-numerix.pi, datamax=numerix.pi)
...     phaseViewer.plot()
...     thetaProductViewer.plot()
```

we iterate the solution in time, plotting as we go if running interactively,

```
>>> steps = 10
>>> from builtins import range
>>> for i in range(steps):
...     theta.updateOld()
...     thetaEq.solve(theta, dt = timeStepDuration)
...     phaseEq.solve(phase, dt = timeStepDuration)
...     if __name__ == '__main__':
...         phaseViewer.plot()
...         thetaProductViewer.plot()
```

The solution is compared with test data. The test data was created with `steps = 10` with a FORTRAN code written by Ryo Kobayashi for phase field modeling. The following code opens the file `mesh40x1.gz` extracts the data and compares it with the theta variable.

```

>>> import os
>>> from future.utils import text_to_native_str
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + text_to_native_str('.gz'))
>>> testData = CellVariable(mesh=mesh, value=testData)
>>> print(theta.allclose(testData))
1

```

`examples.phase.impingement.test`

## 23.11.6 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.

```

>>> from builtins import range
>>> for step in range(steps):
...     phaseEq.solve(phase, dt = timeStepDuration)

```

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
>>> from future.utils import text_to_native_str
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + text_to_native_str('.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.

```
>>> from builtins import range
>>> for step in range(steps):
...     phaseEq.solve(phase, dt = timeStepDuration)
```

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
>>> from future.utils import text_to_native_str
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + text_to_native_str('.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.

```
>>> from builtins import range
>>> for step in range(steps):
...     phaseEq.solve(phase, dt = timeStepDuration)
```

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
>>> from future.utils import text_to_native_str
>>> testData = numerix.loadtxt(os.path.splitext(__file__)[0] + text_to_native_str('.gz'))
>>> print(phase.allclose(testData))
1
```

### examples.phase.missOrientation.test

#### 23.11.7 examples.phase.polyxtal

Solve the dendritic growth of nuclei and subsequent grain impingement.

To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as “undercooling” or “supercooling”). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns. Warren, Kobayashi, Lobkovsky and Carter [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  $\phi$  (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  $\Delta T$  ( $\Delta T = 0$  at the melting point)

```
>>> dT = CellVariable(name=r'\Delta T', mesh=mesh, hasOld=True)
```

and an orientation  $-\pi < \theta \leq \pi$

```
>>> theta = ModularVariable(name=r'\theta', mesh=mesh, hasOld=True)
>>> theta.value = -numerix.pi + 0.0001
```

The hasOld flag causes the storage of the value of variable from the previous timestep. This is necessary for solving equations with non-linear coefficients or for coupling between PDEs.

The governing equation for the temperature field is the heat flux equation, with a source due to the latent heat of solidification

$$\frac{\partial \Delta T}{\partial t} = D_T \nabla^2 \Delta T + \frac{\partial \phi}{\partial t} + c(T_0 - T)$$

```
>>> DT = 2.25
>>> q = Variable(0.)
>>> T_0 = -0.1
>>> heatEq = (TransientTerm()
...          == DiffusionTerm(DT)
...          + (phase - phase.old) / dt
...          + q * T_0 - ImplicitSourceTerm(q))
```

The governing equation for the phase field is

$$\tau_\phi \frac{\partial \phi}{\partial t} = \nabla \cdot D \nabla \phi + \phi(1 - \phi)m(\phi, \Delta T) - 2s\phi|\nabla\theta| - \epsilon^2\phi|\nabla\theta|^2$$

where

$$m(\phi, \Delta T) = \phi - \frac{1}{2} - \frac{\kappa_1}{\pi} \arctan(\kappa_2 \Delta T)$$

represents a source of anisotropy. The coefficient D is an anisotropic diffusion tensor in two dimensions

$$D = \alpha^2 (1 + c\beta) \begin{bmatrix} 1 + c\beta & -c\frac{\partial\beta}{\partial\psi} \\ c\frac{\partial\beta}{\partial\psi} & 1 + c\beta \end{bmatrix}$$

where  $\beta = \frac{1-\Phi^2}{1+\Phi^2}$ ,  $\Phi = \tan\left(\frac{N}{2}\psi\right)$ ,  $\psi = \theta + \arctan\frac{\partial\phi/\partial y}{\partial\phi/\partial x}$ ,  $\theta$  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
```

(continues on next page)

(continued from previous page)

```
>>> 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:

```
>>> from builtins import zip
>>> if __name__ == "__main__":
...     try:
...         class OrientationViewer(Matplotlib2DGridViewer):
...             def __init__(self, phase, orientation, title=None, limits={},
... **kwlimits):
...                 self.phase = phase
...                 Matplotlib2DGridViewer.__init__(self, vars=(orientation),
```

(continues on next page)

(continued from previous page)

```

↪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):
...         from matplotlib import colors
...
...         hsv = numerix.empty(orientation.shape + (3,))
...         hsv[..., 0] = (orientation / numerix.pi + 1) / 2.
...         hsv[..., 1] = 1.
...         hsv[..., 2] = phase

```

(continues on next page)



(continued from previous page)

```

...         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.

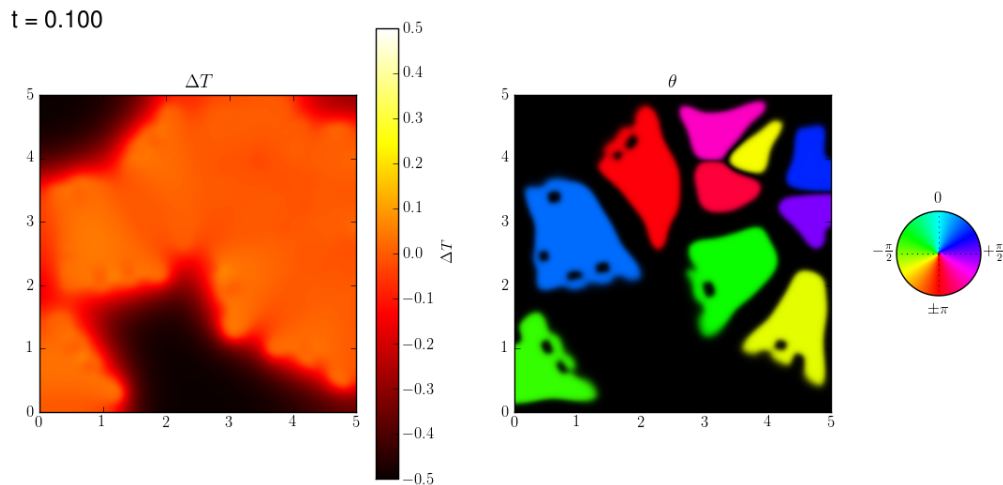
```

(continues on next page)

(continued from previous page)

```
>>> 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.

### 23.11.8 examples.phase.polyxtalCoupled

Simultaneously solve the dendritic growth of nuclei and subsequent grain impingement.

To convert a liquid material to a solid, it must be cooled to a temperature below its melting point (known as “undercooling” or “supercooling”). The rate of solidification is often assumed (and experimentally found) to be proportional to the undercooling. Under the right circumstances, the solidification front can become unstable, leading to dendritic patterns. Warren, Kobayashi, Lobkovsky and Carter [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  $\phi$  (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  $\Delta T$  ( $\Delta T = 0$  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)
...         + 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  $\beta = \frac{1-\Phi^2}{1+\Phi^2}$ ,  $\Phi = \tan\left(\frac{N}{2}\psi\right)$ ,  $\psi = \theta + \arctan\frac{\partial\phi/\partial y}{\partial\phi/\partial x}$ ,  $\theta$  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
```

(continues on next page)

(continued from previous page)

```
>>> gamma = 1e3
>>> thetaSmallValue = 1e-6
>>> phaseMod = phase + ( phase < thetaSmallValue ) * thetaSmallValue
>>> beta_theta = 1e5
>>> expo = epsilon * beta_theta * theta.grad.mag
>>> expo = (expo < 100.) * (expo - 100.) + 100.
>>> Pfunc = 1. + numerix.exp(-expo) * (mu / epsilon - 1.)
```

```
>>> gradMagTheta = theta.faceGrad.mag
>>> eps = 1. / gamma / 10.
>>> gradMagTheta += (gradMagTheta < eps) * eps
>>> IGamma = (gradMagTheta > 1. / gamma) * (1 / gradMagTheta - gamma) + gamma
>>> v_theta = phase.arithmeticFaceValue * (s * IGamma + epsilon**2)
>>> D_theta = phase.arithmeticFaceValue**2 * (s * IGamma + epsilon**2)
```

The source term requires the evaluation of the face gradient without the modular operator. `theta.faceGradNoMod` evaluates the gradient without modular arithmetic.

```
>>> thetaEq = (TransientTerm(coeff=tau_theta * phaseMod**2 * Pfunc, var=theta)
...           == DiffusionTerm(coeff=D_theta, var=theta)
...           + PowerLawConvectionTerm(coeff=v_theta * (theta.faceGrad - theta.
↳ faceGradNoMod), var=phase))
```

We seed a circular solidified region in the center

```
>>> x, y = mesh.cellCenters
>>> numSeeds = 10
>>> numerix.random.seed(12345)
>>> for Cx, Cy, orientation in numerix.random.random([numSeeds, 3]):
...     radius = dx * 5.
...     seed = ((x - Cx * nx * dx)**2 + (y - Cy * ny * dy)**2) < radius**2
...     phase[seed] = 1.
...     theta[seed] = numerix.pi * (2 * orientation - 1)
```

and quench the entire simulation domain below the melting point

```
>>> dT.setValue(-0.5)
```

In a real solidification process, dendritic branching is induced by small thermal fluctuations along an otherwise smooth surface, but the granularity of the *Mesh* is enough “noise” in this case, so we don’t need to explicitly introduce randomness, the way we did in the Cahn-Hilliard problem.

FiPy’s viewers are utilitarian, striving to let the user see *something*, regardless of their operating system or installed packages, so you the default color scheme of grain orientation won’t be very informative “out of the box”. Because all of Python is accessible and FiPy is object oriented, it is not hard to adapt one of the existing viewers to create a specialized display:

```
>>> from builtins import zip
>>> if __name__ == "__main__":
...     try:
...         class OrientationViewer(Matplotlib2DGridViewer):
...             def __init__(self, phase, orientation, title=None, limits={},
↳ **kwargs):
```

(continues on next page)

(continued from previous page)

```

...         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):
...             from matplotlib import colors
...
...             hsv = numerix.empty(orientation.shape + (3,))
...             hsv[..., 0] = (orientation / numerix.pi + 1) / 2.

```

(continues on next page)

(continued from previous page)

```

...         hsv[..., 1] = 1.
...         hsv[..., 2] = phase
...
...         return colors.hsv_to_rgb(hsv)
...
...     @property
...     def _data(self):
...         """convert phase and orientation to rgb image array
...
...         orientation (-pi, pi) -> hue (0, 1)
...         phase (0, 1) -> value (0, 1)
...         """
...         orientation = self._reshape(self.vars[0])
...         phase = self._reshape(self.phase)
...
...         return self._orientation_and_phase_to_rgb(orientation, phase)
...
...     def _plot(self):
...         self.image.set_data(self._data)
...
...     from matplotlib import pyplot
...     pyplot.ion()
...     w, h = pyplot.figaspect(1.)
...     fig = pyplot.figure(figsize=(2*w, h))
...     timer = fig.text(0.1, 0.9, "t = %.3f" % 0, fontsize=18)
...
...     viewer = MultiViewer(viewers=(MatplotlibViewer(vars=dT,
...                                                    cmap=pyplot.cm.hot,
...                                                    datamin=-0.5,
...                                                    datamax=0.5,
...                                                    axes=fig.add_subplot(121)),
...                                OrientationViewer(phase=phase,
...                                                  orientation=theta,
...                                                  title=theta.name,
...                                                  axes=fig.add_subplot(122))))
...
...     except ImportError:
...         viewer = MultiViewer(viewers=(Viewer(vars=dT,
...                                             datamin=-0.5,
...                                             datamax=0.5),
...                                     Viewer(vars=phase,
...                                             datamin=0.,
...                                             datamax=1.),
...                                     Viewer(vars=theta,
...                                             datamin=-numerix.pi,
...                                             datamax=numerix.pi)))
...
...     viewer.plot()

```

and iterate the solution in time, plotting as we go,

```
>>> eq = thetaEq & phaseEq & heatEq
```

```
>>> if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```

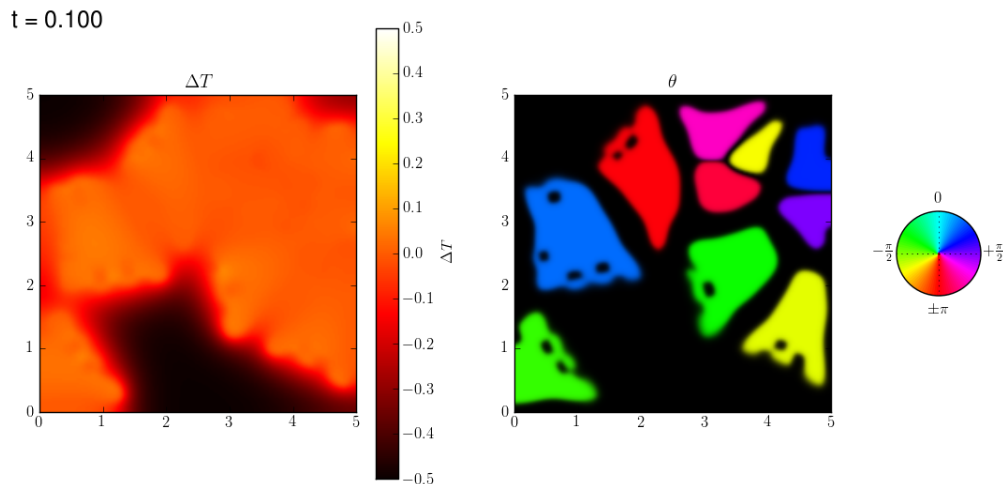
...     total_time = 2.
... else:
...     total_time = dt * 10
>>> elapsed = 0.
>>> save_interval = 0.002
>>> save_at = save_interval

```

```

>>> 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.

### 23.11.9 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  $\phi$

```
>>> 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  $\mu_j$  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  $\phi$

```
>>> def pPrime(phi):
...     return 30. * g(phi)
```

```
>>> def gPrime(phi):
...     return 2. * phi * (1 - phi) * (1 - 2 * phi)
```

We “cook” the standard potentials to give the desired solid and liquid concentrations, with a solid phase rich in interstitials and the solvent and a liquid phase rich in the two substitutional species.

```
>>> interstitials[0].S = 0.3
>>> interstitials[0].L = 0.4
>>> substitutionals[0].S = 0.4
>>> substitutionals[0].L = 0.3
>>> substitutionals[1].S = 0.2
>>> substitutionals[1].L = 0.1
>>> solvent.S = 1.
>>> solvent.L = 1.
>>> for Cj in substitutionals:
...     solvent.S -= Cj.S
...     solvent.L -= Cj.L
```

```
>>> rhoS = rhoL = 1.
>>> for Cj in interstitials:
...     rhoS += Cj.S
...     rhoL += Cj.L
```

```
>>> for Cj in interstitials + substitutionals + [solvent]:
...     Cj.standardPotential = R * T * (numerix.log(Cj.L/rhoL)
...                                     - numerix.log(Cj.S/rhoS))
```

```
>>> for Cj in interstitials:
...     Cj.diffusivity = 1.
...     Cj.barrier = 0.
```

```
>>> for Cj in substitutionals:
...     Cj.diffusivity = 1.
...     Cj.barrier = R * T
```

```
>>> solvent.barrier = R * T
```

We create the phase equation

$$\frac{1}{M_\phi} \frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \sum_{j=0}^N C_j \left[ \mu_j^{\circ SL}(T) p'(\phi) + \frac{W_j}{2} g'(\phi) \right]$$

with a semi-implicit source just as in [examples.phase.simple](#) and [examples.phase.binary](#)

```
>>> enthalpy = 0.
>>> barrier = 0.
>>> for Cj in interstitials + substitutionals + [solvent]:
...     enthalpy += Cj * Cj.standardPotential
...     barrier += Cj * Cj.barrier
```

```
>>> mPhi = -((1 - 2 * phase) * barrier + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * barrier - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
```

```
>>> phase.mobility = 1.
>>> phase.gradientEnergy = 25
>>> phase.equation = TransientTerm(coeff=1/phase.mobility) \
... == DiffusionTerm(coeff=phase.gradientEnergy) \
...     + S0 + ImplicitSourceTerm(coeff = S1)
```

We could construct the diffusion equations one-by-one, in the manner of [examples.phase.binary](#), but it is better to take advantage of the full scripting power of the Python language, where we can easily loop over components or even

make “factory” functions if we desire. For the interstitial diffusion equations, we arrange in canonical form as before:

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = D_j \underbrace{\nabla^2 C_j}_{\text{diffusion}} + D_j \underbrace{\nabla \cdot \frac{C_j}{1 + \sum_{\substack{k=0 \\ k \neq j}}^M C_k} \left\{ \overbrace{\frac{\rho}{RT} \left[ \mu_j^{\circ SL} \nabla p(\phi) + \frac{W_j}{2} \nabla g(\phi) \right]}^{\text{phase transformation}} - \overbrace{\sum_{\substack{i=0 \\ i \neq j}}^M \nabla C_i}_{\text{counter diffusion}} \right\}}_{\text{convection}}$$

```

>>> 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

$$\underbrace{\frac{\partial C_j}{\partial t}}_{\text{transient}} = D_j \underbrace{\nabla^2 C_j}_{\text{diffusion}} + D_j \underbrace{\nabla \cdot \frac{C_j}{1 - \sum_{\substack{k=M+1 \\ k \neq j}}^{N-1} C_k} \left\{ \overbrace{\frac{C_N}{RT} \left[ (\mu_j^{\circ SL} - \mu_N^{\circ SL}) \nabla p(\phi) + \frac{W_j - W_N}{2} \nabla g(\phi) \right]}^{\text{phase transformation}} + \overbrace{\sum_{\substack{i=M+1 \\ i \neq j}}^{N-1} \nabla C_i}_{\text{counter diffusion}} \right\}}_{\text{convection}}$$

```

>>> 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)
...
    
```

(continues on next page)

(continued from previous page)

```

...     CkSum = CellVariable(mesh=mesh, value=0.)
...     for Ck in [Ck for Ck in substitutionals if Ck is not Cj]:
...         CkSum += Ck
...
...     counterDiffusion = CkSum.faceGrad
...
...     convectionCoeff = counterDiffusion + phaseTransformation
...     convectionCoeff *= (Cj.diffusivity
...                          / (1. - CkSum.harmonicFaceValue))
...
...     Cj.equation = (TransientTerm()
...                   == DiffusionTerm(coeff=Cj.diffusivity)
...                   + PowerLawConvectionTerm(coeff=convectionCoeff))

```

We start with a sharp phase boundary

$$\xi = \begin{cases} 1 & \text{for } x \leq L/2, \\ 0 & \text{for } x > L/2, \end{cases}$$

```

>>> x = mesh.cellCenters[0]
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L / 2)

```

and with uniform concentration fields, initially equal to the average of the solidus and liquidus concentrations

```

>>> for Cj in interstitials + substitutionals:
...     Cj.setValue((Cj.S + Cj.L) / 2.)

```

If we're running interactively, we create a viewer

```

>>> if __name__ == '__main__':
...     viewer = Viewer(vars=( [phase]
...                           + interstitials + substitutionals
...                           + [solvent]),
...                    datamin=0, datamax=1)
...     viewer.plot()

```

and again iterate to equilibrium

```

>>> solver = DefaultAsymmetricSolver(tolerance=1e-10)

```

```

>>> dt = 10000
>>> from builtins import range
>>> for i in range(5):
...     for field in [phase] + substitutionals + interstitials:
...         field.updateOld()
...     phase.equation.solve(var = phase, dt = dt)
...     for field in substitutionals + interstitials:
...         field.equation.solve(var = field,
...                               dt = dt,

```

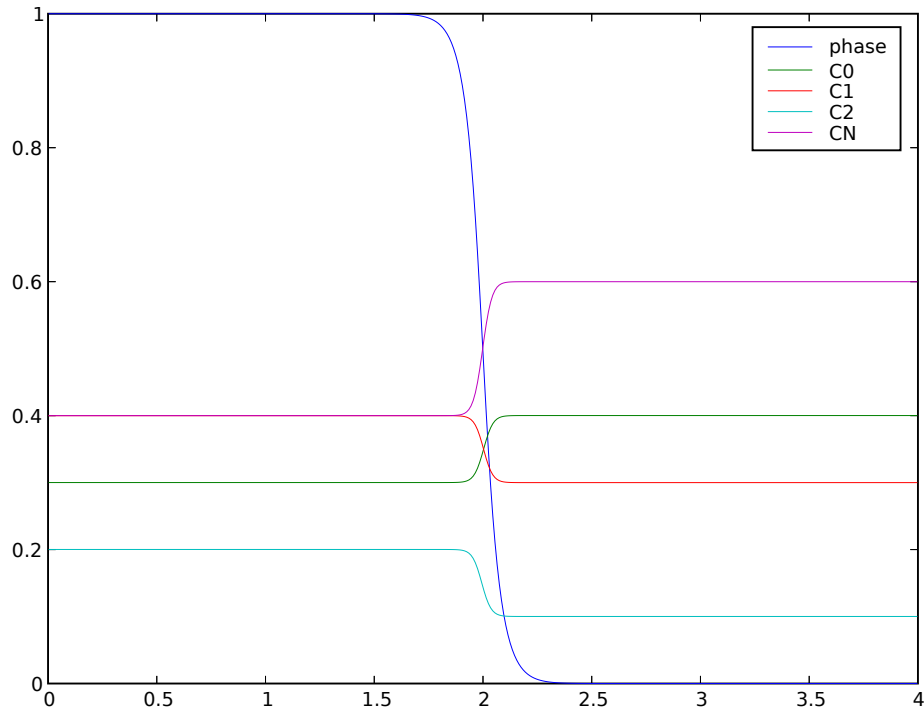
(continues on next page)

(continued from previous page)

```

...         solver = solver)
...     if __name__ == '__main__':
...         viewer.plot()

```



We can confirm that the far-field phases have remained separated

```

>>> X = mesh.faceCenters[0]
>>> print( numerix.allclose(phase.faceValue[X.value==0], 1.0, rtol = 1e-5, atol = 1e-5))
True
>>> print( numerix.allclose(phase.faceValue[X.value==L], 0.0, rtol = 1e-5, atol = 1e-5))
True

```

and that the concentration fields have appropriately segregated into their equilibrium values in each phase

```

>>> equilibrium = True
>>> for Cj in interstitials + substitutionals:
...     equilibrium &= numerix.allclose(Cj.faceValue[X.value==0], Cj.S, rtol = 3e-3,
... atol = 3e-3).value
...     equilibrium &= numerix.allclose(Cj.faceValue[X.value==L], Cj.L, rtol = 3e-3,
... atol = 3e-3).value
>>> print(equilibrium)
True

```

### 23.11.10 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,

$$\frac{1}{M_\phi} \frac{\partial \phi}{\partial t} = \kappa_\phi \nabla^2 \phi - \frac{\partial f}{\partial \phi} \quad (23.11)$$

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)
```

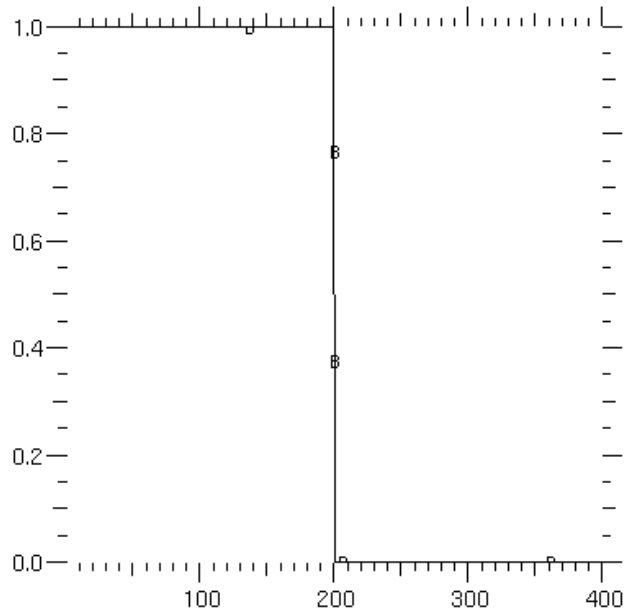
and set a step-function initial condition

$$\phi = \begin{cases} 1 & \text{for } x \leq L/2 \\ 0 & \text{for } x > L/2 \end{cases} \quad \text{at } t = 0$$

```
>>> 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

$$\phi = \frac{1}{2} \left[ 1 - \tanh \frac{x - L/2}{2\sqrt{\kappa/W}} \right] \quad (23.12)$$

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

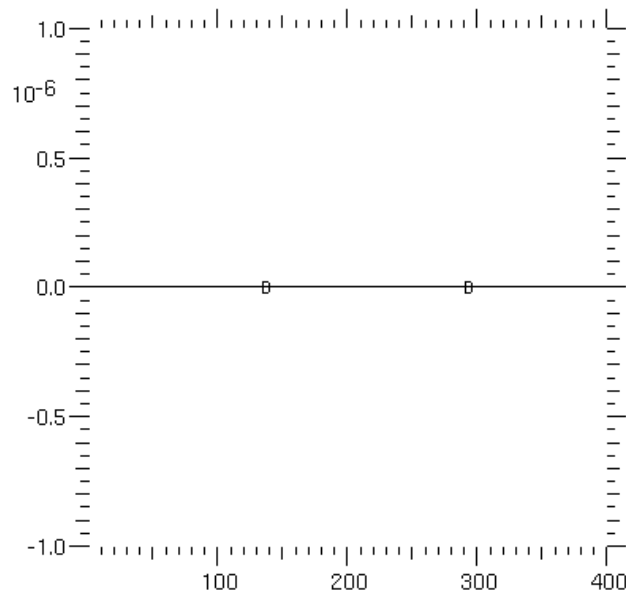
```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> S0 = mPhi * phase * (1 - phase)
>>> eq = S0 + DiffusionTerm(coeff=kappa)
```

After solving this equation

```
>>> eq.solve(var = phase, solver=DummySolver())
```

we obtain the surprising result that  $\phi$  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 (23.11)

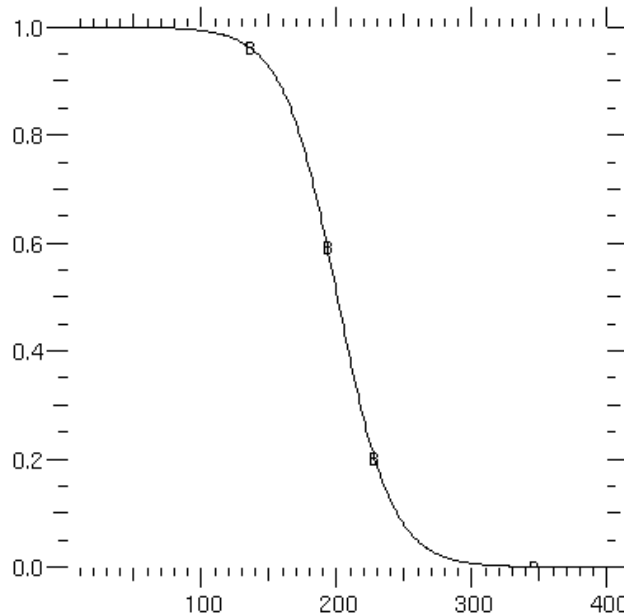
```
>>> eq = TransientTerm() == DiffusionTerm(coeff=kappa) + S0
```

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

```
>>> from builtins import range
>>> for i in range(13):
...     eq.solve(var = phase, dt=1.)
...     if __name__ == '__main__':
...         viewer.plot()
```

After 13 time steps, the solution has converged to the analytical solution

```
>>> print(phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4))
1
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Relaxation, explicit. Press <return> to proceed...")
```



**Note:** The solution is only found accurate to  $\approx 4.3 \times 10^{-5}$  because the infinite-domain analytical solution (23.12) 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  $\phi$ . 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  $\phi^4$  in the linear set of equations  $M\vec{\phi} - \vec{b} = 0$ .

By linearizing a source as  $S = S_0 - S_1\phi$ , we make it more implicit by adding the coefficient  $S_1$  to the matrix diagonal. For numerical stability, this linear coefficient must never be negative.

There are an infinite number of choices for this linearization, but many do not converge very well. One choice is that used by Ryo Kobayashi:

```
>>> S0 = mPhi * phase * (mPhi > 0)
>>> S1 = mPhi * ((mPhi < 0) - phase)
>>> eq = DiffusionTerm(coeff=kappa) + S0 \
...     + ImplicitSourceTerm(coeff = S1)
```

**Note:** Because `mPhi` is a variable field, the quantities `(mPhi > 0)` and `(mPhi < 0)` evaluate to variable *fields* of *True* and *False*, instead of single Boolean values.

This expression converges to the same value given by the explicit relaxation approach, but in only 8 sweeps (note that because there is no transient term, these sweeps are not time steps, but rather repeated iterations at the same time step to reach a converged solution).

**Note:** We use `solve()` instead of `sweep()` because we don't care about the residual. Either function would work, but `solve()` is a bit faster.

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

```
>>> from builtins import range
>>> for i in range(8):
...     eq.solve(var = phase)
>>> print(phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4))
1
>>> from fipy import input
>>> if __name__ == '__main__':
...     viewer.plot()
...     input("Kobayashi, semi-implicit. Press <return> to proceed...")
```

In general, the best convergence is obtained when the linearization gives a good representation of the relationship between the source and the dependent variable. The best practical advice is to perform a Taylor expansion of the source about the previous value of the dependent variable such that  $S = S_{\text{old}} + \left. \frac{\partial S}{\partial \phi} \right|_{\text{old}} (\phi - \phi_{\text{old}}) = (S - \left. \frac{\partial S}{\partial \phi} \phi)_{\text{old}} + \left. \frac{\partial S}{\partial \phi} \right|_{\text{old}} \phi$ .

Now, if our source term is represented by  $S = S_0 + S_1 \phi$ , then  $S_1 = \left. \frac{\partial S}{\partial \phi} \right|_{\text{old}}$  and  $S_0 = (S - \left. \frac{\partial S}{\partial \phi} \phi)_{\text{old}} = S_{\text{old}} - S_1 \phi_{\text{old}}$ . In this way, the linearized source will be tangent to the curve of the actual source as a function of the dependent variable.

For our source,  $S = m_\phi \phi(1 - \phi)$ ,

$$\frac{\partial S}{\partial \phi} = \frac{\partial m_\phi}{\partial \phi} \phi(1 - \phi) + m_\phi(1 - 2\phi)$$

and

$$\frac{\partial m_\phi}{\partial \phi} = 2W - 30(1 - 2\phi)L \frac{T - T_M}{T_M},$$

or

```
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
```

(continues on next page)

(continued from previous page)

```
>>> eq = DiffusionTerm(coeff=kappa) + S0 \
...   + ImplicitSourceTerm(coeff = S1)
```

Using this scheme, where the coefficient of the implicit source term is tangent to the source, we reach convergence in only 5 sweeps

```
>>> phase.setValue(1.)
>>> phase.setValue(0., where=x > L/2)
```

```
>>> from builtins import range
>>> for i in range(5):
...     eq.solve(var = phase)
>>> print(phase.allclose(analyticalArray, rtol = 1e-4, atol = 1e-4))
1
>>> from fipy import input
>>> 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  $\phi$  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  $\sigma$  and the interfacial thickness  $\delta$  by

$$\begin{aligned}\kappa &= 6\sigma\delta \\ W &= \frac{6\sigma}{\delta} \\ M_\phi &= \frac{T_m\beta}{6L\delta}.\end{aligned}$$

We take  $\delta \approx \Delta x$ .

```
>>> delta = 1.5 * dx
>>> sigma = 3.7e-5 # J / cm**2
>>> beta = 0.33 # cm / (K s)
>>> kappa = 6 * sigma * delta # J / cm
>>> W = 6 * sigma / delta # J / cm**3
>>> Mphi = Tm * beta / (6. * Lv * delta) # cm**3 / (J s)
```

```
>>> if __name__ == '__main__':
...     displacement = L * 0.1
... else:
...     displacement = L * 0.025
```

```
>>> analyticalArray = CellVariable(name="tanh", mesh=mesh,
...                               value=0.5 * (1 - numerix.tanh((x - (L / 2. +
↳displacement))
...                               / (2 * delta))))
```

and make a new viewer

```
>>> if __name__ == '__main__':
...     viewer2 = Viewer(vars = (phase, analyticalArray))
...     viewer2.plot()
```

Now we can redefine the transient phase field equation, using the optimal form of the source term shown above

```
>>> mPhi = -((1 - 2 * phase) * W + 30 * phase * (1 - phase) * enthalpy)
>>> dmPhidPhi = 2 * W - 30 * (1 - 2 * phase) * enthalpy
>>> S1 = dmPhidPhi * phase * (1 - phase) + mPhi * (1 - 2 * phase)
>>> S0 = mPhi * phase * (1 - phase) - S1 * phase
>>> eq = TransientTerm(coeff=1/Mphi) == DiffusionTerm(coeff=kappa) \
...     + S0 + ImplicitSourceTerm(coeff = S1)
```

In order to separate the effect of forming the phase field interface from the kinetics of moving it, we first equilibrate at the melting point. We now use the `sweep()` method instead of `solve()` because we require the residual.

```
>>> timeStep = 1e-6
>>> from builtins import range
>>> for i in range(10):
...     phase.updateOld()
...     res = 1e+10
...     while res > 1e-5:
...         res = eq.sweep(var=phase, dt=timeStep)
>>> if __name__ == '__main__':
...     viewer2.plot()
```

and then quench by 1 K

```
>>> T.setValue(T() - 1)
```

In order to have a stable numerical solution, the interface must not move more than one grid point per time step, we thus set the timestep according to the grid spacing  $\Delta x$ , the linear kinetic coefficient  $\beta$ , and the undercooling  $|T_m - T|$ . Again we use the `sweep()` method as a replacement for `solve()`.

```
>>> velocity = beta * abs(Tm - T()) # cm / s
>>> timeStep = .1 * dx / velocity # s
>>> elapsed = 0
>>> while elapsed < displacement / velocity:
...     phase.updateOld()
...     res = 1e+10
...     while res > 1e-5:
...         res = eq.sweep(var=phase, dt=timeStep)
...     elapsed += timeStep
...     if __name__ == '__main__':
...         viewer2.plot()
```

A hyperbolic tangent is not an exact steady-state solution given the quintic polynomial we chose for the  $p$  function, but it gives a reasonable approximation.

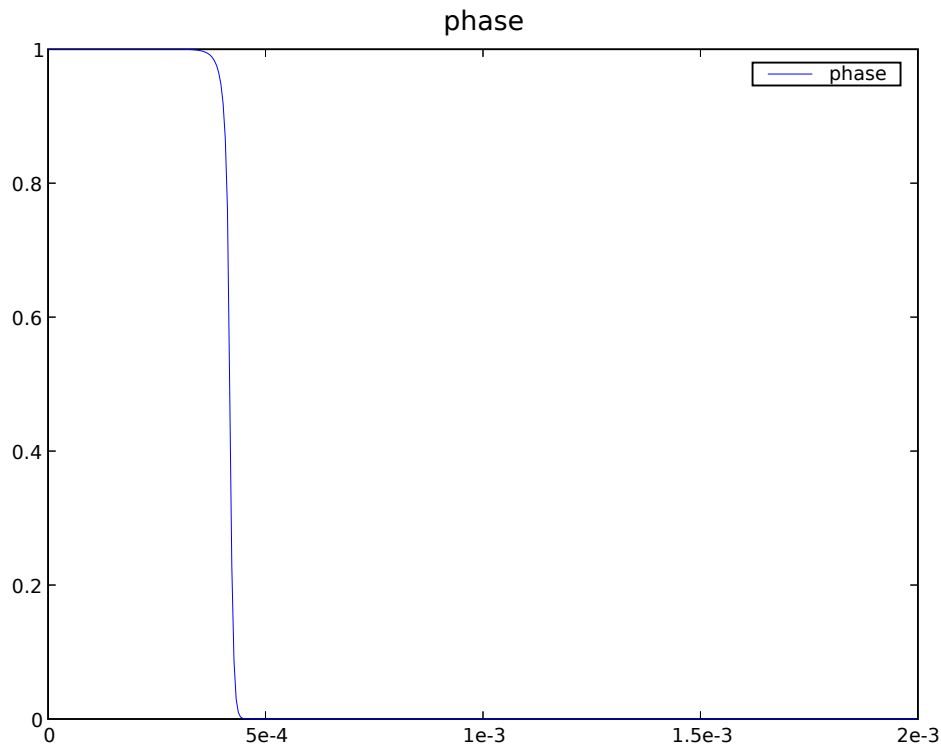
```
>>> print(phase.allclose(analyticalArray, rtol = 5, atol = 2e-3))
1
```

If we had made another common choice of  $p(\phi) = \phi^2(3 - 2\phi)$ , we would have found much better agreement, as that case does give an exact tanh solution in steady state. If SciPy is available, another way to compare against the expected result is to do a least-squared fit to determine the interface velocity and thickness

```
>>> try:
...     def tanhResiduals(p, y, x, t):
...         V, d = p
...         return y - 0.5 * (1 - numerix.tanh((x - V * t - L / 2.) / (2*d)))
...     from scipy.optimize import leastsq
...     x = mesh.cellCenters[0]
...     (V_fit, d_fit), msg = leastsq(tanhResiduals, [L/2., delta],
...                                   args=(phase.globalValue, x.globalValue, elapsed))
... except ImportError:
...     V_fit = d_fit = 0
...     print("The SciPy library is unavailable to fit the interface \
... thickness and velocity")
```

```
>>> print(abs(1 - V_fit / velocity) < 4.1e-2)
True
>>> print(abs(1 - d_fit / delta) < 2e-2)
True
```

```
>>> from fipy import input
>>> if __name__ == '__main__':
...     input("Dimensional, semi-implicit. Press <return> to proceed...")
```



### 23.11.11 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')
>>> from builtins import range
>>> 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
```



## 23.11.12 examples.phase.test

## 23.12 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>

### 23.12.1 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,

$$f = -\frac{e\rho^2}{m^2} + \frac{RT}{m} \left( \ln \frac{\rho}{m - \bar{v}\rho} \right) \quad (23.13)$$

where  $\rho$  is the density. This free energy supports a two phase equilibrium with densities given by  $\rho^l$  and  $\rho^v$  in the liquid and vapor phases, respectively. The densities are determined by solving the following system of equations,

$$P(\rho^l) = P(\rho^v) \quad (23.14)$$

and

$$\mu(\rho^l) = \mu(\rho^v) \quad (23.15)$$

where  $\mu$  is the chemical potential,

$$\mu = \frac{\partial f}{\partial \rho} \quad (23.16)$$

and  $P$  is the pressure,

$$P = \rho\mu - f \quad (23.17)$$

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. (23.14) and (23.15). Firstly, Eqs. (23.13), (23.16) and (23.17) 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,

$$\frac{\partial \rho}{\partial t} + \partial_j (\rho u_j) = 0 \quad (23.18)$$

and

$$\frac{\partial (\rho u_i)}{\partial t} + \partial_j (\rho u_i u_j) = \partial_j (\nu [\partial_j u_i + \partial_i u_j]) - \rho \partial_i \mu^{NC} \quad (23.19)$$

where the non-classical potential,  $\mu^{NC}$ , is given by,

$$\mu^{NC} = \frac{\delta F}{\delta \rho} = \mu - \epsilon T \partial_j^2 \rho \quad (23.20)$$

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  $\mu^{NC}$  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,

$$u_{i,f}^c = \frac{A_f d_f}{\bar{d}_f} \left( \overline{\rho \partial_i \mu^{NC}}_f - \bar{\rho}_f \partial_{i,f} \mu^{NC} \right) \quad (23.21)$$

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. (23.18) such that,

$$\frac{\partial \rho}{\partial t} + \partial_j (\rho [u_j + u_i^c]) = 0 \quad (23.22)$$

Equation (23.22) 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. (23.21) is calculated in an explicit manner in the VanLeerConvectionTerm and the second term is calculated implicitly as a DiffusionTerm with  $\mu^{NC}$  as the independent variable.

In order to write Eq. (23.19) 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  $\mu^{NC}$  variable will be solved implicitly. To do this the Eq. (23.20) is linearized in  $\rho$  such that

$$\mu^{NC} = \mu^* + \left( \frac{\partial \mu}{\partial \rho} \right)^* (\rho - \rho^*) - \epsilon T \partial_j^2 \rho \quad (23.23)$$

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  $\mu^*$  is simply,

```
>>> potential = mu(density)
```

Eq. (23.23) 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  $\mu^{NC}$  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*.

```

>>> while timestep < totalSteps:
...
...     sweep = 0
...     dt *= 1.1
...     residual = 1.
...     initialResidual = None
...
...     density.updateOld()
...     velocity.updateOld()
...     matrixDiagonal.updateOld()
...
...     while residual > tolerance:
...
...         densityPrevious[:] = density
...         velocityPrevious[:] = velocity
...         previousResidual = residual
...
...         dt = min(dt, dx / max(abs(velocity)) * cfl)
...
...         coupledEqn.cacheMatrix()
...         residual = coupledEqn.sweep(dt=dt)
...
...         if initialResidual is None:
...             initialResidual = residual
...
...         residual = residual / initialResidual
...
...         if residual > previousResidual * 1.1 or sweep > 20:
...             density[:] = density.old
...             velocity[:] = velocity.old

```

(continues on next page)

(continued from previous page)

```

...         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

```

### 23.12.2 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 /

```

(continues on next page)

(continued from previous page)

```

↳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) \
...     + 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) \

```

(continues on next page)

(continued from previous page)

```

... == 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 & potentialNCEqn

```

```

>>> 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

```

(continues on next page)



(continued from previous page)

```

>>> dt = 1e-14
>>> timestep = 0
>>> relaxation = 0.5
>>> sweeps = 0
>>> if __name__ == '__main__':
...     totalSteps = 1e+10
...     totalSweeps = 1e+10
... else:
...     totalSteps = 1
...     totalSweeps = 1

```

```

>>> while timestep < totalSteps:
...
...     sweep = 0
...     dt *= 1.1
...     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)
...
...         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

```

(continues on next page)

(continued from previous page)

```

...     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:
...             viewer.plot()
...
...     timestep += 1

```

```

>>> from fipy import input
>>> if __name__ == '__main__':
...     input('finished')

```

### 23.12.3 examples.reactiveWetting.test

Run all the test cases in *examples/reactiveWetting/*

## 23.13 examples.riemann

### Modules

<i>examples.riemann.acoustics</i>	Test
<i>examples.riemann.test</i>	

### 23.13.1 examples.riemann.acoustics

Test

```

>>> print((0.4 < max(q.globalValue[0]) < 0.5))
True

```

## 23.13.2 examples.riemann.test

## 23.14 examples.test

Run all the test cases in examples/

## 23.15 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.

### 23.15.1 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(
...     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):
...

```

(continues on next page)

(continued from previous page)

```
NameError: name 'it' is not defined
```

and test the solution against the analytical result

$$\phi = \frac{1 - \exp(-u_x x/D)}{1 - \exp(-u_x L/D)}$$

or

```
>>> axis = 0
>>> x = mesh.getCellCenters()[:, axis]
>>> from fipy.tools import numerix
>>> CC = 1. - numerix.exp(-convCoeff[axis] * x / diffCoeff)
>>> DD = 1. - numerix.exp(-convCoeff[axis] * L / diffCoeff)
>>> analyticalArray = CC / DD
>>> numerix.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
0
```

If the problem is run interactively, we can view the result:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.grid2DGistViewer import Grid2DGistViewer
Traceback (most recent call last):
...
ImportError: No module named grid2DGistViewer
```

```
...     viewer = Grid2DGistViewer(var)
...     viewer.plot()
```

We see that a number of errors are thrown:

- ImportError: No module named equations.stdyConvDiffScEquation
- NameError: name 'SteadyConvectionDiffusionScEquation' is not defined
- NameError: name 'eq' is not defined
- NameError: name 'it' is not defined
- ImportError: No module named grid2DGistViewer

As is usually the case with computer programming, many of these errors are caused by earlier errors. Let us update the script, section by section:

Although no error was generated by the use of Grid2D, *FiPy* 1.0 supports a true 1D mesh class, so we instantiate the mesh as

```
>>> L = 10.
>>> nx = 1000
>>> from fipy.meshes.grid1D import Grid1D
>>> mesh = Grid1D(dx = L / nx, nx = nx)
```

The Grid2D class with  $n_y = 1$  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 *GridID* mesh does not even have top and bottom faces:

```
>>> valueLeft = 0.
>>> valueRight = 1.
>>> from fipy.boundaryConditions.fixedValue import FixedValue
>>> boundaryConditions = (
...     FixedValue(mesh.getFacesLeft(), valueLeft),
...     FixedValue(mesh.getFacesRight(), valueRight))
```

The creation of the solution variable is unchanged:

```
>>> from fipy.variables.cellVariable import CellVariable
>>> var = CellVariable(name = "concentration",
...                   mesh = mesh,
...                   value = valueLeft)
```

The biggest change between *FiPy* 0.1 and *FiPy* 1.0 is that Equation objects no longer exist at all. Instead, *Term* objects can be simply added, subtracted, and equated to assemble an equation. Where before the assembly of the equation occurred in the black-box of *SteadyConvectionDiffusionScEquation*, we now assemble it directly:

```
>>> from fipy.terms.implicitDiffusionTerm import ImplicitDiffusionTerm
>>> diffTerm = ImplicitDiffusionTerm(coeff = diffCoeff)
```

```
>>> from fipy.terms.exponentialConvectionTerm import ExponentialConvectionTerm
>>> eq = diffTerm + ExponentialConvectionTerm(coeff = convCoeff,
...                                           diffusionTerm = diffTerm)
... 
```

One thing that *SteadyConvectionDiffusionScEquation* took care of automatically was that a *ConvectionTerm* must know about any *DiffusionTerm* in the equation in order to calculate a Péclet number. Now, the *DiffusionTerm* must be explicitly passed to the *ConvectionTerm* in the *diffusionTerm* parameter.

The *Iterator* class still exists, but it is no longer necessary. Instead, the solution to an implicit steady-state problem like this can simply be obtained by telling the equation to solve itself (with an appropriate *solver* if desired, although the default *LinearPCGSolver* is usually suitable):

```
>>> from fipy.solvers import *
>>> eq.solve(var = var,
...         solver = LinearLUSolver(tolerance = 1.e-15, steps = 2000),
...         boundaryConditions = boundaryConditions)
```

---

**Note:** In version 0.1, the Equation object had to be told about the *Variable*, *Solver*, and *BoundaryCondition* objects when it was created (and it, in turn, passed much of this information to the *Term* objects in order to create them). In version 1.0, the *Term* objects (and the equation assembled from them) are abstract. The *Variable*, *Solver*, and *BoundaryCondition* objects are only needed by the *solve()* method (and, in fact, the same equation could be used to solve different variables, with different solvers, subject to different boundary conditions, if desired).

---

The analytical solution is unchanged, and we can test as before

```
>>> numerix.allclose(analyticalArray, var, rtol = 1e-10, atol = 1e-10)
1
```

or we can use the slightly simpler syntax

```
>>> print(var.allclose(analyticalArray, rtol = 1e-10, atol = 1e-10))
1
```

The `ImportError: No module named grid2DGistViewer` results because the `Viewer` classes have been moved and renamed. This error could be resolved by changing the `import` statement appropriately:

```
>>> if __name__ == '__main__':
...     from fipy.viewers.gistViewer.gist1DViewer import Gist1DViewer
...     viewer = Gist1DViewer(vars = var)
...     viewer.plot()
```

Instead, rather than instantiating a particular `Viewer` (which you can still do, if you desire), a generic “factory” method will return a `Viewer` appropriate for the supplied `Variable` object(s):

```
>>> if __name__ == '__main__':
...     import fipy.viewers
...     viewer = fipy.viewers.make(vars = var)
...     viewer.plot()
```

Please do not hesitate to contact us if this example does not help you convert your existing scripts to *FiPy* 1.0.

## 23.15.2 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 superseded *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.linearPCGSolver import LinearPCGSolver
```

to either

```
>>> from fipy import LinearPCGSolver
```

or

```
>>> from fipy.solvers.pysparse.linearPCGSolver import LinearPCGSolver
```

- The `numerix.max()` and `numerix.min()` functions no longer exist. Either call `max()` and `min()` or the `max()` and `min()` methods of a *Variable*.
- The `Numeric` module has not been supported for a long time. Be sure to use

```
>>> from fipy import numerix
```

instead of

```
>>> import Numeric
```

The remaining changes are not *required*, but they make scripts easier to read and we recommend them. *FiPy* may issue a `DeprecationWarning` for some cases, to indicate that we may not maintain the old syntax indefinitely.

- All of the most commonly used classes and functions in *FiPy* are directly accessible in the *fipy* namespace. For brevity, our examples now start with

```
>>> from fipy import *
```

instead of the explicit

```
>>> from fipy.meshes.grid1D import Grid1D
>>> from fipy.terms.powerLawConvectionTerm import PowerLawConvectionTerm
>>> from fipy.variables.cellVariable import CellVariable
```

imports that we used to use. Most of the explicit imports should continue to work, so you do not need to change them if you don't wish to, but we find our own scripts much easier to read without them.

All of the *numerix* module is now imported into the *fipy* namespace, so you can call *numerix* functions a number of different ways, including:

```
>>> from fipy import *
>>> y = exp(x)
```

or

```
>>> from fipy import numerix
>>> y = numerix.exp(x)
```

or

```
>>> from fipy.tools.numerix import exp
>>> y = exp(x)
```

We generally use the first, but you may see us use the others, and should feel free to use whichever form you find most comfortable.

---

**Note:** Internally, *FiPy* uses explicit imports, as is considered *best Python practice*, but we feel that clarity trumps orthodoxy when it comes to the examples.

---

- The function `fipy.viewers.make()` has been renamed to `fipy.viewers.Viewer()`. All of the `limits` can now be supplied as direct arguments, as well (although this is not required). The result is a more natural syntax:

```
>>> from fipy import Viewer
>>> viewer = Viewer(vars=(alpha, beta, gamma), datamin=0, datamax=1)
```

instead of

```
>>> from fipy import viewers
>>> viewer = viewers.make(vars=(alpha, beta, gamma),
...                       limits={'datamin': 0, 'datamax': 1})
```

With the old syntax, there was also a temptation to write

```
>>> from fipy.viewers import make
>>> viewer = make(vars=(alpha, beta, gamma))
```

which can be very hard to understand after the fact (make? make what?).

- A `ConvectionTerm` can now calculate its Péclet number automatically, so the `diffusionTerm` argument is no longer required

```
>>> eq = (TransientTerm()
...       == DiffusionTerm(coeff=diffCoeff)
...       + PowerLawConvectionTerm(coeff=convCoeff))
```

instead of

```
>>> diffTerm = DiffusionTerm(coeff=diffCoeff)
>>> eq = (TransientTerm()
...       == diffTerm
...       + PowerLawConvectionTerm(coeff=convCoeff, diffusionTerm=diffTerm))
```

- An `ImplicitSourceTerm` now “knows” how to partition itself onto the solution matrix, so you can write

```
>>> S0 = mXi * phase * (1 - phase) - phase * S1
>>> source = S0 + ImplicitSourceTerm(coeff=S1)
```

instead of

```
>>> S0 = mXi * phase * (1 - phase) - phase * S1 * (S1 < 0)
>>> source = S0 + ImplicitSourceTerm(coeff=S1 * (S1 < 0))
```

It is definitely still advantageous to hand-linearize your source terms, but it is no longer necessary to worry about putting the “wrong” sign on the diagonal of the matrix.

- To make clearer the distinction between iterations, timesteps, and sweeps (see FAQ *Iterations, timesteps, and sweeps? Oh, my!*) the `steps` argument to a `Solver` object has been renamed `iterations`.
- `ImplicitDiffusionTerm` has been renamed to `DiffusionTerm`.

### 23.15.3 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.
- [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.
- [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.
- [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.
- [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.
- [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.
- [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.
- [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, 919

examples.benchmarking, 920

examples.benchmarking.benchmarker, 920

examples.benchmarking.size, 920

examples.benchmarking.steps, 920

examples.benchmarking.utils, 920

examples.benchmarking.versions, 920

examples.cahnHilliard, 920

examples.cahnHilliard.mesh2D, 921

examples.cahnHilliard.mesh2DCoupled, 923

examples.cahnHilliard.mesh3D, 926

examples.cahnHilliard.sphere, 928

examples.cahnHilliard.sphereDaemon, 930

examples.cahnHilliard.tanh1D, 930

examples.cahnHilliard.test, 932

examples.chemotaxis, 932

examples.chemotaxis.input, 933

examples.chemotaxis.input2D, 935

examples.chemotaxis.parameters, 938

examples.chemotaxis.test, 938

examples.convection, 938

examples.convection.advection, 938

examples.convection.advection.explicitUpwind, 939

examples.convection.advection.implicitUpwind, 939

examples.convection.advection.vanLeerUpwind, 939

examples.convection.exponential1D, 939

examples.convection.exponential1D.cylindricalMesh1D, 940

examples.convection.exponential1D.cylindricalMesh2DNonUniform, 941

examples.convection.exponential1D.mesh1D, 943

examples.convection.exponential1D.tri2D, 944

examples.convection.exponential1DBack, 945

examples.convection.exponential1DBack.mesh1D, 945

examples.convection.exponential1DSource, 946

examples.convection.exponential1DSource.mesh1D, 947

examples.convection.exponential1DSource.tri2D, 948

examples.convection.exponential2D, 949

examples.convection.exponential2D.cylindricalMesh2D, 949

examples.convection.exponential2D.cylindricalMesh2DNonUniform, 951

examples.convection.exponential2D.mesh2D, 952

examples.convection.exponential2D.tri2D, 953

examples.convection.peclet, 954

examples.convection.powerLaw1D, 955

examples.convection.powerLaw1D.mesh1D, 956

examples.convection.powerLaw1D.tri2D, 957

examples.convection.robin, 958

examples.convection.source, 960

examples.convection.test, 961

examples.diffusion, 961

examples.diffusion.anisotropy, 961

examples.diffusion.circle, 963

examples.diffusion.circleQuad, 968

examples.diffusion.coupled, 973

examples.diffusion.electrostatics, 975

examples.diffusion.explicit, 979

examples.diffusion.explicit.mesh1D, 980

examples.diffusion.explicit.mixedelement, 981

examples.diffusion.explicit.test, 982

examples.diffusion.explicit.tri2D, 982

examples.diffusion.mesh1D, 983

examples.diffusion.mesh20x20, 1003

examples.diffusion.mesh20x20Coupled, 1006

examples.diffusion.nthOrder, 1009

examples.diffusion.nthOrder.input4thOrder1D, 1010

examples.diffusion.nthOrder.input4thOrder\_line, 1012

examples.diffusion.nthOrder.test, 1012

examples.diffusion.steadyState, 1012

examples.diffusion.steadyState.mesh1D, 1012

examples.diffusion.steadyState.mesh1D.inputPeriodic, 1013

[examples.diffusion.steadyState.mesh1D.tri2Dinput](#), 1013  
[examples.diffusion.steadyState.mesh20x20](#), 1014  
[examples.diffusion.steadyState.mesh20x20.gmshinput](#), 1014  
[examples.diffusion.steadyState.mesh20x20.isotropic](#), 1014  
[examples.diffusion.steadyState.mesh20x20.modifiedMeshInput](#), 1015  
[examples.diffusion.steadyState.mesh20x20.orthorhombic](#), 1016  
[examples.diffusion.steadyState.mesh20x20.tri2Dinput](#), 1017  
[examples.diffusion.steadyState.mesh50x50](#), 1017  
[examples.diffusion.steadyState.mesh50x50.input](#), 1017  
[examples.diffusion.steadyState.mesh50x50.tri2Dinput](#), 1017  
[examples.diffusion.steadyState.otherMeshes](#), 1018  
[examples.diffusion.steadyState.otherMeshes.cubicalProblem](#), 1018  
[examples.diffusion.steadyState.otherMeshes.grid3Dinput](#), 1018  
[examples.diffusion.steadyState.otherMeshes.prism](#), 1018  
[examples.diffusion.steadyState.test](#), 1019  
[examples.diffusion.test](#), 1019  
[examples.diffusion.variable](#), 1019  
[examples.elphf](#), 1020  
[examples.elphf.diffusion](#), 1021  
[examples.elphf.diffusion.mesh1D](#), 1021  
[examples.elphf.diffusion.mesh1Ddimensional](#), 1024  
[examples.elphf.diffusion.mesh2D](#), 1026  
[examples.elphf.input](#), 1029  
[examples.elphf.phase](#), 1035  
[examples.elphf.phaseDiffusion](#), 1038  
[examples.elphf.poisson](#), 1046  
[examples.elphf.test](#), 1049  
[examples.flow](#), 1049  
[examples.flow.stokesCavity](#), 1050  
[examples.flow.test](#), 1054  
[examples.levelSet](#), 1054  
[examples.levelSet.advection](#), 1055  
[examples.levelSet.advection.circle](#), 1055  
[examples.levelSet.advection.mesh1D](#), 1057  
[examples.levelSet.advection.test](#), 1058  
[examples.levelSet.advection.trench](#), 1058  
[examples.levelSet.distanceFunction](#), 1060  
[examples.levelSet.distanceFunction.circle](#), 1060  
[examples.levelSet.distanceFunction.interior](#), 1061  
[examples.levelSet.distanceFunction.mesh1D](#), 1062  
[examples.levelSet.distanceFunction.square](#), 1063  
[examples.levelSet.distanceFunction.test](#), 1064  
[examples.levelSet.electroChem](#), 1064  
[examples.levelSet.electroChem.adsorbingSurfactantEquation](#), 1064  
[examples.levelSet.electroChem.adsorption](#), 1064  
[examples.levelSet.electroChem.gapFillDistanceVariable](#), 1065  
[examples.levelSet.electroChem.gapFillMesh](#), 1065  
[examples.levelSet.electroChem.gold](#), 1066  
[examples.levelSet.electroChem.howToWriteAScript](#), 1067  
[examples.levelSet.electroChem.leveler](#), 1073  
[examples.levelSet.electroChem.lines](#), 1077  
[examples.levelSet.electroChem.matplotlibSurfactantViewer](#), 1077  
[examples.levelSet.electroChem.mayaviSurfactantViewer](#), 1081  
[examples.levelSet.electroChem.metalIonDiffusionEquation](#), 1084  
[examples.levelSet.electroChem.simpleTrenchSystem](#), 1084  
[examples.levelSet.electroChem.surfactantBulkDiffusionEquation](#), 1087  
[examples.levelSet.electroChem.test](#), 1087  
[examples.levelSet.electroChem.trenchMesh](#), 1087  
[examples.levelSet.surfactant](#), 1087  
[examples.levelSet.surfactant.circle](#), 1088  
[examples.levelSet.surfactant.expandingCircle](#), 1088  
[examples.levelSet.surfactant.square](#), 1089  
[examples.levelSet.surfactant.test](#), 1090  
[examples.levelSet.test](#), 1090  
[examples.meshing](#), 1090  
[examples.meshing.gmshRefinement](#), 1090  
[examples.meshing.inputGrid2D](#), 1090  
[examples.meshing.sphere](#), 1091  
[examples.meshing.test](#), 1092  
[examples.parallel](#), 1092  
[examples.phase](#), 1092  
[examples.phase.anisotropy](#), 1093  
[examples.phase.anisotropyOLD](#), 1097  
[examples.phase.binary](#), 1100  
[examples.phase.binaryCoupled](#), 1109  
[examples.phase.impingement](#), 1119  
[examples.phase.impingement.mesh20x20](#), 1119



- examples.phase.impingement.mesh40x1, 1123
  - examples.phase.impingement.test, 1126
  - examples.phase.missOrientation, 1126
  - examples.phase.missOrientation.circle, 1126
  - examples.phase.missOrientation.mesh1D, 1127
  - examples.phase.missOrientation.modCircle, 1128
  - examples.phase.missOrientation.test, 1128
  - examples.phase.polyxtal, 1128
  - examples.phase.polyxtalCoupled, 1134
  - examples.phase.quaternary, 1140
  - examples.phase.simple, 1147
  - examples.phase.symmetry, 1155
  - examples.phase.test, 1157
  - examples.reactiveWetting, 1157
  - examples.reactiveWetting.liquidVapor1D, 1157
  - examples.reactiveWetting.liquidVapor2D, 1162
  - examples.reactiveWetting.test, 1166
  - examples.riemann, 1166
  - examples.riemann.acoustics, 1166
  - examples.riemann.test, 1167
  - examples.test, 1167
  - examples.updating, 1167
  - examples.updating.update0\_1to1\_0, 1167
  - examples.updating.update1\_0to2\_0, 1171
  - examples.updating.update2\_0to3\_0, 1175
- f**
- fiPy, 125
  - fiPy.boundaryConditions, 126
  - fiPy.boundaryConditions.boundaryCondition, 127
  - fiPy.boundaryConditions.constraint, 127
  - fiPy.boundaryConditions.fixedFlux, 128
  - fiPy.boundaryConditions.fixedValue, 128
  - fiPy.boundaryConditions.nthOrderBoundaryCondition, 129
  - fiPy.boundaryConditions.test, 129
  - fiPy.matrices, 130
  - fiPy.matrices.offsetSparseMatrix, 130
  - fiPy.matrices.petscMatrix, 130
  - fiPy.matrices.pysparseMatrix, 130
  - fiPy.matrices.scipyMatrix, 130
  - fiPy.matrices.sparseMatrix, 130
  - fiPy.matrices.test, 130
  - fiPy.matrices.trilinosMatrix, 130
  - fiPy.meshes, 130
  - fiPy.meshes.abstractMesh, 132
  - fiPy.meshes.builders, 139
  - fiPy.meshes.builders.abstractGridBuilder, 140
  - fiPy.meshes.builders.grid1DBuilder, 140
  - fiPy.meshes.builders.grid2DBuilder, 140
  - fiPy.meshes.builders.grid3DBuilder, 140
  - fiPy.meshes.builders.periodicGrid1DBuilder, 140
  - fiPy.meshes.builders.utilityClasses, 140
  - fiPy.meshes.cylindricalGrid1D, 140
  - fiPy.meshes.cylindricalGrid2D, 140
  - fiPy.meshes.cylindricalNonUniformGrid1D, 140
  - fiPy.meshes.cylindricalNonUniformGrid2D, 149
  - fiPy.meshes.cylindricalUniformGrid1D, 158
  - fiPy.meshes.cylindricalUniformGrid2D, 165
  - fiPy.meshes.factoryMeshes, 172
  - fiPy.meshes.gmshMesh, 175
  - fiPy.meshes.grid1D, 223
  - fiPy.meshes.grid2D, 223
  - fiPy.meshes.grid3D, 223
  - fiPy.meshes.mesh, 223
  - fiPy.meshes.mesh1D, 232
  - fiPy.meshes.mesh2D, 240
  - fiPy.meshes.nonUniformGrid1D, 249
  - fiPy.meshes.nonUniformGrid2D, 257
  - fiPy.meshes.nonUniformGrid3D, 266
  - fiPy.meshes.periodicGrid1D, 274
  - fiPy.meshes.periodicGrid2D, 283
  - fiPy.meshes.periodicGrid3D, 309
  - fiPy.meshes.representations, 365
  - fiPy.meshes.representations.abstractRepresentation, 365
  - fiPy.meshes.representations.gridRepresentation, 365
  - fiPy.meshes.representations.meshRepresentation, 365
  - fiPy.meshes.skewedGrid2D, 365
  - fiPy.meshes.sphericalNonUniformGrid1D, 374
  - fiPy.meshes.sphericalUniformGrid1D, 383
  - fiPy.meshes.test, 390
  - fiPy.meshes.topologies, 390
  - fiPy.meshes.topologies.abstractTopology, 390
  - fiPy.meshes.topologies.gridTopology, 390
  - fiPy.meshes.topologies.meshTopology, 390
  - fiPy.meshes.tri2D, 390
  - fiPy.meshes.uniformGrid, 399
  - fiPy.meshes.uniformGrid1D, 406
  - fiPy.meshes.uniformGrid2D, 413
  - fiPy.meshes.uniformGrid3D, 420
  - fiPy.solvers, 427
  - fiPy.solvers.petsc, 429
  - fiPy.solvers.petsc.comms, 430
  - fiPy.solvers.petsc.comms.parallelPETScCommWrapper, 430
  - fiPy.solvers.petsc.comms.petscCommWrapper, 430
  - fiPy.solvers.petsc.comms.serialPETScCommWrapper, 431
  - fiPy.solvers.petsc.dummySolver, 431
  - fiPy.solvers.petsc.linearBicgSolver, 431

- fiPy.solvers.petsc.linearCGSSolver, 432
- fiPy.solvers.petsc.linearGMRESSolver, 432
- fiPy.solvers.petsc.linearLUSolver, 433
- fiPy.solvers.petsc.linearPCGSolver, 433
- fiPy.solvers.petsc.petscKrylovSolver, 434
- fiPy.solvers.petsc.petscSolver, 435
- fiPy.solvers.pyAMG, 435
- fiPy.solvers.pyAMG.linearCGSSolver, 436
- fiPy.solvers.pyAMG.linearGeneralSolver, 437
- fiPy.solvers.pyAMG.linearGMRESSolver, 436
- fiPy.solvers.pyAMG.linearLUSolver, 438
- fiPy.solvers.pyAMG.linearPCGSolver, 438
- fiPy.solvers.pyAMG.preconditioners, 439
- fiPy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner, 439
- fiPy.solvers.pyamgx, 439
- fiPy.solvers.pyamgx.aggregationAMGSolver, 440
- fiPy.solvers.pyamgx.classicalAMGSolver, 440
- fiPy.solvers.pyamgx.linearBiCGStabSolver, 441
- fiPy.solvers.pyamgx.linearCGSolver, 441
- fiPy.solvers.pyamgx.linearFGMRESSolver, 442
- fiPy.solvers.pyamgx.linearGMRESSolver, 443
- fiPy.solvers.pyamgx.linearLUSolver, 443
- fiPy.solvers.pyamgx.preconditioners, 444
- fiPy.solvers.pyamgx.preconditioners.preconditioners, 444
- fiPy.solvers.pyamgx.pyAMGXSolver, 444
- fiPy.solvers.pyamgx.smoothers, 445
- fiPy.solvers.pyamgx.smoothers.smoothers, 445
- fiPy.solvers.pysparse, 445
- fiPy.solvers.pysparse.linearCGSSolver, 446
- fiPy.solvers.pysparse.linearGMRESSolver, 446
- fiPy.solvers.pysparse.linearJORSolver, 447
- fiPy.solvers.pysparse.linearLUSolver, 447
- fiPy.solvers.pysparse.linearPCGSolver, 448
- fiPy.solvers.pysparse.preconditioners, 448
- fiPy.solvers.pysparse.preconditioners.jacobiPreconditioner, 448
- fiPy.solvers.pysparse.preconditioners.preconditioners, 449
- fiPy.solvers.pysparse.preconditioners.ssrPreconditioner, 449
- fiPy.solvers.pysparse.pysparseSolver, 449
- fiPy.solvers.pysparseMatrixSolver, 450
- fiPy.solvers.scipy, 450
- fiPy.solvers.scipy.linearBicgstabSolver, 450
- fiPy.solvers.scipy.linearCGSSolver, 451
- fiPy.solvers.scipy.linearGMRESSolver, 451
- fiPy.solvers.scipy.linearLUSolver, 452
- fiPy.solvers.scipy.linearPCGSolver, 452
- fiPy.solvers.scipy.scipyKrylovSolver, 453
- fiPy.solvers.scipy.scipySolver, 453
- fiPy.solvers.solver, 453
- fiPy.solvers.test, 458
- fiPy.solvers.trilinos, 458
- fiPy.solvers.trilinos.comms, 459
- fiPy.solvers.trilinos.comms.epetraCommWrapper, 459
- fiPy.solvers.trilinos.comms.parallelEpetraCommWrapper, 460
- fiPy.solvers.trilinos.comms.serialEpetraCommWrapper, 460
- fiPy.solvers.trilinos.linearBicgstabSolver, 461
- fiPy.solvers.trilinos.linearCGSSolver, 461
- fiPy.solvers.trilinos.linearGMRESSolver, 462
- fiPy.solvers.trilinos.linearLUSolver, 462
- fiPy.solvers.trilinos.linearPCGSolver, 463
- fiPy.solvers.trilinos.preconditioners, 464
- fiPy.solvers.trilinos.preconditioners.domDecompPreconditioner, 464
- fiPy.solvers.trilinos.preconditioners.icPreconditioner, 464
- fiPy.solvers.trilinos.preconditioners.jacobiPreconditioner, 465
- fiPy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner, 465
- fiPy.solvers.trilinos.preconditioners.multilevelDDPreconditioner, 465
- fiPy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner, 466
- fiPy.solvers.trilinos.preconditioners.multilevelSAPreconditioner, 466
- fiPy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner, 466
- fiPy.solvers.trilinos.preconditioners.multilevelSolverSmoothers, 467
- fiPy.solvers.trilinos.preconditioners.preconditioner, 467
- fiPy.solvers.trilinos.trilinosAztec00Solver, 468
- fiPy.solvers.trilinos.trilinosMLTest, 469
- fiPy.solvers.trilinos.trilinosNonlinearSolver, 469
- fiPy.solvers.trilinos.trilinosSolver, 470
- fiPy.steps, 471
- fiPy.steps.pidStepper, 473
- fiPy.steps.pseudoRKQSStepper, 474
- fiPy.steps.stepper, 474
- fiPy.terms, 474
- fiPy.terms.abstractBinaryTerm, 481
- fiPy.terms.abstractConvectionTerm, 481
- fiPy.terms.abstractDiffusionTerm, 481
- fiPy.terms.abstractUpwindConvectionTerm, 481
- fiPy.terms.advectionTerm, 481
- fiPy.terms.asymmetricConvectionTerm, 488
- fiPy.terms.binaryTerm, 488
- fiPy.terms.cellTerm, 488

- fiPy.terms.centralDiffConvectionTerm, 492
- fiPy.terms.coupledBinaryTerm, 497
- fiPy.terms.diffusionTerm, 497
- fiPy.terms.diffusionTermCorrection, 501
- fiPy.terms.diffusionTermNoCorrection, 504
- fiPy.terms.explicitDiffusionTerm, 508
- fiPy.terms.explicitSourceTerm, 511
- fiPy.terms.explicitUpwindConvectionTerm, 511
- fiPy.terms.exponentialConvectionTerm, 516
- fiPy.terms.faceTerm, 522
- fiPy.terms.firstOrderAdvectionTerm, 526
- fiPy.terms.hybridConvectionTerm, 531
- fiPy.terms.implicitDiffusionTerm, 536
- fiPy.terms.implicitSourceTerm, 536
- fiPy.terms.nonDiffusionTerm, 540
- fiPy.terms.powerLawConvectionTerm, 540
- fiPy.terms.residualTerm, 546
- fiPy.terms.sourceTerm, 550
- fiPy.terms.term, 554
- fiPy.terms.test, 558
- fiPy.terms.transientTerm, 558
- fiPy.terms.unaryTerm, 563
- fiPy.terms.upwindConvectionTerm, 563
- fiPy.terms.vanLeerConvectionTerm, 568
- fiPy.testFiPy, 573
- fiPy.tests, 573
- fiPy.tests.doctestPlus, 573
- fiPy.tests.lateImportTest, 575
- fiPy.tests.test, 575
- fiPy.tests.testProgram, 576
- fiPy.tools, 576
- fiPy.tools.comms, 593
- fiPy.tools.comms.commWrapper, 594
- fiPy.tools.comms.dummyComm, 594
- fiPy.tools.debug, 594
- fiPy.tools.decorators, 595
- fiPy.tools.dimensions, 595
- fiPy.tools.dimensions.DictWithDefault, 595
- fiPy.tools.dimensions.NumberDict, 595
- fiPy.tools.dimensions.physicalField, 595
- fiPy.tools.dump, 621
- fiPy.tools.inline, 622
- fiPy.tools.logging, 622
- fiPy.tools.logging.environment, 622
- fiPy.tools.numerix, 623
- fiPy.tools.parser, 628
- fiPy.tools.sharedtempfile, 629
- fiPy.tools.test, 630
- fiPy.tools.timer, 630
- fiPy.tools.vector, 630
- fiPy.tools.version, 631
- fiPy.variables, 631
- fiPy.variables.addOverFacesVariable, 633
- fiPy.variables.arithmeticCellToFaceVariable, 633
- fiPy.variables.betaNoiseVariable, 633
- fiPy.variables.binaryOperatorVariable, 649
- fiPy.variables.cellToFaceVariable, 649
- fiPy.variables.cellVariable, 649
- fiPy.variables.constant, 664
- fiPy.variables.constraintMask, 664
- fiPy.variables.coupledCellVariable, 664
- fiPy.variables.distanceVariable, 664
- fiPy.variables.exponentialNoiseVariable, 683
- fiPy.variables.faceGradContributionsVariable, 698
- fiPy.variables.faceGradVariable, 698
- fiPy.variables.faceVariable, 698
- fiPy.variables.gammaNoiseVariable, 709
- fiPy.variables.gaussCellGradVariable, 725
- fiPy.variables.gaussianNoiseVariable, 725
- fiPy.variables.harmonicCellToFaceVariable, 742
- fiPy.variables.histogramVariable, 742
- fiPy.variables.interfaceAreaVariable, 756
- fiPy.variables.interfaceFlagVariable, 756
- fiPy.variables.leastSquaresCellGradVariable, 756
- fiPy.variables.levelSetDiffusionVariable, 756
- fiPy.variables.meshVariable, 756
- fiPy.variables.minmodCellToFaceVariable, 767
- fiPy.variables.modCellGradVariable, 767
- fiPy.variables.modCellToFaceVariable, 767
- fiPy.variables.modFaceGradVariable, 767
- fiPy.variables.modPhysicalField, 767
- fiPy.variables.modularVariable, 767
- fiPy.variables.noiseVariable, 782
- fiPy.variables.operatorVariable, 798
- fiPy.variables.scharfetterGummelFaceVariable, 798
- fiPy.variables.surfactantConvectionVariable, 809
- fiPy.variables.surfactantVariable, 820
- fiPy.variables.test, 836
- fiPy.variables.unaryOperatorVariable, 836
- fiPy.variables.uniformNoiseVariable, 836
- fiPy.variables.variable, 851
- fiPy.viewers, 861
- fiPy.viewers.matplotlibViewer, 865
- fiPy.viewers.matplotlibViewer.abstractMatplotlib2DViewer, 867
- fiPy.viewers.matplotlibViewer.abstractMatplotlibViewer, 870
- fiPy.viewers.matplotlibViewer.matplotlib1DViewer, 872
- fiPy.viewers.matplotlibViewer.matplotlib2DContourViewer, 875

[fiPy.viewers.matplotlibViewer.matplotlib2DGridContourViewer](#),  
877

[fiPy.viewers.matplotlibViewer.matplotlib2DGridViewer](#),  
881

[fiPy.viewers.matplotlibViewer.matplotlib2DViewer](#),  
883

[fiPy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer](#),  
886

[fiPy.viewers.matplotlibViewer.matplotlibStreamViewer](#),  
886

[fiPy.viewers.matplotlibViewer.matplotlibVectorViewer](#),  
891

[fiPy.viewers.matplotlibViewer.test](#), 894

[fiPy.viewers.mayaviViewer](#), 894

[fiPy.viewers.mayaviViewer.mayaviClient](#), 898

[fiPy.viewers.mayaviViewer.mayaviDaemon](#), 901

[fiPy.viewers.mayaviViewer.test](#), 902

[fiPy.viewers.multiViewer](#), 902

[fiPy.viewers.test](#), 904

[fiPy.viewers.testinteractive](#), 904

[fiPy.viewers.tsvViewer](#), 904

[fiPy.viewers.viewer](#), 906

[fiPy.viewers.vtkViewer](#), 908

[fiPy.viewers.vtkViewer.test](#), 912

[fiPy.viewers.vtkViewer.vtkCellViewer](#), 912

[fiPy.viewers.vtkViewer.vtkFaceViewer](#), 914

[fiPy.viewers.vtkViewer.vtkViewer](#), 916

## p

[package](#), 121

[package.subpackage](#), 121

[package.subpackage.base](#), 122

[package.subpackage.object](#), 123

# Index

## Symbols

- `:math:`\pi``, 1094, 1099, 1120, 1125
- `__abs__` ○ (*fipy.tools.PhysicalField* method), 577
- `__abs__` ○ (*fipy.tools.dimensions.physicalField.PhysicalField* method), 599
- `__abs__` ○ (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 635
- `__abs__` ○ (*fipy.variables.cellVariable.CellVariable* method), 650
- `__abs__` ○ (*fipy.variables.distanceVariable.DistanceVariable* method), 667
- `__abs__` ○ (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 684
- `__abs__` ○ (*fipy.variables.faceVariable.FaceVariable* method), 699
- `__abs__` ○ (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 711
- `__abs__` ○ (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 727
- `__abs__` ○ (*fipy.variables.histogramVariable.HistogramVariable* method), 742
- `__abs__` ○ (*fipy.variables.meshVariable.MeshVariable* method), 757
- `__abs__` ○ (*fipy.variables.modularVariable.ModularVariable* method), 768
- `__abs__` ○ (*fipy.variables.noiseVariable.NoiseVariable* method), 783
- `__abs__` ○ (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 798
- `__abs__` ○ (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 810
- `__abs__` ○ (*fipy.variables.surfactantVariable.SurfactantVariable* method), 821
- `__abs__` ○ (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 837
- `__abs__` ○ (*fipy.variables.variable.Variable* method), 852
- `__add__` ○ (*fipy.meshes.abstractMesh.AbstractMesh* method), 132
- `__add__` ○ (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 141
- `__add__` ○ (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 149
- `__add__` ○ (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* method), 158
- `__add__` ○ (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* method), 165
- `__add__` ○ (*fipy.meshes.gmshMesh.Gmsh2D* method), 179
- `__add__` ○ (*fipy.meshes.gmshMesh.Gmsh2DIn3Dspace* method), 188
- `__add__` ○ (*fipy.meshes.gmshMesh.Gmsh3D* method), 196
- `__add__` ○ (*fipy.meshes.gmshMesh.GmshGrid2D* method), 205
- `__add__` ○ (*fipy.meshes.gmshMesh.GmshGrid3D* method), 213
- `__add__` ○ (*fipy.meshes.mesh.Mesh* method), 223
- `__add__` ○ (*fipy.meshes.mesh1D.Mesh1D* method), 232
- `__add__` ○ (*fipy.meshes.mesh2D.Mesh2D* method), 240
- `__add__` ○ (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 249
- `__add__` ○ (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 257
- `__add__` ○ (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 266
- `__add__` ○ (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 275
- `__add__` ○ (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 284
- `__add__` ○ (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 292
- `__add__` ○ (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 301
- `__add__` ○ (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 311
- `__add__` ○ (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 318
- `__add__` ○ (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 326
- `__add__` ○ (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 334
- `__add__` ○ (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 342
- `__add__` ○ (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 350
- `__add__` ○ (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 357
- `__add__` ○ (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 366
- `__add__` ○ (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 375
- `__add__` ○ (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* method), 383
- `__add__` ○ (*fipy.meshes.tri2D.Tri2D* method), 391
- `__add__` ○ (*fipy.meshes.uniformGrid.UniformGrid* method), 399
- `__add__` ○ (*fipy.meshes.uniformGrid1D.UniformGrid1D* method), 406
- `__add__` ○ (*fipy.meshes.uniformGrid2D.UniformGrid2D* method), 413
- `__add__` ○ (*fipy.meshes.uniformGrid3D.UniformGrid3D* method), 421
- `__add__` ○ (*fipy.tools.PhysicalField* method), 577
- `__add__` ○ (*fipy.tools.dimensions.physicalField.PhysicalField* method), 599

- `__and__` () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable method*), 635
- `__and__` () (*fipy.variables.cellVariable.CellVariable method*), 650
- `__and__` () (*fipy.variables.distanceVariable.DistanceVariable method*), 667
- `__and__` () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 684
- `__and__` () (*fipy.variables.faceVariable.FaceVariable method*), 699
- `__and__` () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 711
- `__and__` () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 728
- `__and__` () (*fipy.variables.histogramVariable.HistogramVariable method*), 742
- `__and__` () (*fipy.variables.meshVariable.MeshVariable method*), 757
- `__and__` () (*fipy.variables.modularVariable.ModularVariable method*), 768
- `__and__` () (*fipy.variables.noiseVariable.NoiseVariable method*), 784
- `__and__` () (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 798
- `__and__` () (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 810
- `__and__` () (*fipy.variables.surfactantVariable.SurfactantVariable method*), 821
- `__and__` () (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 837
- `__and__` () (*fipy.variables.variable.Variable method*), 852
- `__array__` () (*fipy.tools.PhysicalField method*), 577
- `__array__` () (*fipy.tools.dimensions.physicalField.PhysicalField method*), 599
- `__array__` () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable method*), 635
- `__array__` () (*fipy.variables.cellVariable.CellVariable method*), 650
- `__array__` () (*fipy.variables.distanceVariable.DistanceVariable method*), 668
- `__array__` () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 685
- `__array__` () (*fipy.variables.faceVariable.FaceVariable method*), 699
- `__array__` () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 712
- `__array__` () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 728
- `__array__` () (*fipy.variables.histogramVariable.HistogramVariable method*), 743
- `__array__` () (*fipy.variables.meshVariable.MeshVariable method*), 758
- `__array__` () (*fipy.variables.modularVariable.ModularVariable method*), 769
- `__array__` () (*fipy.variables.noiseVariable.NoiseVariable method*), 784
- `__array__` () (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 799
- `__array__` () (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 811
- `__array__` () (*fipy.variables.surfactantVariable.SurfactantVariable method*), 822
- `__array__` () (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 838
- `__array__` () (*fipy.variables.variable.Variable method*), 853
- `__bool__` () (*fipy.tools.PhysicalField method*), 578
- `__bool__` () (*fipy.tools.dimensions.physicalField.PhysicalField method*), 600
- `__bool__` () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable method*), 636
- `__bool__` () (*fipy.variables.cellVariable.CellVariable method*), 651
- `__bool__` () (*fipy.variables.distanceVariable.DistanceVariable method*), 668
- `__bool__` () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 685
- `__bool__` () (*fipy.variables.faceVariable.FaceVariable method*), 700
- `__bool__` () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 712
- `__bool__` () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 728
- `__bool__` () (*fipy.variables.histogramVariable.HistogramVariable method*), 743
- `__bool__` () (*fipy.variables.meshVariable.MeshVariable method*), 758
- `__bool__` () (*fipy.variables.modularVariable.ModularVariable method*), 769
- `__array_wrap__` () (*fipy.variables.variable.Variable method*), 853
- `__array_wrap__` () (*fipy.tools.PhysicalField method*), 578
- `__array_wrap__` () (*fipy.tools.dimensions.physicalField.PhysicalField method*), 600
- `__array_wrap__` () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable method*), 636
- `__array_wrap__` () (*fipy.variables.cellVariable.CellVariable method*), 650
- `__array_wrap__` () (*fipy.variables.distanceVariable.DistanceVariable method*), 668
- `__array_wrap__` () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 685
- `__array_wrap__` () (*fipy.variables.faceVariable.FaceVariable method*), 699
- `__array_wrap__` () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 712
- `__array_wrap__` () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 728
- `__array_wrap__` () (*fipy.variables.histogramVariable.HistogramVariable method*), 743
- `__array_wrap__` () (*fipy.variables.meshVariable.MeshVariable method*), 758
- `__array_wrap__` () (*fipy.variables.modularVariable.ModularVariable method*), 769

\_\_bool\_\_ () (fipy.variables.noiseVariable.NoiseVariable method), 784  
 \_\_bool\_\_ () (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 799  
 \_\_bool\_\_ () (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 811  
 \_\_bool\_\_ () (fipy.variables.surfactantVariable.SurfactantVariable method), 822  
 \_\_bool\_\_ () (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 838  
 \_\_bool\_\_ () (fipy.variables.variable.Variable method), 853  
 \_\_call\_\_ () (fipy.solvers.pyamgx.preconditioners.preconditioners.Preconditioner method), 444  
 \_\_call\_\_ () (fipy.solvers.pyamgx.smoothers.smoothers.Smoother method), 445  
 \_\_call\_\_ () (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 636  
 \_\_call\_\_ () (fipy.variables.cellVariable.CellVariable method), 651  
 \_\_call\_\_ () (fipy.variables.distanceVariable.DistanceVariable method), 668  
 \_\_call\_\_ () (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 685  
 \_\_call\_\_ () (fipy.variables.faceVariable.FaceVariable method), 700  
 \_\_call\_\_ () (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 712  
 \_\_call\_\_ () (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 729  
 \_\_call\_\_ () (fipy.variables.histogramVariable.HistogramVariable method), 743  
 \_\_call\_\_ () (fipy.variables.meshVariable.MeshVariable method), 758  
 \_\_call\_\_ () (fipy.variables.modularVariable.ModularVariable method), 769  
 \_\_call\_\_ () (fipy.variables.noiseVariable.NoiseVariable method), 785  
 \_\_call\_\_ () (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 799  
 \_\_call\_\_ () (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 811  
 \_\_call\_\_ () (fipy.variables.surfactantVariable.SurfactantVariable method), 822  
 \_\_call\_\_ () (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 838  
 \_\_call\_\_ () (fipy.variables.variable.Variable method), 853  
 \_\_cause\_\_ (fipy.meshes.abstractMesh.MeshAdditionError attribute), 139  
 \_\_cause\_\_ (fipy.meshes.gmshMesh.GmshException attribute), 204  
 \_\_cause\_\_ (fipy.meshes.gmshMesh.MeshExportError attribute), 222  
 \_\_cause\_\_ (fipy.meshes.mesh.MeshAdditionError attribute), 231  
 \_\_cause\_\_ (fipy.solvers.SerialSolverError attribute), 428  
 \_\_cause\_\_ (fipy.solvers.solver.IllConditionedPreconditionerWarning attribute), 453  
 \_\_cause\_\_ (fipy.solvers.solver.MatrixIllConditionedWarning attribute), 454  
 \_\_cause\_\_ (fipy.solvers.solver.MaximumIterationWarning attribute), 455  
 \_\_cause\_\_ (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning attribute), 455  
 \_\_cause\_\_ (fipy.solvers.solver.PreconditionerWarning attribute), 456  
 \_\_cause\_\_ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning attribute), 456  
 \_\_cause\_\_ (fipy.solvers.solver.SolverConvergenceWarning attribute), 457  
 \_\_cause\_\_ (fipy.solvers.solver.StagnatedSolverWarning attribute), 458  
 \_\_cause\_\_ (fipy.terms.AbstractBaseClassError attribute), 475  
 \_\_cause\_\_ (fipy.terms.ExplicitVariableError attribute), 475  
 \_\_cause\_\_ (fipy.terms.IncorrectSolutionVariable attribute), 476  
 \_\_cause\_\_ (fipy.terms.SolutionVariableNumberError attribute), 477  
 \_\_cause\_\_ (fipy.terms.SolutionVariableRequiredError attribute), 477  
 \_\_cause\_\_ (fipy.terms.TermMultiplyError attribute), 478  
 \_\_cause\_\_ (fipy.terms.TransientTermError attribute), 478  
 \_\_cause\_\_ (fipy.terms.VectorCoeffError attribute), 479  
 \_\_cause\_\_ (fipy.viewers.MeshDimensionError attribute), 863  
 \_\_context\_\_ (fipy.meshes.abstractMesh.MeshAdditionError attribute), 139  
 \_\_context\_\_ (fipy.meshes.gmshMesh.GmshException attribute), 204  
 \_\_context\_\_ (fipy.meshes.gmshMesh.MeshExportError attribute), 222  
 \_\_context\_\_ (fipy.meshes.mesh.MeshAdditionError attribute), 231  
 \_\_context\_\_ (fipy.solvers.SerialSolverError attribute), 428  
 \_\_context\_\_ (fipy.solvers.solver.IllConditionedPreconditionerWarning attribute), 453  
 \_\_context\_\_ (fipy.solvers.solver.MatrixIllConditionedWarning attribute), 454  
 \_\_context\_\_ (fipy.solvers.solver.MaximumIterationWarning attribute), 455  
 \_\_context\_\_ (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning attribute), 455  
 \_\_context\_\_ (fipy.solvers.solver.PreconditionerWarning attribute), 456  
 \_\_context\_\_ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning attribute), 456  
 \_\_context\_\_ (fipy.solvers.solver.SolverConvergenceWarning attribute), 457  
 \_\_context\_\_ (fipy.solvers.solver.StagnatedSolverWarning attribute), 458  
 \_\_context\_\_ (fipy.terms.AbstractBaseClassError attribute), 475  
 \_\_context\_\_ (fipy.terms.ExplicitVariableError attribute), 476  
 \_\_context\_\_ (fipy.terms.IncorrectSolutionVariable attribute), 476  
 \_\_context\_\_ (fipy.terms.SolutionVariableNumberError attribute), 477  
 \_\_context\_\_ (fipy.terms.SolutionVariableRequiredError attribute), 477  
 \_\_context\_\_ (fipy.terms.TermMultiplyError attribute), 478  
 \_\_context\_\_ (fipy.terms.TransientTermError attribute), 478  
 \_\_context\_\_ (fipy.terms.VectorCoeffError attribute), 479  
 \_\_context\_\_ (fipy.viewers.MeshDimensionError attribute), 863  
 \_\_delattr\_\_ () (fipy.meshes.abstractMesh.MeshAdditionError method), 139  
 \_\_delattr\_\_ () (fipy.meshes.gmshMesh.GmshException method), 204  
 \_\_delattr\_\_ () (fipy.meshes.gmshMesh.MeshExportError method), 222  
 \_\_delattr\_\_ () (fipy.meshes.mesh.MeshAdditionError method), 231  
 \_\_delattr\_\_ () (fipy.solvers.SerialSolverError method), 428  
 \_\_delattr\_\_ () (fipy.solvers.solver.IllConditionedPreconditionerWarning method), 453  
 \_\_delattr\_\_ () (fipy.solvers.solver.MatrixIllConditionedWarning method), 454  
 \_\_delattr\_\_ () (fipy.solvers.solver.MaximumIterationWarning method), 455  
 \_\_delattr\_\_ () (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning method), 455  
 \_\_delattr\_\_ () (fipy.solvers.solver.PreconditionerWarning method), 456

- `__delattr__` () (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning method), 456
- `__delattr__` () (fipy.solvers.solver.SolverConvergenceWarning method), 457
- `__delattr__` () (fipy.solvers.solver.StagnatedSolverWarning method), 458
- `__delattr__` () (fipy.terms.AbstractBaseClassError method), 475
- `__delattr__` () (fipy.terms.ExplicitVariableError method), 476
- `__delattr__` () (fipy.terms.IncorrectSolutionVariable method), 476
- `__delattr__` () (fipy.terms.SolutionVariableNumberError method), 477
- `__delattr__` () (fipy.terms.SolutionVariableRequiredError method), 477
- `__delattr__` () (fipy.terms.TermMultiplyError method), 478
- `__delattr__` () (fipy.terms.TransientTermError method), 478
- `__delattr__` () (fipy.terms.VectorCoeffError method), 479
- `__delattr__` () (fipy.viewers.MeshDimensionError method), 863
- `__div__` () (fipy.meshes.abstractMesh.AbstractMesh method), 134
- `__div__` () (fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D method), 143
- `__div__` () (fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D method), 151
- `__div__` () (fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D method), 160
- `__div__` () (fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D method), 167
- `__div__` () (fipy.meshes.gmshMesh.Gmsh2D method), 181
- `__div__` () (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace method), 189
- `__div__` () (fipy.meshes.gmshMesh.Gmsh3D method), 198
- `__div__` () (fipy.meshes.gmshMesh.GmshGrid2D method), 206
- `__div__` () (fipy.meshes.gmshMesh.GmshGrid3D method), 215
- `__div__` () (fipy.meshes.mesh.Mesh method), 225
- `__div__` () (fipy.meshes.mesh1D.Mesh1D method), 234
- `__div__` () (fipy.meshes.mesh2D.Mesh2D method), 242
- `__div__` () (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D method), 251
- `__div__` () (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D method), 259
- `__div__` () (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D method), 268
- `__div__` () (fipy.meshes.periodicGrid1D.PeriodicGrid1D method), 276
- `__div__` () (fipy.meshes.periodicGrid2D.PeriodicGrid2D method), 286
- `__div__` () (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight method), 294
- `__div__` () (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom method), 302
- `__div__` () (fipy.meshes.periodicGrid3D.PeriodicGrid3D method), 312
- `__div__` () (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack method), 320
- `__div__` () (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight method), 328
- `__div__` () (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack method), 336
- `__div__` () (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom method), 343
- `__div__` () (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom method), 351
- `__div__` () (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack method), 359
- `__div__` () (fipy.meshes.skewedGrid2D.SkewedGrid2D method), 367
- `__div__` () (fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D method), 376
- `__div__` () (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D method), 385
- `__div__` () (fipy.meshes.tri2D.Tri2D method), 392
- `__div__` () (fipy.meshes.uniformGrid.UniformGrid method), 401
- `__div__` () (fipy.meshes.uniformGrid1D.UniformGrid1D method), 408
- `__div__` () (fipy.meshes.uniformGrid2D.UniformGrid2D method), 415
- `__div__` () (fipy.meshes.uniformGrid3D.UniformGrid3D method), 422
- `__div__` () (fipy.tools.physicalField.physicalField.physicalField method), 578
- `__div__` () (fipy.tools.dimensions.physicalField.physicalField method), 600
- `__div__` () (fipy.tools.dimensions.physicalField.physicalField.physicalUnit method), 614
- `__eq__` () (fipy.terms.advectionTerm.AdvectionTerm method), 484
- `__eq__` () (fipy.terms.cellTerm.CellTerm method), 488
- `__eq__` () (fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm method), 493
- `__eq__` () (fipy.terms.diffusionTerm.DiffusionTerm method), 498
- `__eq__` () (fipy.terms.diffusionTermCorrection.DiffusionTermCorrection method), 501
- `__eq__` () (fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection method), 504
- `__eq__` () (fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm method), 508
- `__eq__` () (fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm method), 513
- `__eq__` () (fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm method), 518
- `__eq__` () (fipy.terms.faceTerm.FaceTerm method), 522
- `__eq__` () (fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm method), 527
- `__eq__` () (fipy.terms.hybridConvectionTerm.HybridConvectionTerm method), 532
- `__eq__` () (fipy.terms.implicitSourceTerm.ImplicitSourceTerm method), 537
- `__eq__` () (fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm method), 542
- `__eq__` () (fipy.terms.residualTerm.ResidualTerm method), 546
- `__eq__` () (fipy.terms.sourceTerm.SourceTerm method), 550
- `__eq__` () (fipy.terms.term.Term method), 555
- `__eq__` () (fipy.terms.transientTerm.TransientTerm method), 559
- `__eq__` () (fipy.terms.upwindConvectionTerm.UpwindConvectionTerm method), 564
- `__eq__` () (fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm method), 569
- `__eq__` () (fipy.tools.physicalField.physicalField method), 578
- `__eq__` () (fipy.tools.dimensions.physicalField.physicalField method), 600
- `__eq__` () (fipy.tools.dimensions.physicalField.physicalField.physicalUnit method), 615
- `__eq__` () (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 637
- `__eq__` () (fipy.variables.cellVariable.CellVariable method), 651



- \_\_eq\_\_ (fipy.variables.distanceVariable.DistanceVariable method), 669
- \_\_eq\_\_ (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 686
- \_\_eq\_\_ (fipy.variables.faceVariable.FaceVariable method), 700
- \_\_eq\_\_ (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 713
- \_\_eq\_\_ (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 729
- \_\_eq\_\_ (fipy.variables.histogramVariable.HistogramVariable method), 744
- \_\_eq\_\_ (fipy.variables.meshVariable.MeshVariable method), 758
- \_\_eq\_\_ (fipy.variables.modularVariable.ModularVariable method), 770
- \_\_eq\_\_ (fipy.variables.noiseVariable.NoiseVariable method), 785
- \_\_eq\_\_ (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 799
- \_\_eq\_\_ (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 811
- \_\_eq\_\_ (fipy.variables.surfactantVariable.SurfactantVariable method), 823
- \_\_eq\_\_ (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 839
- \_\_eq\_\_ (fipy.variables.variable.Variable method), 853
- \_\_float\_\_ (fipy.tools.physicalField.PhysicalField method), 578
- \_\_float\_\_ (fipy.tools.dimensions.physicalField.PhysicalField method), 600
- \_\_ge\_\_ (fipy.tools.physicalField.PhysicalField method), 579
- \_\_ge\_\_ (fipy.tools.dimensions.physicalField.PhysicalField method), 601
- \_\_ge\_\_ (fipy.tools.dimensions.physicalField.PhysicalUnit method), 615
- \_\_ge\_\_ (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 637
- \_\_ge\_\_ (fipy.variables.cellVariable.CellVariable method), 652
- \_\_ge\_\_ (fipy.variables.distanceVariable.DistanceVariable method), 669
- \_\_ge\_\_ (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 686
- \_\_ge\_\_ (fipy.variables.faceVariable.FaceVariable method), 700
- \_\_ge\_\_ (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 713
- \_\_ge\_\_ (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 729
- \_\_ge\_\_ (fipy.variables.histogramVariable.HistogramVariable method), 744
- \_\_ge\_\_ (fipy.variables.meshVariable.MeshVariable method), 759
- \_\_ge\_\_ (fipy.variables.modularVariable.ModularVariable method), 770
- \_\_ge\_\_ (fipy.variables.noiseVariable.NoiseVariable method), 785
- \_\_ge\_\_ (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 800
- \_\_ge\_\_ (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 811
- \_\_ge\_\_ (fipy.variables.surfactantVariable.SurfactantVariable method), 823
- \_\_ge\_\_ (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 839
- \_\_ge\_\_ (fipy.variables.variable.Variable method), 854
- \_\_getattr\_\_ (fipy.meshes.abstractMesh.MeshAdditionError method), 139
- \_\_getattr\_\_ (fipy.meshes.gmshMesh.GmshException method), 204
- \_\_getattr\_\_ (fipy.meshes.gmshMesh.MeshExportError method), 222
- \_\_getattr\_\_ (fipy.meshes.mesh.MeshAdditionError method), 231
- \_\_getattr\_\_ (fipy.solvers.SerialSolverError method), 428
- \_\_getattr\_\_ (fipy.solvers.solver.IllConditionedPreconditionerWarning method), 454
- \_\_getattr\_\_ (fipy.solvers.solver.MatrixIllConditionedWarning method), 454
- \_\_getattr\_\_ (fipy.solvers.solver.MaximumIterationWarning method), 455
- \_\_getattr\_\_ (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning method), 455
- \_\_getattr\_\_ (fipy.solvers.solver.PreconditionerWarning method), 456
- \_\_getattr\_\_ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning method), 456
- \_\_getattr\_\_ (fipy.solvers.solver.SolverConvergenceWarning method), 457
- \_\_getattr\_\_ (fipy.solvers.solver.StagnatedSolverWarning method), 458
- \_\_getattr\_\_ (fipy.terms.AbstractBaseClassError method), 475
- \_\_getattr\_\_ (fipy.terms.ExplicitVariableError method), 476
- \_\_getattr\_\_ (fipy.terms.IncorrectSolutionVariable method), 476
- \_\_getattr\_\_ (fipy.terms.SolutionVariableNumberError method), 477
- \_\_getattr\_\_ (fipy.terms.SolutionVariableRequiredError method), 477
- \_\_getattr\_\_ (fipy.terms.TermMultiplyError method), 478
- \_\_getattr\_\_ (fipy.terms.TransientTermError method), 479
- \_\_getattr\_\_ (fipy.terms.VectorCoeffError method), 479
- \_\_getattr\_\_ (fipy.viewers.MeshDimensionError method), 863
- \_\_getitem\_\_ (fipy.tools.physicalField.PhysicalField method), 579
- \_\_getitem\_\_ (fipy.tools.dimensions.physicalField.PhysicalField method), 601
- \_\_getitem\_\_ (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 637
- \_\_getitem\_\_ (fipy.variables.cellVariable.CellVariable method), 652
- \_\_getitem\_\_ (fipy.variables.distanceVariable.DistanceVariable method), 669
- \_\_getitem\_\_ (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 686
- \_\_getitem\_\_ (fipy.variables.faceVariable.FaceVariable method), 701
- \_\_getitem\_\_ (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 713
- \_\_getitem\_\_ (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 730
- \_\_getitem\_\_ (fipy.variables.histogramVariable.HistogramVariable method), 744
- \_\_getitem\_\_ (fipy.variables.meshVariable.MeshVariable method), 759
- \_\_getitem\_\_ (fipy.variables.modularVariable.ModularVariable method), 771

`__getitem__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 786  
`__getitem__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 800  
`__getitem__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 812  
`__getitem__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 824  
`__getitem__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 839  
`__getitem__()` (*fipy.variables.variable.Variable* method), 854  
`__getstate__()` (*fipy.meshes.abstractMesh.AbstractMesh* method), 134  
`__getstate__()` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 143  
`__getstate__()` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 151  
`__getstate__()` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* method), 160  
`__getstate__()` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* method), 167  
`__getstate__()` (*fipy.meshes.gmshMesh.Gmsh2D* method), 181  
`__getstate__()` (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* method), 189  
`__getstate__()` (*fipy.meshes.gmshMesh.Gmsh3D* method), 198  
`__getstate__()` (*fipy.meshes.gmshMesh.GmshGrid2D* method), 207  
`__getstate__()` (*fipy.meshes.gmshMesh.GmshGrid3D* method), 215  
`__getstate__()` (*fipy.meshes.mesh.Mesh* method), 225  
`__getstate__()` (*fipy.meshes.mesh1D.Mesh1D* method), 234  
`__getstate__()` (*fipy.meshes.mesh2D.Mesh2D* method), 242  
`__getstate__()` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 251  
`__getstate__()` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 259  
`__getstate__()` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 268  
`__getstate__()` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 276  
`__getstate__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 286  
`__getstate__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 294  
`__getstate__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 303  
`__getstate__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 312  
`__getstate__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 320  
`__getstate__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 328  
`__getstate__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 336  
`__getstate__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 344  
`__getstate__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 351  
`__getstate__()` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 359  
`__getstate__()` (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 367  
`__getstate__()` (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 377  
`__getstate__()` (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* method), 385  
`__getstate__()` (*fipy.meshes.tri2D.Tri2D* method), 393  
`__getstate__()` (*fipy.meshes.uniformGrid1D.UniformGrid1D* method), 401  
`__getstate__()` (*fipy.meshes.uniformGrid2D.UniformGrid2D* method), 408  
`__getstate__()` (*fipy.meshes.uniformGrid2D.UniformGrid2D* method), 415  
`__getstate__()` (*fipy.meshes.uniformGrid3D.UniformGrid3D* method), 422  
`__getstate__()` (*fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper* method), 430  
`__getstate__()` (*fipy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper* method), 430  
`__getstate__()` (*fipy.solvers.petsc.comms.serialPETScCommWrapper.SerialPETScCommWrapper* method), 431  
`__getstate__()` (*fipy.solvers.trilinos.comms.epetraCommWrapper.EpetraCommWrapper* method), 459  
`__getstate__()` (*fipy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper* method), 460  
`__getstate__()` (*fipy.solvers.trilinos.comms.serialEpetraCommWrapper.SerialEpetraCommWrapper* method), 460  
`__getstate__()` (*fipy.tools.comms.commWrapper.CommWrapper* method), 594  
`__getstate__()` (*fipy.tools.comms.dummyComm.DummyComm* method), 594  
`__getstate__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 637  
`__getstate__()` (*fipy.variables.cellVariable.CellVariable* method), 652  
`__getstate__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 670  
`__getstate__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 687  
`__getstate__()` (*fipy.variables.faceVariable.FaceVariable* method), 701  
`__getstate__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 714  
`__getstate__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 730  
`__getstate__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 744  
`__getstate__()` (*fipy.variables.meshVariable.MeshVariable* method),

759  
\_\_getstate\_\_() (fipy.variables.modularVariable.ModularVariable method), 771  
\_\_getstate\_\_() (fipy.variables.noiseVariable.NoiseVariable method), 786  
\_\_getstate\_\_() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 800  
\_\_getstate\_\_() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 812  
\_\_getstate\_\_() (fipy.variables.surfactantVariable.SurfactantVariable method), 824  
\_\_getstate\_\_() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 840  
\_\_getstate\_\_() (fipy.variables.variable.Variable method), 854  
\_\_gt\_\_() (fipy.tools.physicalField.PhysicalField method), 579  
\_\_gt\_\_() (fipy.tools.dimensions.physicalField.PhysicalField method), 601  
\_\_gt\_\_() (fipy.tools.dimensions.physicalField.PhysicalUnit method), 615  
\_\_gt\_\_() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 637  
\_\_gt\_\_() (fipy.variables.cellVariable.CellVariable method), 652  
\_\_gt\_\_() (fipy.variables.distanceVariable.DistanceVariable method), 670  
\_\_gt\_\_() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 687  
\_\_gt\_\_() (fipy.variables.faceVariable.FaceVariable method), 701  
\_\_gt\_\_() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 714  
\_\_gt\_\_() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 730  
\_\_gt\_\_() (fipy.variables.histogramVariable.HistogramVariable method), 745  
\_\_gt\_\_() (fipy.variables.meshVariable.MeshVariable method), 759  
\_\_gt\_\_() (fipy.variables.modularVariable.ModularVariable method), 771  
\_\_gt\_\_() (fipy.variables.noiseVariable.NoiseVariable method), 786  
\_\_gt\_\_() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 800  
\_\_gt\_\_() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 812  
\_\_gt\_\_() (fipy.variables.surfactantVariable.SurfactantVariable method), 824  
\_\_gt\_\_() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 840  
\_\_gt\_\_() (fipy.variables.variable.Variable method), 854  
\_\_hash\_\_() (fipy.tools.dimensions.physicalField.PhysicalUnit attribute), 615  
\_\_hash\_\_() (fipy.terms.advectionTerm.AdvectionTerm method), 484  
\_\_hash\_\_() (fipy.terms.cellTerm.CellTerm method), 488  
\_\_hash\_\_() (fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm method), 494  
\_\_hash\_\_() (fipy.terms.diffusionTerm.DiffusionTerm method), 498  
\_\_hash\_\_() (fipy.terms.diffusionTermCorrection.DiffusionTermCorrection method), 501  
\_\_hash\_\_() (fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection method), 505  
\_\_hash\_\_() (fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm method), 508  
\_\_hash\_\_() (fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm method), 513  
\_\_hash\_\_() (fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm method), 518  
\_\_hash\_\_() (fipy.terms.faceTerm.FaceTerm method), 522  
\_\_hash\_\_() (fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm method), 527  
\_\_hash\_\_() (fipy.terms.hybridConvectionTerm.HybridConvectionTerm method), 533  
\_\_hash\_\_() (fipy.terms.implicitSourceTerm.ImplicitSourceTerm method), 537  
\_\_hash\_\_() (fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm method), 542  
\_\_hash\_\_() (fipy.terms.residualTerm.ResidualTerm method), 546  
\_\_hash\_\_() (fipy.terms.sourceTerm.SourceTerm method), 550  
\_\_hash\_\_() (fipy.terms.term.Term method), 555  
\_\_hash\_\_() (fipy.terms.transientTerm.TransientTerm method), 559  
\_\_hash\_\_() (fipy.terms.upwindConvectionTerm.UpwindConvectionTerm method), 564  
\_\_hash\_\_() (fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm method), 569  
\_\_hash\_\_() (fipy.tools.physicalField.PhysicalField method), 580  
\_\_hash\_\_() (fipy.tools.dimensions.physicalField.PhysicalField method), 602  
\_\_hash\_\_() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 638  
\_\_hash\_\_() (fipy.variables.cellVariable.CellVariable method), 652  
\_\_hash\_\_() (fipy.variables.distanceVariable.DistanceVariable method), 670  
\_\_hash\_\_() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 687  
\_\_hash\_\_() (fipy.variables.faceVariable.FaceVariable method), 701  
\_\_hash\_\_() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 714  
\_\_hash\_\_() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 730  
\_\_hash\_\_() (fipy.variables.histogramVariable.HistogramVariable method), 745  
\_\_hash\_\_() (fipy.variables.meshVariable.MeshVariable method), 759  
\_\_hash\_\_() (fipy.variables.modularVariable.ModularVariable method), 771  
\_\_hash\_\_() (fipy.variables.noiseVariable.NoiseVariable method), 786  
\_\_hash\_\_() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 801  
\_\_hash\_\_() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 812  
\_\_hash\_\_() (fipy.variables.surfactantVariable.SurfactantVariable method), 824  
\_\_hash\_\_() (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 840  
\_\_hash\_\_() (fipy.variables.variable.Variable method), 855  
\_\_invert\_\_() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 638  
\_\_invert\_\_() (fipy.variables.cellVariable.CellVariable method), 652

- `__invert__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 670
- `__invert__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 687
- `__invert__()` (*fipy.variables.faceVariable.FaceVariable* method), 701
- `__invert__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 714
- `__invert__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 730
- `__invert__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 745
- `__invert__()` (*fipy.variables.meshVariable.MeshVariable* method), 760
- `__invert__()` (*fipy.variables.modularVariable.ModularVariable* method), 771
- `__invert__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 786
- `__invert__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 801
- `__invert__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 812
- `__invert__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 824
- `__invert__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 840
- `__invert__()` (*fipy.variables.variable.Variable* method), 855
- `__le__()` (*fipy.tools.physicalField.physicalField.PhysicalField* method), 580
- `__le__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 602
- `__le__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 615
- `__le__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 638
- `__le__()` (*fipy.variables.cellVariable.CellVariable* method), 653
- `__le__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 670
- `__le__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 687
- `__le__()` (*fipy.variables.faceVariable.FaceVariable* method), 701
- `__le__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 714
- `__le__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 730
- `__le__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 745
- `__le__()` (*fipy.variables.meshVariable.MeshVariable* method), 760
- `__le__()` (*fipy.variables.modularVariable.ModularVariable* method), 771
- `__le__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 786
- `__le__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 801
- `__le__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 812
- `__le__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 824
- `__le__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 840
- `__le__()` (*fipy.variables.variable.Variable* method), 855
- `__lt__()` (*fipy.tools.physicalField.physicalField.PhysicalField* method), 580
- `__lt__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 602
- `__lt__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 615
- `__lt__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 638
- `__lt__()` (*fipy.variables.cellVariable.CellVariable* method), 653
- `__lt__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 670
- `__lt__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 687
- `__lt__()` (*fipy.variables.faceVariable.FaceVariable* method), 702
- `__lt__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 714
- `__lt__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 731
- `__lt__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 745
- `__lt__()` (*fipy.variables.meshVariable.MeshVariable* method), 760
- `__lt__()` (*fipy.variables.modularVariable.ModularVariable* method), 772
- `__lt__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 787
- `__lt__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 801
- `__lt__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 813
- `__lt__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 825
- `__lt__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 840
- `__lt__()` (*fipy.variables.variable.Variable* method), 855
- `__mod__()` (*fipy.tools.physicalField.physicalField.PhysicalField* method), 580
- `__mod__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 602
- `__mul__()` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 143
- `__mul__()` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 151
- `__mul__()` (*fipy.meshes.gmshMesh.Gmsh2D* method), 181
- `__mul__()` (*fipy.meshes.gmshMesh.Gmsh2DIn3Dspace* method), 189
- `__mul__()` (*fipy.meshes.gmshMesh.Gmsh3D* method), 198
- `__mul__()` (*fipy.meshes.gmshMesh.GmshGrid2D* method), 207
- `__mul__()` (*fipy.meshes.gmshMesh.GmshGrid3D* method), 215
- `__mul__()` (*fipy.meshes.mesh.Mesh* method), 225
- `__mul__()` (*fipy.meshes.mesh1D.Mesh1D* method), 234
- `__mul__()` (*fipy.meshes.mesh2D.Mesh2D* method), 242
- `__mul__()` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 251
- `__mul__()` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 259
- `__mul__()` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 268
- `__mul__()` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 276
- `__mul__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 286
- `__mul__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 294
- `__mul__()` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 303

- `__mul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 312
- `__mul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 320
- `__mul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 328
- `__mul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 336
- `__mul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 344
- `__mul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 351
- `__mul__` () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 359
- `__mul__` () (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 367
- `__mul__` () (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 377
- `__mul__` () (*fipy.meshes.tri2D.Tri2D* method), 393
- `__mul__` () (*fipy.terms.advectionTerm.AdvectionTerm* method), 484
- `__mul__` () (*fipy.terms.cellTerm.CellTerm* method), 488
- `__mul__` () (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 494
- `__mul__` () (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 513
- `__mul__` () (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 518
- `__mul__` () (*fipy.terms.faceTerm.FaceTerm* method), 522
- `__mul__` () (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 527
- `__mul__` () (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 533
- `__mul__` () (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 537
- `__mul__` () (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 542
- `__mul__` () (*fipy.terms.residualTerm.ResidualTerm* method), 546
- `__mul__` () (*fipy.terms.sourceTerm.SourceTerm* method), 550
- `__mul__` () (*fipy.terms.transientTerm.TransientTerm* method), 559
- `__mul__` () (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 564
- `__mul__` () (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 569
- `__mul__` () (*fipy.tools.PhysicalField* method), 580
- `__mul__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 602
- `__mul__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 615
- `__ne__` () (*fipy.tools.PhysicalField* method), 580
- `__ne__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 602
- `__ne__` () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 616
- `__ne__` () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 639
- `__ne__` () (*fipy.variables.cellVariable.CellVariable* method), 653
- `__ne__` () (*fipy.variables.distanceVariable.DistanceVariable* method), 671
- `__ne__` () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 688
- `__ne__` () (*fipy.variables.faceVariable.FaceVariable* method), 702
- `__ne__` () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 715
- `__ne__` () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 731
- `__ne__` () (*fipy.variables.histogramVariable.HistogramVariable* method), 746
- `__ne__` () (*fipy.variables.meshVariable.MeshVariable* method), 760
- `__ne__` () (*fipy.variables.modularVariable.ModularVariable* method), 772
- `__ne__` () (*fipy.variables.noiseVariable.NoiseVariable* method), 787
- `__ne__` () (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 801
- `__ne__` () (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 813
- `__ne__` () (*fipy.variables.surfactantVariable.SurfactantVariable* method), 825
- `__ne__` () (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 841
- `__ne__` () (*fipy.variables.variable.Variable* method), 855
- `__neg__` () (*fipy.terms.advectionTerm.AdvectionTerm* method), 484
- `__neg__` () (*fipy.terms.cellTerm.CellTerm* method), 489
- `__neg__` () (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 494
- `__neg__` () (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 513
- `__neg__` () (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 518
- `__neg__` () (*fipy.terms.faceTerm.FaceTerm* method), 523
- `__neg__` () (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 528
- `__neg__` () (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 533
- `__neg__` () (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 537
- `__neg__` () (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 542
- `__neg__` () (*fipy.terms.residualTerm.ResidualTerm* method), 546
- `__neg__` () (*fipy.terms.sourceTerm.SourceTerm* method), 551
- `__neg__` () (*fipy.terms.transientTerm.TransientTerm* method), 559
- `__neg__` () (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 564
- `__neg__` () (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 569
- `__neg__` () (*fipy.tools.PhysicalField* method), 580
- `__neg__` () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 602
- `__new__` () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* static method), 639
- `__new__` () (*fipy.variables.cellVariable.CellVariable* static method), 653
- `__new__` () (*fipy.variables.distanceVariable.DistanceVariable* static method), 671
- `__new__` () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* static method), 688

*static method*), 688  
 \_\_new\_\_ () (*fipy.variables.faceVariable.FaceVariable static method*), 702  
 \_\_new\_\_ () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable static method*), 715  
 \_\_new\_\_ () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable static method*), 731  
 \_\_new\_\_ () (*fipy.variables.histogramVariable.HistogramVariable static method*), 746  
 \_\_new\_\_ () (*fipy.variables.meshVariable.MeshVariable static method*), 761  
 \_\_new\_\_ () (*fipy.variables.modularVariable.ModularVariable static method*), 772  
 \_\_new\_\_ () (*fipy.variables.noiseVariable.NoiseVariable static method*), 787  
 \_\_new\_\_ () (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable static method*), 802  
 \_\_new\_\_ () (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable static method*), 813  
 \_\_new\_\_ () (*fipy.variables.surfactantVariable.SurfactantVariable static method*), 825  
 \_\_new\_\_ () (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable static method*), 841  
 \_\_new\_\_ () (*fipy.variables.variable.Variable static method*), 856  
 \_\_nonzero\_\_ () (*fipy.tools.physicalField.physicalField method*), 580  
 \_\_nonzero\_\_ () (*fipy.tools.dimensions.physicalField.PhysicalField method*), 602  
 \_\_nonzero\_\_ () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable method*), 639  
 \_\_nonzero\_\_ () (*fipy.variables.cellVariable.CellVariable method*), 654  
 \_\_nonzero\_\_ () (*fipy.variables.distanceVariable.DistanceVariable method*), 671  
 \_\_nonzero\_\_ () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 688  
 \_\_nonzero\_\_ () (*fipy.variables.faceVariable.FaceVariable method*), 702  
 \_\_nonzero\_\_ () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 715  
 \_\_nonzero\_\_ () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 731  
 \_\_nonzero\_\_ () (*fipy.variables.histogramVariable.HistogramVariable method*), 746  
 \_\_nonzero\_\_ () (*fipy.variables.meshVariable.MeshVariable method*), 761  
 \_\_nonzero\_\_ () (*fipy.variables.modularVariable.ModularVariable method*), 772  
 \_\_nonzero\_\_ () (*fipy.variables.noiseVariable.NoiseVariable method*), 787  
 \_\_nonzero\_\_ () (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 802  
 \_\_nonzero\_\_ () (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 813  
 \_\_nonzero\_\_ () (*fipy.variables.surfactantVariable.SurfactantVariable method*), 825  
 \_\_nonzero\_\_ () (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 841  
 \_\_nonzero\_\_ () (*fipy.variables.variable.Variable method*), 856  
 \_\_or\_\_ () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable method*), 639  
 \_\_or\_\_ () (*fipy.variables.cellVariable.CellVariable method*), 654  
 \_\_or\_\_ () (*fipy.variables.distanceVariable.DistanceVariable method*), 671  
 \_\_or\_\_ () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 688  
 \_\_or\_\_ () (*fipy.variables.faceVariable.FaceVariable method*), 702  
 \_\_or\_\_ () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 715  
 \_\_or\_\_ () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 732  
 \_\_or\_\_ () (*fipy.variables.histogramVariable.HistogramVariable method*), 746  
 \_\_or\_\_ () (*fipy.variables.meshVariable.MeshVariable method*), 761  
 \_\_or\_\_ () (*fipy.variables.modularVariable.ModularVariable method*), 773  
 \_\_or\_\_ () (*fipy.variables.noiseVariable.NoiseVariable method*), 788  
 \_\_or\_\_ () (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 802  
 \_\_or\_\_ () (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 814  
 \_\_or\_\_ () (*fipy.variables.surfactantVariable.SurfactantVariable method*), 825  
 \_\_or\_\_ () (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 842  
 \_\_or\_\_ () (*fipy.variables.variable.Variable method*), 856  
 \_\_pow\_\_ () (*fipy.tools.physicalField.physicalField method*), 581  
 \_\_pow\_\_ () (*fipy.tools.dimensions.physicalField.PhysicalField method*), 603  
 \_\_pow\_\_ () (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 616  
 \_\_pow\_\_ () (*fipy.variables.betaNoiseVariable.BetaNoiseVariable method*), 639  
 \_\_pow\_\_ () (*fipy.variables.cellVariable.CellVariable method*), 654  
 \_\_pow\_\_ () (*fipy.variables.distanceVariable.DistanceVariable method*), 672  
 \_\_pow\_\_ () (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 689  
 \_\_pow\_\_ () (*fipy.variables.faceVariable.FaceVariable method*), 703  
 \_\_pow\_\_ () (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 716  
 \_\_pow\_\_ () (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 732  
 \_\_pow\_\_ () (*fipy.variables.histogramVariable.HistogramVariable method*), 746  
 \_\_pow\_\_ () (*fipy.variables.meshVariable.MeshVariable method*), 761  
 \_\_pow\_\_ () (*fipy.variables.modularVariable.ModularVariable method*), 773  
 \_\_pow\_\_ () (*fipy.variables.noiseVariable.NoiseVariable method*), 788  
 \_\_pow\_\_ () (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 802  
 \_\_pow\_\_ () (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 814  
 \_\_pow\_\_ () (*fipy.variables.surfactantVariable.SurfactantVariable method*), 826  
 \_\_pow\_\_ () (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 842  
 \_\_pow\_\_ () (*fipy.variables.variable.Variable method*), 856  
 \_\_radd\_\_ () (*fipy.meshes.abstractMesh.AbstractMesh method*), 134

- `__radd__()` (*fiPy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 143
- `__radd__()` (*fiPy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 151
- `__radd__()` (*fiPy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* method), 160
- `__radd__()` (*fiPy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* method), 167
- `__radd__()` (*fiPy.meshes.gmshMesh.Gmsh2D* method), 181
- `__radd__()` (*fiPy.meshes.gmshMesh.Gmsh2DIn3DSpace* method), 190
- `__radd__()` (*fiPy.meshes.gmshMesh.Gmsh3D* method), 199
- `__radd__()` (*fiPy.meshes.gmshMesh.GmshGrid2D* method), 207
- `__radd__()` (*fiPy.meshes.gmshMesh.GmshGrid3D* method), 215
- `__radd__()` (*fiPy.meshes.mesh.Mesh* method), 226
- `__radd__()` (*fiPy.meshes.mesh1D.Mesh1D* method), 234
- `__radd__()` (*fiPy.meshes.mesh2D.Mesh2D* method), 242
- `__radd__()` (*fiPy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 252
- `__radd__()` (*fiPy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 260
- `__radd__()` (*fiPy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 268
- `__radd__()` (*fiPy.meshes.periodicGrid1D.PeriodicGrid1D* method), 277
- `__radd__()` (*fiPy.meshes.periodicGrid2D.PeriodicGrid2D* method), 286
- `__radd__()` (*fiPy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 295
- `__radd__()` (*fiPy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 303
- `__radd__()` (*fiPy.meshes.periodicGrid3D.PeriodicGrid3D* method), 313
- `__radd__()` (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 321
- `__radd__()` (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 329
- `__radd__()` (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 336
- `__radd__()` (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 344
- `__radd__()` (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 352
- `__radd__()` (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 360
- `__radd__()` (*fiPy.meshes.skewedGrid2D.SkewedGrid2D* method), 368
- `__radd__()` (*fiPy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 377
- `__radd__()` (*fiPy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* method), 385
- `__radd__()` (*fiPy.meshes.tri2D.Tri2D* method), 393
- `__radd__()` (*fiPy.meshes.uniformGrid.UniformGrid* method), 401
- `__radd__()` (*fiPy.meshes.uniformGrid1D.UniformGrid1D* method), 408
- `__radd__()` (*fiPy.meshes.uniformGrid2D.UniformGrid2D* method), 415
- `__radd__()` (*fiPy.meshes.uniformGrid3D.UniformGrid3D* method), 422
- `__radd__()` (*fiPy.tools.PhysicalField* method), 581
- `__radd__()` (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 603
- `__rdiv__()` (*fiPy.tools.dimensions.physicalField.PhysicalUnit* method), 616
- `__reduce__()` (*fiPy.meshes.abstractMesh.MeshAdditionError* method), 139
- `__reduce__()` (*fiPy.meshes.gmshMesh.GmshException* method), 204
- `__reduce__()` (*fiPy.meshes.gmshMesh.MeshExportError* method), 222
- `__reduce__()` (*fiPy.meshes.mesh.MeshAdditionError* method), 231
- `__reduce__()` (*fiPy.solvers.SerialSolverError* method), 428
- `__reduce__()` (*fiPy.solvers.solver.IllConditionedPreconditionerWarning* method), 454
- `__reduce__()` (*fiPy.solvers.solver.MatrixIllConditionedWarning* method), 454
- `__reduce__()` (*fiPy.solvers.solver.MaximumIterationWarning* method), 455
- `__reduce__()` (*fiPy.solvers.solver.PreconditionerNotPositiveDefiniteWarning* method), 455
- `__reduce__()` (*fiPy.solvers.solver.PreconditionerWarning* method), 456
- `__reduce__()` (*fiPy.solvers.solver.ScalarQuantityOutOfRangeWarning* method), 456
- `__reduce__()` (*fiPy.solvers.solver.SolverConvergenceWarning* method), 457
- `__reduce__()` (*fiPy.solvers.solver.StagnatedSolverWarning* method), 458
- `__reduce__()` (*fiPy.terms.AbstractBaseClassError* method), 475
- `__reduce__()` (*fiPy.terms.ExplicitVariableError* method), 476
- `__reduce__()` (*fiPy.terms.IncorrectSolutionVariable* method), 476
- `__reduce__()` (*fiPy.terms.SolutionVariableNumberError* method), 477
- `__reduce__()` (*fiPy.terms.SolutionVariableRequiredError* method), 477
- `__reduce__()` (*fiPy.terms.TermMultiplyError* method), 478
- `__reduce__()` (*fiPy.terms.TransientTermError* method), 479
- `__reduce__()` (*fiPy.terms.VectorCoeffError* method), 479
- `__reduce__()` (*fiPy.viewers.MeshDimensionError* method), 863
- `__repr__()` (*fiPy.boundaryConditions.boundaryCondition.BoundaryCondition* method), 127
- `__repr__()` (*fiPy.boundaryConditions.constraint.Constraint* method), 127
- `__repr__()` (*fiPy.boundaryConditions.fixedFlux.FixedFlux* method), 128
- `__repr__()` (*fiPy.boundaryConditions.fixedValue.FixedValue* method), 129
- `__repr__()` (*fiPy.boundaryConditions.nthOrderBoundaryCondition.NthOrderBoundaryCondition* method), 129
- `__repr__()` (*fiPy.meshes.abstractMesh.AbstractMesh* method), 136
- `__repr__()` (*fiPy.meshes.abstractMesh.MeshAdditionError* method), 139
- `__repr__()` (*fiPy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 145
- `__repr__()` (*fiPy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 153
- `__repr__()` (*fiPy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* method), 161
- `__repr__()` (*fiPy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* method), 169
- `__repr__()` (*fiPy.meshes.gmshMesh.Gmsh2D* method), 183
- `__repr__()` (*fiPy.meshes.gmshMesh.Gmsh2DIn3DSpace* method), 192

- \_\_repr\_\_ (fipy.meshes.gmshMesh.Gmsh3D method), 200
- \_\_repr\_\_ (fipy.meshes.gmshMesh.GmshException method), 204
- \_\_repr\_\_ (fipy.meshes.gmshMesh.GmshGrid2D method), 209
- \_\_repr\_\_ (fipy.meshes.gmshMesh.GmshGrid3D method), 217
- \_\_repr\_\_ (fipy.meshes.gmshMesh.MeshExportError method), 222
- \_\_repr\_\_ (fipy.meshes.mesh.Mesh method), 227
- \_\_repr\_\_ (fipy.meshes.mesh.MeshAdditionError method), 231
- \_\_repr\_\_ (fipy.meshes.mesh1D.Mesh1D method), 236
- \_\_repr\_\_ (fipy.meshes.mesh2D.Mesh2D method), 244
- \_\_repr\_\_ (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D method), 253
- \_\_repr\_\_ (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D method), 261
- \_\_repr\_\_ (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D method), 270
- \_\_repr\_\_ (fipy.meshes.periodicGrid1D.PeriodicGrid1D method), 279
- \_\_repr\_\_ (fipy.meshes.periodicGrid2D.PeriodicGrid2D method), 288
- \_\_repr\_\_ (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight method), 296
- \_\_repr\_\_ (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom method), 305
- \_\_repr\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3D method), 315
- \_\_repr\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack method), 322
- \_\_repr\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight method), 330
- \_\_repr\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack method), 338
- \_\_repr\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom method), 346
- \_\_repr\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom method), 353
- \_\_repr\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack method), 361
- \_\_repr\_\_ (fipy.meshes.skewedGrid2D.SkewedGrid2D method), 370
- \_\_repr\_\_ (fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D method), 379
- \_\_repr\_\_ (fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D method), 386
- \_\_repr\_\_ (fipy.meshes.tri2D.Tri2D method), 395
- \_\_repr\_\_ (fipy.meshes.uniformGrid.UniformGrid method), 403
- \_\_repr\_\_ (fipy.meshes.uniformGrid1D.UniformGrid1D method), 410
- \_\_repr\_\_ (fipy.meshes.uniformGrid2D.UniformGrid2D method), 417
- \_\_repr\_\_ (fipy.meshes.uniformGrid3D.UniformGrid3D method), 424
- \_\_repr\_\_ (fipy.solvers.SerialSolverError method), 428
- \_\_repr\_\_ (fipy.solvers.petsc.comms.parallelPETScCommWrapper.ParallelPETScCommWrapper method), 430
- \_\_repr\_\_ (fipy.solvers.petsc.comms.petscCommWrapper.PETScCommWrapper method), 430
- \_\_repr\_\_ (fipy.solvers.petsc.comms.serialPETScCommWrapper.SerialPETScCommWrapper method), 431
- \_\_repr\_\_ (fipy.solvers.petsc.dummySolver.DummySolver method), 431
- \_\_repr\_\_ (fipy.solvers.petsc.linearBicgSolver.LinearBicgSolver method), 432
- \_\_repr\_\_ (fipy.solvers.petsc.linearCGSSolver.LinearCGSSolver method), 432
- \_\_repr\_\_ (fipy.solvers.petsc.linearGMRESSolver.LinearGMRESSolver method), 433
- \_\_repr\_\_ (fipy.solvers.petsc.linearLUSolver.LinearLUSolver method), 433
- \_\_repr\_\_ (fipy.solvers.petsc.linearPCGSolver.LinearPCGSolver method), 434
- \_\_repr\_\_ (fipy.solvers.petsc.petscKrylovSolver.PETScKrylovSolver method), 434
- \_\_repr\_\_ (fipy.solvers.petsc.petscSolver.PETScSolver method), 435
- \_\_repr\_\_ (fipy.solvers.pyAMG.linearCGSSolver.LinearCGSSolver method), 436
- \_\_repr\_\_ (fipy.solvers.pyAMG.linearGMRESSolver.LinearGMRESSolver method), 437
- \_\_repr\_\_ (fipy.solvers.pyAMG.linearGeneralSolver.LinearGeneralSolver method), 437
- \_\_repr\_\_ (fipy.solvers.pyAMG.linearLUSolver.LinearLUSolver method), 438
- \_\_repr\_\_ (fipy.solvers.pyAMG.linearPCGSolver.LinearPCGSolver method), 438
- \_\_repr\_\_ (fipy.solvers.pyamgx.aggregationAMGSolver.AggregationAMGSolver method), 440
- \_\_repr\_\_ (fipy.solvers.pyamgx.classicalAMGSolver.ClassicalAMGSolver method), 441
- \_\_repr\_\_ (fipy.solvers.pyamgx.linearBiCGStabSolver.LinearBiCGStabSolver method), 441
- \_\_repr\_\_ (fipy.solvers.pyamgx.linearCGSolver.LinearCGSolver method), 442
- \_\_repr\_\_ (fipy.solvers.pyamgx.linearFGMRESSolver.LinearFGMRESSolver method), 442
- \_\_repr\_\_ (fipy.solvers.pyamgx.linearGMRESSolver.LinearGMRESSolver method), 443
- \_\_repr\_\_ (fipy.solvers.pyamgx.linearLUSolver.LinearLUSolver method), 443
- \_\_repr\_\_ (fipy.solvers.pyamgx.pyAMGXSolver.PyAMGXSolver method), 444
- \_\_repr\_\_ (fipy.solvers.pysparse.linearCGSSolver.LinearCGSSolver method), 446
- \_\_repr\_\_ (fipy.solvers.pysparse.linearGMRESSolver.LinearGMRESSolver method), 446
- \_\_repr\_\_ (fipy.solvers.pysparse.linearJORSolver.LinearJORSolver method), 447
- \_\_repr\_\_ (fipy.solvers.pysparse.linearLUSolver.LinearLUSolver method), 447
- \_\_repr\_\_ (fipy.solvers.pysparse.linearPCGSolver.LinearPCGSolver method), 447
- \_\_repr\_\_ (fipy.solvers.pysparse.pysparseSolver.PysparseSolver method), 450
- \_\_repr\_\_ (fipy.solvers.scipy.linearBicgstabSolver.LinearBicgstabSolver method), 450
- \_\_repr\_\_ (fipy.solvers.scipy.linearCGSSolver.LinearCGSSolver method), 451
- \_\_repr\_\_ (fipy.solvers.scipy.linearGMRESSolver.LinearGMRESSolver method), 451



- \_\_repr\_\_ (fipy.solvers.scipy.linearLUSolver.LinearLUSolver method), 452
- \_\_repr\_\_ (fipy.solvers.scipy.linearPCGSolver.LinearPCGSolver method), 452
- \_\_repr\_\_ (fipy.solvers.solver.IllConditionedPreconditionerWarning method), 454
- \_\_repr\_\_ (fipy.solvers.solver.MatrixIllConditionedWarning method), 454
- \_\_repr\_\_ (fipy.solvers.solver.MaximumIterationWarning method), 455
- \_\_repr\_\_ (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning method), 455
- \_\_repr\_\_ (fipy.solvers.solver.PreconditionerWarning method), 456
- \_\_repr\_\_ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning method), 457
- \_\_repr\_\_ (fipy.solvers.solver.Solver method), 457
- \_\_repr\_\_ (fipy.solvers.solver.SolverConvergenceWarning method), 457
- \_\_repr\_\_ (fipy.solvers.solver.StagnatedSolverWarning method), 458
- \_\_repr\_\_ (fipy.solvers.trilinos.comms.epetraCommWrapper.EpetraCommWrapper method), 460
- \_\_repr\_\_ (fipy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper method), 460
- \_\_repr\_\_ (fipy.solvers.trilinos.comms.serialEpetraCommWrapper.SerialEpetraCommWrapper method), 461
- \_\_repr\_\_ (fipy.solvers.trilinos.linearBicgstabSolver.LinearBicgstabSolver method), 461
- \_\_repr\_\_ (fipy.solvers.trilinos.linearCGSSolver.LinearCGSSolver method), 462
- \_\_repr\_\_ (fipy.solvers.trilinos.linearGMRESSolver.LinearGMRESSolver method), 462
- \_\_repr\_\_ (fipy.solvers.trilinos.linearLUSolver.LinearLUSolver method), 463
- \_\_repr\_\_ (fipy.solvers.trilinos.linearPCGSolver.LinearPCGSolver method), 463
- \_\_repr\_\_ (fipy.solvers.trilinos.trilinosAztecOOSolver.TrilinosAztecOOSolver method), 468
- \_\_repr\_\_ (fipy.solvers.trilinos.trilinosMLTest.TrilinosMLTest method), 469
- \_\_repr\_\_ (fipy.solvers.trilinos.trilinosNonlinearSolver.TrilinosNonlinearSolver method), 470
- \_\_repr\_\_ (fipy.solvers.trilinos.trilinosSolver.TrilinosSolver method), 471
- \_\_repr\_\_ (fipy.terms.AbstractBaseClassError method), 475
- \_\_repr\_\_ (fipy.terms.ExplicitVariableError method), 476
- \_\_repr\_\_ (fipy.terms.IncorrectSolutionVariable method), 476
- \_\_repr\_\_ (fipy.terms.SolutionVariableNumberError method), 477
- \_\_repr\_\_ (fipy.terms.SolutionVariableRequiredError method), 477
- \_\_repr\_\_ (fipy.terms.TermMultiplyError method), 478
- \_\_repr\_\_ (fipy.terms.TransientTermError method), 479
- \_\_repr\_\_ (fipy.terms.VectorCoeffError method), 479
- \_\_repr\_\_ (fipy.terms.advectionTerm.AdvectionTerm method), 484
- \_\_repr\_\_ (fipy.terms.cellTerm.CellTerm method), 489
- \_\_repr\_\_ (fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm method), 494
- \_\_repr\_\_ (fipy.terms.diffusionTerm.DiffusionTerm method), 498
- \_\_repr\_\_ (fipy.terms.diffusionTermCorrection.DiffusionTermCorrection method), 501
- \_\_repr\_\_ (fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection method), 505
- \_\_repr\_\_ (fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm method), 508
- \_\_repr\_\_ (fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm method), 513
- \_\_repr\_\_ (fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm method), 518
- \_\_repr\_\_ (fipy.terms.faceTerm.FaceTerm method), 523
- \_\_repr\_\_ (fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm method), 528
- \_\_repr\_\_ (fipy.terms.hybridConvectionTerm.HybridConvectionTerm method), 533
- \_\_repr\_\_ (fipy.terms.implicitSourceTerm.ImplicitSourceTerm method), 537
- \_\_repr\_\_ (fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm method), 542
- \_\_repr\_\_ (fipy.terms.residualTerm.ResidualTerm method), 546
- \_\_repr\_\_ (fipy.terms.sourceTerm.SourceTerm method), 551
- \_\_repr\_\_ (fipy.terms.term.Term method), 555
- \_\_repr\_\_ (fipy.terms.transientTerm.TransientTerm method), 559
- \_\_repr\_\_ (fipy.terms.upwindConvectionTerm.UpwindConvectionTerm method), 564
- \_\_repr\_\_ (fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm method), 569
- \_\_repr\_\_ (fipy.tools.PhysicalField method), 581
- \_\_repr\_\_ (fipy.tools.comms.commWrapper.CommWrapper method), 594
- \_\_repr\_\_ (fipy.tools.comms.dummyComm.DummyComm method), 594
- \_\_repr\_\_ (fipy.tools.dimensions.physicalField.PhysicalField method), 603
- \_\_repr\_\_ (fipy.tools.dimensions.physicalField.PhysicalUnit method), 617
- \_\_repr\_\_ (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 639
- \_\_repr\_\_ (fipy.variables.cellVariable.CellVariable method), 654
- \_\_repr\_\_ (fipy.variables.distanceVariable.DistanceVariable method), 672
- \_\_repr\_\_ (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 689
- \_\_repr\_\_ (fipy.variables.faceVariable.FaceVariable method), 703
- \_\_repr\_\_ (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 716
- \_\_repr\_\_ (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 732
- \_\_repr\_\_ (fipy.variables.histogramVariable.HistogramVariable method), 747
- \_\_repr\_\_ (fipy.variables.meshVariable.MeshVariable method), 761
- \_\_repr\_\_ (fipy.variables.modularVariable.ModularVariable method), 773
- \_\_repr\_\_ (fipy.variables.noiseVariable.NoiseVariable method), 788
- \_\_repr\_\_ (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 802
- \_\_repr\_\_ (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 814

- \_\_repr\_\_ (fipy.variables.surfactantVariable.SurfactantVariable method), 826
- \_\_repr\_\_ (fipy.variables.uniformNoiseVariable.UniformNoiseVariable method), 842
- \_\_repr\_\_ (fipy.variables.variable.Variable method), 856
- \_\_repr\_\_ (fipy.viewers.MeshDimensionError method), 863
- \_\_rmul\_\_ (fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D method), 145
- \_\_rmul\_\_ (fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D method), 153
- \_\_rmul\_\_ (fipy.meshes.gmshMesh.Gmsh2D method), 183
- \_\_rmul\_\_ (fipy.meshes.gmshMesh.Gmsh2DIn3DSpace method), 192
- \_\_rmul\_\_ (fipy.meshes.gmshMesh.Gmsh3D method), 200
- \_\_rmul\_\_ (fipy.meshes.gmshMesh.GmshGrid2D method), 209
- \_\_rmul\_\_ (fipy.meshes.gmshMesh.GmshGrid3D method), 217
- \_\_rmul\_\_ (fipy.meshes.mesh.Mesh method), 227
- \_\_rmul\_\_ (fipy.meshes.mesh1D.Mesh1D method), 236
- \_\_rmul\_\_ (fipy.meshes.mesh2D.Mesh2D method), 244
- \_\_rmul\_\_ (fipy.meshes.nonUniformGrid1D.NonUniformGrid1D method), 253
- \_\_rmul\_\_ (fipy.meshes.nonUniformGrid2D.NonUniformGrid2D method), 261
- \_\_rmul\_\_ (fipy.meshes.nonUniformGrid3D.NonUniformGrid3D method), 270
- \_\_rmul\_\_ (fipy.meshes.periodicGrid1D.PeriodicGrid1D method), 279
- \_\_rmul\_\_ (fipy.meshes.periodicGrid2D.PeriodicGrid2D method), 288
- \_\_rmul\_\_ (fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight method), 296
- \_\_rmul\_\_ (fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom method), 305
- \_\_rmul\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3D method), 315
- \_\_rmul\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack method), 323
- \_\_rmul\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight method), 330
- \_\_rmul\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack method), 338
- \_\_rmul\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom method), 346
- \_\_rmul\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom method), 354
- \_\_rmul\_\_ (fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack method), 361
- \_\_rmul\_\_ (fipy.meshes.skewedGrid2D.SkewedGrid2D method), 370
- \_\_rmul\_\_ (fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D method), 379
- \_\_rmul\_\_ (fipy.meshes.tri2D.Tri2D method), 395
- \_\_rmul\_\_ (fipy.terms.advectionTerm.AdvectionTerm method), 484
- \_\_rmul\_\_ (fipy.terms.cellTerm.CellTerm method), 489
- \_\_rmul\_\_ (fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm method), 494
- \_\_rmul\_\_ (fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm method), 513
- \_\_rmul\_\_ (fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm method), 518
- \_\_rmul\_\_ (fipy.terms.faceTerm.FaceTerm method), 523
- \_\_rmul\_\_ (fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm method), 528
- \_\_rmul\_\_ (fipy.terms.hybridConvectionTerm.HybridConvectionTerm method), 533
- \_\_rmul\_\_ (fipy.terms.implicitSourceTerm.ImplicitSourceTerm method), 537
- \_\_rmul\_\_ (fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm method), 542
- \_\_rmul\_\_ (fipy.terms.residualTerm.ResidualTerm method), 546
- \_\_rmul\_\_ (fipy.terms.sourceTerm.SourceTerm method), 551
- \_\_rmul\_\_ (fipy.terms.transientTerm.TransientTerm method), 559
- \_\_rmul\_\_ (fipy.terms.upwindConvectionTerm.UpwindConvectionTerm method), 565
- \_\_rmul\_\_ (fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm method), 570
- \_\_rmul\_\_ (fipy.tools.physicalField.physicalField method), 581
- \_\_rmul\_\_ (fipy.tools.dimensions.physicalField.physicalField method), 603
- \_\_rmul\_\_ (fipy.tools.dimensions.physicalField.physicalUnit method), 617
- \_\_rtruediv\_\_ (fipy.tools.dimensions.physicalField.physicalUnit method), 617
- \_\_setattr\_\_ (fipy.meshes.abstractMesh.MeshAdditionError method), 139
- \_\_setattr\_\_ (fipy.meshes.gmshMesh.GmshException method), 204
- \_\_setattr\_\_ (fipy.meshes.gmshMesh.MeshExportError method), 222
- \_\_setattr\_\_ (fipy.meshes.mesh.MeshAdditionError method), 231
- \_\_setattr\_\_ (fipy.solvers.SerialSolverError method), 428
- \_\_setattr\_\_ (fipy.solvers.solver.IllConditionedPreconditionerWarning method), 454
- \_\_setattr\_\_ (fipy.solvers.solver.MatrixIllConditionedWarning method), 454
- \_\_setattr\_\_ (fipy.solvers.solver.MaximumIterationWarning method), 455
- \_\_setattr\_\_ (fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning method), 455
- \_\_setattr\_\_ (fipy.solvers.solver.PreconditionerWarning method), 456
- \_\_setattr\_\_ (fipy.solvers.solver.ScalarQuantityOutOfRangeWarning method), 457
- \_\_setattr\_\_ (fipy.solvers.solver.SolverConvergenceWarning method), 457
- \_\_setattr\_\_ (fipy.solvers.solver.StagnatedSolverWarning method), 458
- \_\_setattr\_\_ (fipy.terms.AbstractBaseClassError method), 475
- \_\_setattr\_\_ (fipy.terms.ExplicitVariableError method), 476
- \_\_setattr\_\_ (fipy.terms.IncorrectSolutionVariable method), 476
- \_\_setattr\_\_ (fipy.terms.SolutionVariableNumberError method), 477
- \_\_setattr\_\_ (fipy.terms.SolutionVariableRequiredError method), 478
- \_\_setattr\_\_ (fipy.terms.TermMultiplyError method), 478
- \_\_setattr\_\_ (fipy.terms.TransientTermError method), 479
- \_\_setattr\_\_ (fipy.terms.VectorCoeffError method), 479
- \_\_setattr\_\_ (fipy.viewers.MeshDimensionError method), 863
- \_\_setitem\_\_ (fipy.tools.physicalField.physicalField method), 581
- \_\_setitem\_\_ (fipy.tools.dimensions.physicalField.physicalField method), 603

- `__setstate__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 640
- `__setstate__()` (*fipy.variables.cellVariable.CellVariable* method), 654
- `__setstate__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 672
- `__setstate__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 689
- `__setstate__()` (*fipy.variables.faceVariable.FaceVariable* method), 703
- `__setstate__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 716
- `__setstate__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 732
- `__setstate__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 747
- `__setstate__()` (*fipy.variables.meshVariable.MeshVariable* method), 761
- `__setstate__()` (*fipy.variables.modularVariable.ModularVariable* method), 773
- `__setstate__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 788
- `__setstate__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 802
- `__setstate__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 814
- `__setstate__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 826
- `__setstate__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 842
- `__setstate__()` (*fipy.variables.variable.Variable* method), 856
- `__str__()` (*fipy.meshes.abstractMesh.MeshAdditionError* method), 139
- `__str__()` (*fipy.meshes.gmshMesh.GmshException* method), 204
- `__str__()` (*fipy.meshes.gmshMesh.MeshExportError* method), 222
- `__str__()` (*fipy.meshes.mesh.MeshAdditionError* method), 231
- `__str__()` (*fipy.solvers.SerialSolverError* method), 428
- `__str__()` (*fipy.solvers.solver.IllConditionedPreconditionerWarning* method), 454
- `__str__()` (*fipy.solvers.solver.MatrixIllConditionedWarning* method), 454
- `__str__()` (*fipy.solvers.solver.MaximumIterationWarning* method), 455
- `__str__()` (*fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning* method), 455
- `__str__()` (*fipy.solvers.solver.PreconditionerWarning* method), 456
- `__str__()` (*fipy.solvers.solver.ScalarQuantityOutOfRangeWarning* method), 457
- `__str__()` (*fipy.solvers.solver.SolverConvergenceWarning* method), 458
- `__str__()` (*fipy.solvers.solver.StagnatedSolverWarning* method), 458
- `__str__()` (*fipy.terms.AbstractBaseClassError* method), 475
- `__str__()` (*fipy.terms.ExplicitVariableError* method), 476
- `__str__()` (*fipy.terms.IncorrectSolutionVariable* method), 476
- `__str__()` (*fipy.terms.SolutionVariableNumberError* method), 477
- `__str__()` (*fipy.terms.SolutionVariableRequiredError* method), 478
- `__str__()` (*fipy.terms.TermMultiplyError* method), 478
- `__str__()` (*fipy.terms.TransientTermError* method), 479
- `__str__()` (*fipy.terms.VectorCoeffError* method), 479
- `__str__()` (*fipy.tools.PhysicalField* method), 581
- `__str__()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 603
- `__str__()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 618
- `__str__()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 640
- `__str__()` (*fipy.variables.cellVariable.CellVariable* method), 654
- `__str__()` (*fipy.variables.distanceVariable.DistanceVariable* method), 672
- `__str__()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 689
- `__str__()` (*fipy.variables.faceVariable.FaceVariable* method), 703
- `__str__()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 716
- `__str__()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 732
- `__str__()` (*fipy.variables.histogramVariable.HistogramVariable* method), 747
- `__str__()` (*fipy.variables.meshVariable.MeshVariable* method), 761
- `__str__()` (*fipy.variables.modularVariable.ModularVariable* method), 773
- `__str__()` (*fipy.variables.noiseVariable.NoiseVariable* method), 788
- `__str__()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 802
- `__str__()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 814
- `__str__()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 826
- `__str__()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 842
- `__str__()` (*fipy.variables.variable.Variable* method), 856
- `__str__()` (*fipy.viewers.MeshDimensionError* method), 863
- `__sub__()` (*fipy.meshes.abstractMesh.AbstractMesh* method), 136
- `__sub__()` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 146
- `__sub__()` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 154
- `__sub__()` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* method), 162
- `__sub__()` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* method), 169
- `__sub__()` (*fipy.meshes.gmshMesh.Gmsh2D* method), 184
- `__sub__()` (*fipy.meshes.gmshMesh.Gmsh2DIn3Dspace* method), 192
- `__sub__()` (*fipy.meshes.gmshMesh.Gmsh3D* method), 201
- `__sub__()` (*fipy.meshes.gmshMesh.GmshGrid2D* method), 209
- `__sub__()` (*fipy.meshes.gmshMesh.GmshGrid3D* method), 218
- `__sub__()` (*fipy.meshes.mesh.Mesh* method), 228
- `__sub__()` (*fipy.meshes.mesh1D.Mesh1D* method), 237
- `__sub__()` (*fipy.meshes.mesh2D.Mesh2D* method), 245
- `__sub__()` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 254
- `__sub__()` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 262
- `__sub__()` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 271
- `__sub__()` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 279

- \_\_sub\_\_ () (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 289
- \_\_sub\_\_ () (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 297
- \_\_sub\_\_ () (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 305
- \_\_sub\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 315
- \_\_sub\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 323
- \_\_sub\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 331
- \_\_sub\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 339
- \_\_sub\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 346
- \_\_sub\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 354
- \_\_sub\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 362
- \_\_sub\_\_ () (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 370
- \_\_sub\_\_ () (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 379
- \_\_sub\_\_ () (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* method), 387
- \_\_sub\_\_ () (*fipy.meshes.tri2D.Tri2D* method), 395
- \_\_sub\_\_ () (*fipy.meshes.uniformGrid.UniformGrid* method), 403
- \_\_sub\_\_ () (*fipy.meshes.uniformGrid1D.UniformGrid1D* method), 410
- \_\_sub\_\_ () (*fipy.meshes.uniformGrid2D.UniformGrid2D* method), 417
- \_\_sub\_\_ () (*fipy.meshes.uniformGrid3D.UniformGrid3D* method), 424
- \_\_sub\_\_ () (*fipy.tools.PhysicalField* method), 582
- \_\_sub\_\_ () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 604
- \_\_truediv\_\_ () (*fipy.meshes.abstractMesh.AbstractMesh* method), 136
- \_\_truediv\_\_ () (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* method), 146
- \_\_truediv\_\_ () (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 154
- \_\_truediv\_\_ () (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* method), 162
- \_\_truediv\_\_ () (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* method), 169
- \_\_truediv\_\_ () (*fipy.meshes.gmshMesh.Gmsh2D* method), 184
- \_\_truediv\_\_ () (*fipy.meshes.gmshMesh.Gmsh2DIn3Dspace* method), 192
- \_\_truediv\_\_ () (*fipy.meshes.gmshMesh.Gmsh3D* method), 201
- \_\_truediv\_\_ () (*fipy.meshes.gmshMesh.GmshGrid2D* method), 209
- \_\_truediv\_\_ () (*fipy.meshes.gmshMesh.GmshGrid3D* method), 218
- \_\_truediv\_\_ () (*fipy.meshes.mesh.Mesh* method), 228
- \_\_truediv\_\_ () (*fipy.meshes.mesh1D.Mesh1D* method), 237
- \_\_truediv\_\_ () (*fipy.meshes.mesh2D.Mesh2D* method), 245
- \_\_truediv\_\_ () (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* method), 254
- \_\_truediv\_\_ () (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 262
- \_\_truediv\_\_ () (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* method), 271
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* method), 279
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 289
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 297
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 305
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* method), 315
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* method), 323
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* method), 331
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* method), 339
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* method), 346
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* method), 354
- \_\_truediv\_\_ () (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* method), 362
- \_\_truediv\_\_ () (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 370
- \_\_truediv\_\_ () (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* method), 379
- \_\_truediv\_\_ () (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* method), 387
- \_\_truediv\_\_ () (*fipy.meshes.tri2D.Tri2D* method), 395
- \_\_truediv\_\_ () (*fipy.meshes.uniformGrid.UniformGrid* method), 403
- \_\_truediv\_\_ () (*fipy.meshes.uniformGrid1D.UniformGrid1D* method), 410
- \_\_truediv\_\_ () (*fipy.meshes.uniformGrid2D.UniformGrid2D* method), 417
- \_\_truediv\_\_ () (*fipy.meshes.uniformGrid3D.UniformGrid3D* method), 424
- \_\_truediv\_\_ () (*fipy.tools.PhysicalField* method), 582
- \_\_truediv\_\_ () (*fipy.tools.dimensions.physicalField.PhysicalField* method), 604
- \_\_truediv\_\_ () (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 618
- cache  
command line option, 33
- inline  
command line option, 33
- lsmlib  
command line option, 34
- no-cache  
command line option, 33
- no-pyparse  
command line option, 34
- pyang  
command line option, 34
- pyangx  
command line option, 34
- pyparse  
command line option, 34
- scipy

command line option, 34  
 --skfmm  
 command line option, 34  
 --trilinos  
 command line option, 34

## A

AbstractBaseClassError, 475

AbstractMatplotlib2DViewer (class in *fiPy.viewers.matplotlibViewer.abstractMatplotlib2DViewer*), 867

AbstractMatplotlibViewer (class in *fiPy.viewers.matplotlibViewer.abstractMatplotlibViewer*), 870

AbstractMesh (class in *fiPy.meshes.abstractMesh*), 132

AbstractViewer (class in *fiPy.viewers.viewer*), 906

add() (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 604

add() (*fiPy.tools.PhysicalField* method), 582

add\_note() (*fiPy.meshes.abstractMesh.MeshAdditionError* method), 139

add\_note() (*fiPy.meshes.gmshMesh.GmshException* method), 204

add\_note() (*fiPy.meshes.gmshMesh.MeshExportError* method), 222

add\_note() (*fiPy.meshes.mesh.MeshAdditionError* method), 231

add\_note() (*fiPy.solvers.SerialSolverError* method), 428

add\_note() (*fiPy.solvers.solver.IllConditionedPreconditionerWarning* method), 454

add\_note() (*fiPy.solvers.solver.MatrixIllConditionedWarning* method), 454

add\_note() (*fiPy.solvers.solver.MaximumIterationWarning* method), 455

add\_note() (*fiPy.solvers.solver.PreconditionerNotPositiveDefiniteWarning* method), 456

add\_note() (*fiPy.solvers.solver.PreconditionerWarning* method), 456

add\_note() (*fiPy.solvers.solver.ScalarQuantityOutOfRangeWarning* method), 457

add\_note() (*fiPy.solvers.solver.SolverConvergenceWarning* method), 458

add\_note() (*fiPy.solvers.solver.StagnatedSolverWarning* method), 458

add\_note() (*fiPy.terms.AbstractBaseClassError* method), 475

add\_note() (*fiPy.terms.ExplicitVariableError* method), 476

add\_note() (*fiPy.terms.IncorrectSolutionVariable* method), 476

add\_note() (*fiPy.terms.SolutionVariableNumberError* method), 477

add\_note() (*fiPy.terms.SolutionVariableRequiredError* method), 478

add\_note() (*fiPy.terms.TermMultiplyError* method), 478

add\_note() (*fiPy.terms.TransientTermError* method), 479

add\_note() (*fiPy.terms.VectorCoeffError* method), 479

add\_note() (*fiPy.viewers.MeshDimensionError* method), 863

AdvectionTerm, 1071

AdvectionTerm (class in *fiPy.terms.advectionTerm*), 481

AggregationAMGSolver (class in *fiPy.solvers.pyamgx.aggregationAMGSolver*), 440

all() (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* method), 640

all() (*fiPy.variables.cellVariable.CellVariable* method), 654

all() (*fiPy.variables.distanceVariable.DistanceVariable* method), 672

all() (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 689

all() (*fiPy.variables.faceVariable.FaceVariable* method), 703

all() (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 716

all() (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 732

all() (*fiPy.variables.histogramVariable.HistogramVariable* method), 747

all() (*fiPy.variables.meshVariable.MeshVariable* method), 761

all() (*fiPy.variables.modularVariable.ModularVariable* method), 773

all() (*fiPy.variables.noiseVariable.NoiseVariable* method), 788

all() (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 802

all() (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 814

all() (*fiPy.variables.surfactantVariable.SurfactantVariable* method), 826

all() (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 842

all() (*fiPy.variables.variable.Variable* method), 856

all() (in module *fiPy.tools.numerix*), 625

allclose, 1100, 1146

allclose() (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 604

allclose() (*fiPy.tools.PhysicalField* method), 582

allclose() (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* method), 640

allclose() (*fiPy.variables.cellVariable.CellVariable* method), 654

allclose() (*fiPy.variables.distanceVariable.DistanceVariable* method), 672

allclose() (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 689

allclose() (*fiPy.variables.faceVariable.FaceVariable* method), 703

allclose() (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 716

allclose() (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 732

allclose() (*fiPy.variables.histogramVariable.HistogramVariable* method), 747

allclose() (*fiPy.variables.meshVariable.MeshVariable* method), 762

allclose() (*fiPy.variables.modularVariable.ModularVariable* method), 773

allclose() (*fiPy.variables.noiseVariable.NoiseVariable* method), 788

allclose() (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 803

allclose() (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 814

allclose() (*fiPy.variables.surfactantVariable.SurfactantVariable* method), 826

allclose() (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 842

allclose() (*fiPy.variables.variable.Variable* method), 857

allclose() (in module *fiPy.tools.numerix*), 625

allequal() (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 604

allequal() (*fiPy.tools.PhysicalField* method), 582

allequal() (in module *fiPy.tools.numerix*), 625

allgather()

(*fiPy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper* method), 460

any() (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* method), 640

any() (*fiPy.variables.cellVariable.CellVariable* method), 655

any() (*fiPy.variables.distanceVariable.DistanceVariable* method), 672

any()

(*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 689

any() (*fiPy.variables.faceVariable.FaceVariable* method), 704

- any() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 716
- any() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 733
- any() (*fipy.variables.histogramVariable.HistogramVariable method*), 747
- any() (*fipy.variables.meshVariable.MeshVariable method*), 762
- any() (*fipy.variables.modularVariable.ModularVariable method*), 773
- any() (*fipy.variables.noiseVariable.NoiseVariable method*), 789
- any() (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 803
- any() (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 815
- any() (*fipy.variables.surfactantVariable.SurfactantVariable method*), 826
- any() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 842
- any() (*fipy.variables.variable.Variable method*), 857
- Appveyor, 107
- arccos() (*fipy.tools.dimensions.physicalField.PhysicalField method*), 604
- arccos() (*fipy.tools.PhysicalField method*), 582
- arccosh() (*fipy.tools.dimensions.physicalField.PhysicalField method*), 605
- arccosh() (*fipy.tools.PhysicalField method*), 583
- arcsin() (*fipy.tools.dimensions.physicalField.PhysicalField method*), 605
- arcsin() (*fipy.tools.PhysicalField method*), 583
- arctan, 1094, 1099
- arctan() (*fipy.tools.dimensions.physicalField.PhysicalField method*), 605
- arctan() (*fipy.tools.PhysicalField method*), 583
- arctan2, 1094, 1099
- arctan2() (*fipy.tools.dimensions.physicalField.PhysicalField method*), 605
- arctan2() (*fipy.tools.PhysicalField method*), 583
- arctanh() (*fipy.tools.dimensions.physicalField.PhysicalField method*), 606
- arctanh() (*fipy.tools.PhysicalField method*), 584
- arithmeticFaceValue (*fipy.variables.betaNoiseVariable.BetaNoiseVariable property*), 640
- arithmeticFaceValue (*fipy.variables.cellVariable.CellVariable property*), 655
- arithmeticFaceValue (*fipy.variables.distanceVariable.DistanceVariable property*), 672
- arithmeticFaceValue (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable property*), 689
- arithmeticFaceValue (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable property*), 716
- arithmeticFaceValue (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable property*), 733
- arithmeticFaceValue (*fipy.variables.histogramVariable.HistogramVariable property*), 747
- arithmeticFaceValue (*fipy.variables.modularVariable.ModularVariable property*), 774
- arithmeticFaceValue (*fipy.variables.noiseVariable.NoiseVariable property*), 789
- arithmeticFaceValue (*fipy.variables.surfactantVariable.SurfactantVariable property*), 827
- arithmeticFaceValue (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable property*), 842
- array, 1105, 1114
- aspect2D (*fipy.meshes.abstractMesh.AbstractMesh property*), 136
- aspect2D (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D property*), 146
- aspect2D (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D property*), 154
- aspect2D (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property*), 162
- aspect2D (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property*), 169
- aspect2D (*fipy.meshes.gmshMesh.Gmsh2D property*), 184
- aspect2D (*fipy.meshes.gmshMesh.Gmsh2DIn3Dspace property*), 192
- aspect2D (*fipy.meshes.gmshMesh.Gmsh3D property*), 201
- aspect2D (*fipy.meshes.gmshMesh.GmshGrid2D property*), 210
- aspect2D (*fipy.meshes.gmshMesh.GmshGrid3D property*), 218
- aspect2D (*fipy.meshes.mesh.Mesh property*), 228
- aspect2D (*fipy.meshes.mesh1D.Mesh1D property*), 237
- aspect2D (*fipy.meshes.mesh2D.Mesh2D property*), 245
- aspect2D (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D property*), 254
- aspect2D (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D property*), 262
- aspect2D (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D property*), 271
- aspect2D (*fipy.meshes.periodicGrid1D.PeriodicGrid1D property*), 279
- aspect2D (*fipy.meshes.periodicGrid2D.PeriodicGrid2D property*), 289
- aspect2D (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight property*), 297
- aspect2D (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom property*), 306
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3D property*), 315
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property*), 323
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property*), 331
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property*), 339
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property*), 347
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property*), 354
- aspect2D (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property*), 362
- aspect2D (*fipy.meshes.skewedGrid2D.SkewedGrid2D property*), 370
- aspect2D (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property*), 380
- aspect2D (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property*), 387
- aspect2D (*fipy.meshes.tri2D.Tri2D property*), 396
- aspect2D (*fipy.meshes.uniformGrid.UniformGrid property*), 403
- aspect2D (*fipy.meshes.uniformGrid1D.UniformGrid1D property*), 410
- aspect2D (*fipy.meshes.uniformGrid2D.UniformGrid2D property*), 417
- aspect2D (*fipy.meshes.uniformGrid3D.UniformGrid3D property*), 424
- axes (*examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer*

	<i>property</i> ), 1080	<code>cacheMatrix()</code> ( <i>fipy.terms.hybridConvectionTerm.HybridConvectionTerm</i> <i>method</i> ), 537
axes	( <i>fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer</i> <i>property</i> ), 868	<code>cacheMatrix()</code> ( <i>fipy.terms.implicitSourceTerm.ImplicitSourceTerm</i> <i>method</i> ), 537
axes	( <i>fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer</i> <i>property</i> ), 871	<code>cacheMatrix()</code> ( <i>fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm</i> <i>method</i> ), 543
axes	( <i>fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer</i> <i>property</i> ), 873	<code>cacheMatrix()</code> ( <i>fipy.terms.residualTerm.ResidualTerm</i> <i>method</i> ), 547
axes	( <i>fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer</i> <i>property</i> ), 876	<code>cacheMatrix()</code> ( <i>fipy.terms.sourceTerm.SourceTerm</i> <i>method</i> ), 551
axes	( <i>fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer</i> <i>property</i> ), 879	<code>cacheMatrix()</code> ( <i>fipy.terms.term.Term</i> <i>method</i> ), 555
axes	( <i>fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer</i> <i>property</i> ), 882	<code>cacheMatrix()</code> ( <i>fipy.terms.transientTerm.TransientTerm</i> <i>method</i> ), 560
axes	( <i>fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer</i> <i>property</i> ), 885	<code>cacheMatrix()</code> ( <i>fipy.terms.upwindConvectionTerm.UpwindConvectionTerm</i> <i>method</i> ), 565
axes	( <i>fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer</i> <i>property</i> ), 889	<code>cacheMatrix()</code> ( <i>fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm</i> <i>method</i> ), 570
axes	( <i>fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer</i> <i>property</i> ), 893	<code>cacheRHSvector</code> , 1052
Azure, 107		<code>cacheRHSvector()</code> ( <i>fipy.terms.advectionTerm.AdvectionTerm</i> <i>method</i> ), 485
<b>B</b>		<code>cacheRHSvector()</code> ( <i>fipy.terms.cellTerm.CellTerm</i> <i>method</i> ), 489
Base (class in package.subpackage.base), 122		<code>cacheRHSvector()</code> ( <i>fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm</i> <i>method</i> ), 494
BetaNoiseVariable (class in <i>fipy.variables.betaNoiseVariable</i> ), 633		<code>cacheRHSvector()</code> ( <i>fipy.terms.diffusionTerm.DiffusionTerm</i> <i>method</i> ), 498
BoundaryCondition (class in <i>fipy.boundaryConditions.boundaryCondition</i> ), 127		<code>cacheRHSvector()</code> ( <i>fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm</i> <i>method</i> ), 508
Buildbot, 107		<code>cacheRHSvector()</code> ( <i>fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm</i> <i>method</i> ), 513
<b>C</b>		<code>cacheRHSvector()</code> ( <i>fipy.terms.diffusionTermCorrection.DiffusionTermCorrection</i> <i>method</i> ), 501
<code>cacheMatrix</code> , 1052		<code>cacheRHSvector()</code> ( <i>fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection</i> <i>method</i> ), 505
<code>cacheMatrix()</code> ( <i>fipy.terms.advectionTerm.AdvectionTerm</i> <i>method</i> ), 485		<code>cacheRHSvector()</code> ( <i>fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm</i> <i>method</i> ), 518
<code>cacheMatrix()</code> ( <i>fipy.terms.cellTerm.CellTerm</i> <i>method</i> ), 489		<code>cacheRHSvector()</code> ( <i>fipy.terms.faceTerm.FaceTerm</i> <i>method</i> ), 523
<code>cacheMatrix()</code> ( <i>fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm</i> <i>method</i> ), 494		<code>cacheRHSvector()</code> ( <i>fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm</i> <i>method</i> ), 528
<code>cacheMatrix()</code> ( <i>fipy.terms.diffusionTerm.DiffusionTerm</i> <i>method</i> ), 498		<code>cacheRHSvector()</code> ( <i>fipy.terms.hybridConvectionTerm.HybridConvectionTerm</i> <i>method</i> ), 533
<code>cacheMatrix()</code> ( <i>fipy.terms.diffusionTermCorrection.DiffusionTermCorrection</i> <i>method</i> ), 501		<code>cacheRHSvector()</code> ( <i>fipy.terms.implicitSourceTerm.ImplicitSourceTerm</i> <i>method</i> ), 537
<code>cacheMatrix()</code> ( <i>fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection</i> <i>method</i> ), 505		<code>cacheRHSvector()</code> ( <i>fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm</i> <i>method</i> ), 543
<code>cacheMatrix()</code> ( <i>fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm</i> <i>method</i> ), 508		<code>cacheRHSvector()</code> ( <i>fipy.terms.residualTerm.ResidualTerm</i> <i>method</i> ), 547
<code>cacheMatrix()</code> ( <i>fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm</i> <i>method</i> ), 513		<code>cacheRHSvector()</code> ( <i>fipy.terms.sourceTerm.SourceTerm</i> <i>method</i> ), 551
<code>cacheMatrix()</code> ( <i>fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm</i> <i>method</i> ), 518		<code>cacheRHSvector()</code> ( <i>fipy.terms.term.Term</i> <i>method</i> ), 555
<code>cacheMatrix()</code> ( <i>fipy.terms.faceTerm.FaceTerm</i> <i>method</i> ), 523		<code>cacheRHSvector()</code> ( <i>fipy.terms.transientTerm.TransientTerm</i> <i>method</i> ), 560
<code>cacheMatrix()</code> ( <i>fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm</i> <i>method</i> ), 528		<code>cacheRHSvector()</code> ( <i>fipy.terms.upwindConvectionTerm.UpwindConvectionTerm</i> <i>method</i> ), 565
		<code>cacheRHSvector()</code> ( <i>fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm</i> <i>method</i> ), 570
		<code>calcDistanceFunction()</code>

- `(fipy.variables.distanceVariable.DistanceVariable method)`, 673
- `ceil()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 606
- `ceil()` (*fipy.tools.physicalField.PhysicalField method*), 584
- `cellCenters` (*fipy.meshes.abstractMesh.AbstractMesh property*), 136
- `cellCenters` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D property*), 146
- `cellCenters` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D property*), 154
- `cellCenters` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property*), 162
- `cellCenters` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property*), 169
- `cellCenters` (*fipy.meshes.gmshMesh.Gmsh2D property*), 184
- `cellCenters` (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property*), 193
- `cellCenters` (*fipy.meshes.gmshMesh.Gmsh3D property*), 201
- `cellCenters` (*fipy.meshes.gmshMesh.GmshGrid2D property*), 210
- `cellCenters` (*fipy.meshes.gmshMesh.GmshGrid3D property*), 218
- `cellCenters` (*fipy.meshes.mesh.Mesh property*), 228
- `cellCenters` (*fipy.meshes.mesh1D.Mesh1D property*), 237
- `cellCenters` (*fipy.meshes.mesh2D.Mesh2D property*), 245
- `cellCenters` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D property*), 254
- `cellCenters` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D property*), 262
- `cellCenters` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D property*), 271
- `cellCenters` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D property*), 280
- `cellCenters` (*fipy.meshes.periodicGrid2D.PeriodicGrid2D property*), 289
- `cellCenters` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight property*), 297
- `cellCenters` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom property*), 306
- `cellCenters` (*fipy.meshes.periodicGrid3D.PeriodicGrid3D property*), 316
- `cellCenters` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property*), 323
- `cellCenters` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property*), 331
- `cellCenters` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property*), 339
- `cellCenters` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property*), 347
- `cellCenters` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property*), 354
- `cellCenters` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property*), 362
- `cellCenters` (*fipy.meshes.skewedGrid2D.SkewedGrid2D property*), 371
- `cellCenters` (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property*), 380
- `cellCenters` (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property*), 387
- `cellCenters` (*fipy.meshes.tri2D.Tri2D property*), 396
- `cellCenters` (*fipy.meshes.uniformGrid.UniformGrid property*), 403
- `cellCenters` (*fipy.meshes.uniformGrid1D.UniformGrid1D property*), 410
- `cellCenters` (*fipy.meshes.uniformGrid2D.UniformGrid2D property*), 417
- `cellCenters` (*fipy.meshes.uniformGrid3D.UniformGrid3D property*), 424
- `cellFaceIDs` (*fipy.meshes.abstractMesh.AbstractMesh property*), 136
- `cellFaceIDs` (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D property*), 146
- `cellFaceIDs` (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D property*), 154
- `cellFaceIDs` (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property*), 162
- `cellFaceIDs` (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property*), 169
- `cellFaceIDs` (*fipy.meshes.gmshMesh.Gmsh2D property*), 184
- `cellFaceIDs` (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property*), 193
- `cellFaceIDs` (*fipy.meshes.gmshMesh.Gmsh3D property*), 201
- `cellFaceIDs` (*fipy.meshes.gmshMesh.GmshGrid2D property*), 210
- `cellFaceIDs` (*fipy.meshes.gmshMesh.GmshGrid3D property*), 218
- `cellFaceIDs` (*fipy.meshes.mesh.Mesh property*), 228
- `cellFaceIDs` (*fipy.meshes.mesh1D.Mesh1D property*), 237
- `cellFaceIDs` (*fipy.meshes.mesh2D.Mesh2D property*), 245
- `cellFaceIDs` (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D property*), 254
- `cellFaceIDs` (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D property*), 262
- `cellFaceIDs` (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D property*), 271
- `cellFaceIDs` (*fipy.meshes.periodicGrid1D.PeriodicGrid1D property*), 280
- `cellFaceIDs` (*fipy.meshes.periodicGrid2D.PeriodicGrid2D property*), 289
- `cellFaceIDs` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight property*), 297
- `cellFaceIDs` (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom property*), 306
- `cellFaceIDs` (*fipy.meshes.periodicGrid3D.PeriodicGrid3D property*), 316
- `cellFaceIDs` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property*), 323
- `cellFaceIDs` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property*), 331
- `cellFaceIDs` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property*), 339
- `cellFaceIDs` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property*), 347
- `cellFaceIDs` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property*), 354
- `cellFaceIDs` (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property*), 362
- `cellFaceIDs` (*fipy.meshes.skewedGrid2D.SkewedGrid2D property*), 371
- `cellFaceIDs` (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property*), 380
- `cellFaceIDs` (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property*), 387
- `cellFaceIDs` (*fipy.meshes.tri2D.Tri2D property*), 396



cellFaceIDs (*fipy.meshes.uniformGrid.UniformGrid* property), 403  
cellFaceIDs (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 410  
cellFaceIDs (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 417  
cellFaceIDs (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 424  
cellInterfaceAreas (*fipy.variables.distanceVariable.DistanceVariable* property), 673  
CellTerm (class in *fipy.terms.cellTerm*), 488  
CellVariable, 931, 947, 1010, 1051, 1070, 1098, 1100, 1109, 1120, 1124, 1141, 1147, 1168, 1170  
CellVariable (class in *fipy.variables.cellVariable*), 649  
cellVolumeAverage (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 641  
cellVolumeAverage (*fipy.variables.cellVariable.CellVariable* property), 655  
cellVolumeAverage (*fipy.variables.distanceVariable.DistanceVariable* property), 674  
cellVolumeAverage (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 690  
cellVolumeAverage (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 717  
cellVolumeAverage (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 733  
cellVolumeAverage (*fipy.variables.histogramVariable.HistogramVariable* property), 748  
cellVolumeAverage (*fipy.variables.modularVariable.ModularVariable* property), 774  
cellVolumeAverage (*fipy.variables.noiseVariable.NoiseVariable* property), 789  
cellVolumeAverage (*fipy.variables.surfactantVariable.SurfactantVariable* property), 827  
cellVolumeAverage (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 843  
CentralDifferenceConvectionTerm (class in *fipy.terms.centralDiffConvectionTerm*), 492  
CircleCI, 107  
ClassicalAMGSolver (class in *fipy.solvers.pyamgx.classicalAMGSolver*), 440  
clock\_ns() (*fipy.tools.timer.Timer* static method), 630  
cmap (examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer property), 1080  
cmap (*fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* property), 868  
cmap (*fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* property), 871  
cmap (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 873  
cmap (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* property), 876  
cmap (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* property), 879  
cmap (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* property), 882  
cmap (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* property), 885  
cmap (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 889  
cmap (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 889  
cmap (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* property), 893  
collection (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* property), 885  
colorbar (examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer property), 1080  
colorbar (*fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* property), 869  
colorbar (*fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* property), 871  
colorbar (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 873  
colorbar (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* property), 876  
colorbar (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* property), 879  
colorbar (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* property), 882  
colorbar (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* property), 885  
colorbar (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 889  
colorbar (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* property), 893  
command line option  
--cache, 33  
--inline, 33  
--lsmlib, 34  
--no-cache, 33  
--no-pysparse, 34  
--pyamgx, 34  
--pysparse, 34  
--skfmm, 34  
--trilinos, 34  
comm (multiprocessors in *fipy.tools.comms.comnWrapper*), 594  
conda, 107  
conda\_info() (in module *fipy.tools.logging.environment*), 622  
conjugate() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 606  
conjugate() (*fipy.tools.PhysicalField* method), 584  
conjugate() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 641  
constrain() (*fipy.variables.cellVariable.CellVariable* method), 656

- `constrain()` (*fipy.variables.distanceVariable.DistanceVariable* method), 674
  - `constrain()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 690
  - `constrain()` (*fipy.variables.faceVariable.FaceVariable* method), 704
  - `constrain()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 717
  - `constrain()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 734
  - `constrain()` (*fipy.variables.histogramVariable.HistogramVariable* method), 748
  - `constrain()` (*fipy.variables.meshVariable.MeshVariable* method), 762
  - `constrain()` (*fipy.variables.modularVariable.ModularVariable* method), 774
  - `constrain()` (*fipy.variables.noiseVariable.NoiseVariable* method), 790
  - `constrain()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 803
  - `constrain()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 815
  - `constrain()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 827
  - `constrain()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 843
  - `constrain()` (*fipy.variables.variable.Variable* method), 857
  - `Constraint` (class in *fipy.boundaryConditions.constraint*), 127
  - `constraintMask` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 642
  - `constraintMask` (*fipy.variables.cellVariable.CellVariable* property), 657
  - `constraintMask` (*fipy.variables.distanceVariable.DistanceVariable* property), 675
  - `constraintMask` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 691
  - `constraintMask` (*fipy.variables.faceVariable.FaceVariable* property), 705
  - `constraintMask` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 718
  - `constraintMask` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 734
  - `constraintMask` (*fipy.variables.histogramVariable.HistogramVariable* property), 749
  - `constraintMask` (*fipy.variables.meshVariable.MeshVariable* property), 763
  - `constraintMask` (*fipy.variables.modularVariable.ModularVariable* property), 775
  - `constraintMask` (*fipy.variables.noiseVariable.NoiseVariable* property), 790
  - `constraintMask` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 804
  - `constraintMask` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 816
  - `constraintMask` (*fipy.variables.surfactantVariable.SurfactantVariable* property), 828
  - `constraintMask` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 844
  - `constraintMask` (*fipy.variables.variable.Variable* property), 858
  - `Continuous Integration`, 107
  - `conversionFactorTo()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 618
  - `conversionTupleTo()` (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 619
  - `convertToUnit()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 606
  - `convertToUnit()` (*fipy.tools.PhysicalField* method), 584
  - `copy()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 606
  - `copy()` (*fipy.tools.PhysicalField* method), 584
  - `copy()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 642
  - `copy()` (*fipy.variables.cellVariable.CellVariable* method), 657
  - `copy()` (*fipy.variables.distanceVariable.DistanceVariable* method), 676
  - `copy()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 692
  - `copy()` (*fipy.variables.faceVariable.FaceVariable* method), 705
  - `copy()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 719
  - `copy()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 735
  - `copy()` (*fipy.variables.histogramVariable.HistogramVariable* method), 750
  - `copy()` (*fipy.variables.meshVariable.MeshVariable* method), 763
  - `copy()` (*fipy.variables.modularVariable.ModularVariable* method), 775
  - `copy()` (*fipy.variables.noiseVariable.NoiseVariable* method), 791
  - `copy()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 804
  - `copy()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 816
  - `copy()` (*fipy.variables.surfactantVariable.SurfactantVariable* method), 829
  - `copy()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 845
  - `copy()` (*fipy.variables.variable.Variable* method), 858
  - `cos()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 607
  - `cos()` (*fipy.tools.PhysicalField* method), 585
  - `cosh()` (*fipy.tools.dimensions.physicalField.PhysicalField* method), 607
  - `cosh()` (*fipy.tools.PhysicalField* method), 585
  - `CylindricalGrid1D()` (in module *fipy.meshes.factoryMeshes*), 172
  - `CylindricalGrid2D()` (in module *fipy.meshes.factoryMeshes*), 172
  - `CylindricalNonUniformGrid1D` (class in *fipy.meshes.cylindricalNonUniformGrid1D*), 140
  - `CylindricalNonUniformGrid2D` (class in *fipy.meshes.cylindricalNonUniformGrid2D*), 149
  - `CylindricalUniformGrid1D` (class in *fipy.meshes.cylindricalUniformGrid1D*), 158
  - `CylindricalUniformGrid2D` (class in *fipy.meshes.cylindricalUniformGrid2D*), 165
- ## D
- `DefaultAsymmetricSolver`, 947, 1145
  - `DefaultAsymmetricSolver` (in module *fipy.solvers*), 427
  - `DefaultSolver`, 931
  - `DefaultSolver` (in module *fipy.solvers*), 427
  - `deprecate()` (in module *fipy.tools.decorators*), 595

- DeprecationErroringTestProgram (class in *fipy.tests.test*), 575  
 DiffusionTerm (class in *fipy.terms.diffusionTerm*), 497  
 DiffusionTermCorrection (class in *fipy.terms.diffusionTermCorrection*), 501  
 DiffusionTermNoCorrection (class in *fipy.terms.diffusionTermNoCorrection*), 504  
 DistanceVariable, 1069  
 DistanceVariable (class in *fipy.variables.distanceVariable*), 664  
 divergence (*fipy.variables.faceVariable.FaceVariable* property), 705  
 divergence  
   (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 805  
 divergence  
   (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 817  
 divide() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 607  
 divide() (*fipy.tools.physicalField.PhysicalField* method), 585  
 doctest\_raw\_input() (in module *fipy*), 125  
 DomDecompPreconditioner (class in *fipy.solvers.trilinos.preconditioners.domDecompPreconditioner*), 464  
 dot() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 608  
 dot() (*fipy.tools.physicalField.PhysicalField* method), 586  
 dot() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 643  
 dot() (*fipy.variables.cellVariable.CellVariable* method), 658  
 dot() (*fipy.variables.distanceVariable.DistanceVariable* method), 676  
 dot()  
   (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 692  
 dot() (*fipy.variables.faceVariable.FaceVariable* method), 706  
 dot() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 719  
 dot() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 735  
 dot() (*fipy.variables.histogramVariable.HistogramVariable* method), 750  
 dot() (*fipy.variables.meshVariable.MeshVariable* method), 764  
 dot() (*fipy.variables.modularVariable.ModularVariable* method), 776  
 dot() (*fipy.variables.noiseVariable.NoiseVariable* method), 791  
 dot()  
   (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 805  
 dot()  
   (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 817  
 dot() (*fipy.variables.surfactantVariable.SurfactantVariable* method), 829  
 dot() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 845  
 dot() (in module *fipy.tools.numerix*), 625  
 DummyComm (class in *fipy.tools.comms.dummyComm*), 594  
 DummySolver (class in *fipy.solvers.petsc.dummySolver*), 431  
 DummySolver (in module *fipy.solvers*), 428  
 DummyViewer (class in *fipy.viewers*), 861
- ## E
- elapsed (*fipy.tools.timer.Timer* property), 630  
 environment variable  
   FIPY\_CACHE, 35  
   FIPY\_DISPLAY\_MATRIX, 34  
   FIPY\_INCLUDE\_NUMERIX\_ALL, 35, 1176  
   FIPY\_INLINE, 34  
   FIPY\_INLINE\_COMMENT, 34  
   FIPY\_LOG\_CONFIG, 32, 34, 35  
   FIPY\_SOLVERS, 25, 34  
   FIPY\_VERBOSE\_SOLVER, 34  
   FIPY\_VIEWER, 35  
   PETSC\_OPTIONS, 35  
 EpetraCommWrapper (class in *fipy.solvers.trilinos.comms.epetraCommWrapper*), 459  
 error() (in module *fipy.steppers*), 472  
 examples  
   module, 919  
   examples.benchmarking  
     module, 920  
     examples.benchmarking.benchmarker  
       module, 920  
     examples.benchmarking.size  
       module, 920  
     examples.benchmarking.steps  
       module, 920  
     examples.benchmarking.utils  
       module, 920  
     examples.benchmarking.versions  
       module, 920  
     examples.cahnHilliard  
       module, 920  
     examples.cahnHilliard.mesh2D  
       module, 921  
     examples.cahnHilliard.mesh2DCoupled  
       module, 923  
     examples.cahnHilliard.mesh3D  
       module, 926  
     examples.cahnHilliard.sphere  
       module, 928  
     examples.cahnHilliard.sphereDaemon  
       module, 930  
     examples.cahnHilliard.tanh1D  
       module, 930  
     examples.cahnHilliard.test  
       module, 932  
     examples.chemotaxis  
       module, 932  
     examples.chemotaxis.input  
       module, 933  
     examples.chemotaxis.input2D  
       module, 935  
     examples.chemotaxis.parameters  
       module, 938  
     examples.chemotaxis.test  
       module, 938  
     examples.convection  
       module, 938  
     examples.convection.advection  
       module, 938  
     examples.convection.advection.explicitUpwind  
       module, 939  
     examples.convection.advection.implicitUpwind  
       module, 939  
     examples.convection.advection.vanLeerUpwind  
       module, 939  
     examples.convection.exponential1D  
       module, 939  
     examples.convection.exponential1D.cylindricalMesh1D  
       module, 940  
     examples.convection.exponential1D.cylindricalMesh1DNonUniform  
       module, 941  
     examples.convection.exponential1D.mesh1D  
       module, 943  
     examples.convection.exponential1D.tri2D  
       module, 944  
     examples.convection.exponential1DBack  
       module, 945

examples.convection.exponential1DBack.mesh1D  
     module, 945  
 examples.convection.exponential1DSource  
     module, 946  
 examples.convection.exponential1DSource.mesh1D  
     module, 947  
 examples.convection.exponential1DSource.tri2D  
     module, 948  
 examples.convection.exponential2D  
     module, 949  
 examples.convection.exponential2D.cylindricalMesh2D  
     module, 949  
 examples.convection.exponential2D.cylindricalMesh2DNonUniform  
     module, 951  
 examples.convection.exponential2D.mesh2D  
     module, 952  
 examples.convection.exponential2D.tri2D  
     module, 953  
 examples.convection.pecllet  
     module, 954  
 examples.convection.powerLaw1D  
     module, 955  
 examples.convection.powerLaw1D.mesh1D  
     module, 956  
 examples.convection.powerLaw1D.tri2D  
     module, 957  
 examples.convection.robin  
     module, 958  
 examples.convection.source  
     module, 960  
 examples.convection.test  
     module, 961  
 examples.diffusion.anisotropy  
     module, 961  
 examples.diffusion.circle  
     module, 963  
 examples.diffusion.circleQuad  
     module, 968  
 examples.diffusion.coupled  
     module, 973  
 examples.diffusion.electrostatics  
     module, 975  
 examples.diffusion.explicit  
     module, 979  
 examples.diffusion.explicit.mesh1D  
     module, 980  
 examples.diffusion.explicit.mixedelement  
     module, 981  
 examples.diffusion.explicit.test  
     module, 982  
 examples.diffusion.explicit.tri2D  
     module, 982  
 examples.diffusion.mesh1D  
     module, 983  
 examples.diffusion.mesh20x20  
     module, 1003  
 examples.diffusion.mesh20x20Coupled  
     module, 1006  
 examples.diffusion.nthOrder  
     module, 1009  
 examples.diffusion.nthOrder.input4thOrder1D  
     module, 1010  
 examples.diffusion.nthOrder.input4thOrder\_line  
     module, 1012  
 examples.diffusion.nthOrder.test  
     module, 1012  
 examples.diffusion.steadyState  
     module, 1012  
 examples.diffusion.steadyState.mesh1D  
     module, 1012  
 examples.diffusion.steadyState.mesh1D.inputPeriodic  
     module, 1013  
 examples.diffusion.steadyState.mesh1D.tri2Dinput  
     module, 1013  
 examples.diffusion.steadyState.mesh20x20  
     module, 1014  
 examples.diffusion.steadyState.mesh20x20.gmshinput  
     module, 1014  
 examples.diffusion.steadyState.mesh20x20.isotropy  
     module, 1014  
 examples.diffusion.steadyState.mesh20x20.modifiedMeshInput  
     module, 1015  
 examples.diffusion.steadyState.mesh20x20.orthoerror  
     module, 1016  
 examples.diffusion.steadyState.mesh20x20.tri2Dinput  
     module, 1017  
 examples.diffusion.steadyState.mesh50x50  
     module, 1017  
 examples.diffusion.steadyState.mesh50x50.input  
     module, 1017  
 examples.diffusion.steadyState.mesh50x50.tri2Dinput  
     module, 1017  
 examples.diffusion.steadyState.otherMeshes  
     module, 1018  
 examples.diffusion.steadyState.otherMeshes.cubicalProblem  
     module, 1018  
 examples.diffusion.steadyState.otherMeshes.grid3Dinput  
     module, 1018  
 examples.diffusion.steadyState.otherMeshes.prism  
     module, 1018  
 examples.diffusion.steadyState.test  
     module, 1019  
 examples.diffusion.test  
     module, 1019  
 examples.diffusion.variable  
     module, 1019  
 examples.elphf  
     module, 1020  
 examples.elphf.diffusion  
     module, 1021  
 examples.elphf.diffusion.mesh1D  
     module, 1021  
 examples.elphf.diffusion.mesh1Ddimensional  
     module, 1024  
 examples.elphf.diffusion.mesh2D  
     module, 1026  
 examples.elphf.input  
     module, 1029  
 examples.elphf.phase  
     module, 1035  
 examples.elphf.phaseDiffusion  
     module, 1038  
 examples.elphf.poisson  
     module, 1046  
 examples.elphf.test  
     module, 1049  
 examples.flow  
     module, 1049  
 examples.flow.stokesCavity  
     module, 1050  
 examples.flow.test  
     module, 1054  
 examples.levelSet  
     module, 1054

examples.levelSet.advection  
     module, 1055  
 examples.levelSet.advection.circle  
     module, 1055  
 examples.levelSet.advection.mesh1D  
     module, 1057  
 examples.levelSet.advection.test  
     module, 1058  
 examples.levelSet.advection.trench  
     module, 1058  
 examples.levelSet.distanceFunction  
     module, 1060  
 examples.levelSet.distanceFunction.circle  
     module, 1060  
 examples.levelSet.distanceFunction.interior  
     module, 1061  
 examples.levelSet.distanceFunction.mesh1D  
     module, 1062  
 examples.levelSet.distanceFunction.square  
     module, 1063  
 examples.levelSet.distanceFunction.test  
     module, 1064  
 examples.levelSet.electroChem  
     module, 1064  
 examples.levelSet.electroChem.adsorbingSurfactantEquation  
     module, 1064  
 examples.levelSet.electroChem.adsorption  
     module, 1064  
 examples.levelSet.electroChem.gapFillDistanceVariable  
     module, 1065  
 examples.levelSet.electroChem.gapFillMesh  
     module, 1065  
 examples.levelSet.electroChem.gold  
     module, 1066  
 examples.levelSet.electroChem.howToWriteAScript  
     module, 1067  
 examples.levelSet.electroChem.leveler  
     module, 1073  
 examples.levelSet.electroChem.lines  
     module, 1077  
 examples.levelSet.electroChem.matplotlibSurfactantViewer  
     module, 1077  
 examples.levelSet.electroChem.mayaviSurfactantViewer  
     module, 1081  
 examples.levelSet.electroChem.metalIonDiffusionEquation  
     module, 1084  
 examples.levelSet.electroChem.simpleTrenchSystem  
     module, 1084  
 examples.levelSet.electroChem.surfactantBulkDiffusionEquation  
     module, 1087  
 examples.levelSet.electroChem.test  
     module, 1087  
 examples.levelSet.electroChem.trenchMesh  
     module, 1087  
 examples.levelSet.surfactant  
     module, 1087  
 examples.levelSet.surfactant.circle  
     module, 1088  
 examples.levelSet.surfactant.expandingCircle  
     module, 1088  
 examples.levelSet.surfactant.square  
     module, 1089  
 examples.levelSet.surfactant.test  
     module, 1090  
 examples.levelSet.test  
     module, 1090  
 examples.meshing  
     module, 1090  
 examples.meshing.gmshRefinement  
     module, 1090  
 examples.meshing.inputGrid2D  
     module, 1090  
 examples.meshing.sphere  
     module, 1091  
 examples.meshing.test  
     module, 1092  
 examples.parallel  
     module, 1092  
 examples.phase  
     module, 1092  
 examples.phase.anisotropy  
     module, 1093  
 examples.phase.anisotropyOLD  
     module, 1097  
 examples.phase.binary  
     module, 1100  
 examples.phase.binaryCoupled  
     module, 1109  
 examples.phase.impingement  
     module, 1119  
 examples.phase.impingement.mesh20x20  
     module, 1119  
 examples.phase.impingement.mesh40x1  
     module, 1123  
 examples.phase.impingement.test  
     module, 1126  
 examples.phase.missOrientation  
     module, 1126  
 examples.phase.missOrientation.circle  
     module, 1126  
 examples.phase.missOrientation.mesh1D  
     module, 1127  
 examples.phase.missOrientation.modCircle  
     module, 1128  
 examples.phase.missOrientation.test  
     module, 1128  
 examples.phase.polyxtal  
     module, 1128  
 examples.phase.polyxtalCoupled  
     module, 1134  
 examples.phase.quaternary  
     module, 1140  
 examples.phase.simple  
     module, 1147  
 examples.phase.symmetry  
     module, 1155  
 examples.phase.test  
     module, 1157  
 examples.reactiveWetting  
     module, 1157  
 examples.reactiveWetting.liquidVapor1D  
     module, 1157  
 examples.reactiveWetting.liquidVapor2D  
     module, 1162  
 examples.reactiveWetting.test  
     module, 1166  
 examples.riemann  
     module, 1166  
 examples.riemann.acoustics  
     module, 1166  
 examples.riemann.test  
     module, 1167  
 examples.test  
     module, 1167  
 examples.updating  
     module, 1167

examples.updating.update0\_1to1\_0 module, 1167

examples.updating.update1\_0to2\_0 module, 1171

examples.updating.update2\_0to3\_0 module, 1175

execButNoTest() (in module *fipy.tests.doctestPlus*), 573

exp, 932, 940, 942, 944, 947, 950, 952, 1068, 1105, 1114, 1121, 1125

ExplicitDiffusionTerm, 984, 1099, 1120, 1124

ExplicitDiffusionTerm (class in *fipy.terms.explicitDiffusionTerm*), 508

ExplicitUpwindConvectionTerm (class in *fipy.terms.explicitUpwindConvectionTerm*), 511

ExplicitVariableError, 475

ExponentialConvectionTerm, 1168

ExponentialConvectionTerm (class in *fipy.terms.exponentialConvectionTerm*), 516

ExponentialNoiseVariable (class in *fipy.variables.exponentialNoiseVariable*), 683

extendVariable() (*fipy.variables.distanceVariable.DistanceVariable* method), 676

exteriorFaces (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 162

exteriorFaces (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 387

exteriorFaces (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 410

extrude() (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* method), 154

extrude() (*fipy.meshes.gmshMesh.Gmsh2D* method), 184

extrude() (*fipy.meshes.gmshMesh.Gmsh2DIn3Dspace* method), 193

extrude() (*fipy.meshes.gmshMesh.GmshGrid2D* method), 210

extrude() (*fipy.meshes.mesh2D.Mesh2D* method), 245

extrude() (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* method), 262

extrude() (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* method), 289

extrude() (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* method), 297

extrude() (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* method), 306

extrude() (*fipy.meshes.skewedGrid2D.SkewedGrid2D* method), 371

extrude() (*fipy.meshes.tri2D.Tri2D* method), 396

## F

faceGrad (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 643

faceGrad (*fipy.variables.cellVariable.CellVariable* property), 658

faceGrad (*fipy.variables.distanceVariable.DistanceVariable* property), 677

faceGrad (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 692

faceGrad (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 719

faceGrad (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 735

faceGrad (*fipy.variables.histogramVariable.HistogramVariable* property), 750

faceGrad (*fipy.variables.modularVariable.ModularVariable* property), 776

faceGrad (*fipy.variables.noiseVariable.NoiseVariable* property), 791

faceGrad (*fipy.variables.surfactantVariable.SurfactantVariable* property), 829

faceGrad (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 845

faceGradAverage (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 643

faceGradAverage (*fipy.variables.cellVariable.CellVariable* property), 658

faceGradAverage (*fipy.variables.distanceVariable.DistanceVariable* property), 677

faceGradAverage (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 692

faceGradAverage (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 719

faceGradAverage (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 735

faceGradAverage (*fipy.variables.histogramVariable.HistogramVariable* property), 750

faceGradAverage (*fipy.variables.modularVariable.ModularVariable* property), 776

faceGradAverage (*fipy.variables.noiseVariable.NoiseVariable* property), 791

faceGradAverage (*fipy.variables.surfactantVariable.SurfactantVariable* property), 829

faceGradAverage (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 845

faceGradMod (*fipy.variables.modularVariable.ModularVariable* property), 776

facesBack (*fipy.meshes.abstractMesh.AbstractMesh* property), 136

facesBack (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 146

facesBack (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 155

facesBack (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 162

facesBack (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 169

facesBack (*fipy.meshes.gmshMesh.Gmsh2D* property), 185

facesBack (*fipy.meshes.gmshMesh.Gmsh2DIn3Dspace* property), 193

facesBack (*fipy.meshes.gmshMesh.Gmsh3D* property), 201

facesBack (*fipy.meshes.gmshMesh.GmshGrid2D* property), 210

facesBack (*fipy.meshes.gmshMesh.GmshGrid3D* property), 218

facesBack (*fipy.meshes.mesh.Mesh* property), 228

facesBack (*fipy.meshes.mesh1D.Mesh1D* property), 237

facesBack (*fipy.meshes.mesh2D.Mesh2D* property), 246

facesBack (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 254

facesBack (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 263

facesBack (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 271

facesBack (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 280

facesBack (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 289

facesBack (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 298

- facesBack** (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 306  
**facesBack** (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 316  
**facesBack** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 323  
**facesBack** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 331  
**facesBack** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 339  
**facesBack** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 347  
**facesBack** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 354  
**facesBack** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 362  
**facesBack** (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 371  
**facesBack** (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 380  
**facesBack** (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 387  
**facesBack** (*fipy.meshes.tri2D.Tri2D* property), 396  
**facesBack** (*fipy.meshes.uniformGrid.UniformGrid* property), 403  
**facesBack** (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 410  
**facesBack** (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 417  
**facesBack** (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 424  
**facesBottom** (*fipy.meshes.abstractMesh.AbstractMesh* property), 136  
**facesBottom** (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 146  
**facesBottom** (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 155  
**facesBottom** (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 162  
**facesBottom** (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 169  
**facesBottom** (*fipy.meshes.gmshMesh.Gmsh2D* property), 185  
**facesBottom** (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 193  
**facesBottom** (*fipy.meshes.gmshMesh.Gmsh3D* property), 201  
**facesBottom** (*fipy.meshes.gmshMesh.GmshGrid2D* property), 210  
**facesBottom** (*fipy.meshes.gmshMesh.GmshGrid3D* property), 218  
**facesBottom** (*fipy.meshes.mesh.Mesh* property), 228  
**facesBottom** (*fipy.meshes.mesh1D.Mesh1D* property), 237  
**facesBottom** (*fipy.meshes.mesh2D.Mesh2D* property), 246  
**facesBottom** (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 254  
**facesBottom** (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 263  
**facesBottom** (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 271  
**facesBottom** (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 280  
**facesBottom** (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 290  
**facesBottom** (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 298  
**facesBottom** (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 306  
**facesBottom** (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 316  
**facesBottom** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 324  
**facesBottom** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 331  
**facesBottom** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 339  
**facesBottom** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 347  
**facesBottom** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 355  
**facesBottom** (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 362  
**facesBottom** (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 371  
**facesBottom** (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 380  
**facesBottom** (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 387  
**facesBottom** (*fipy.meshes.tri2D.Tri2D* property), 396  
**facesBottom** (*fipy.meshes.uniformGrid.UniformGrid* property), 403  
**facesBottom** (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 410  
**facesBottom** (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 417  
**facesBottom** (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 425  
**facesDown** (*fipy.meshes.abstractMesh.AbstractMesh* property), 136  
**facesDown** (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 146  
**facesDown** (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 155  
**facesDown** (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 162  
**facesDown** (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 169  
**facesDown** (*fipy.meshes.gmshMesh.Gmsh2D* property), 185  
**facesDown** (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 194  
**facesDown** (*fipy.meshes.gmshMesh.Gmsh3D* property), 202  
**facesDown** (*fipy.meshes.gmshMesh.GmshGrid2D* property), 211  
**facesDown** (*fipy.meshes.gmshMesh.GmshGrid3D* property), 218  
**facesDown** (*fipy.meshes.mesh.Mesh* property), 229  
**facesDown** (*fipy.meshes.mesh1D.Mesh1D* property), 237  
**facesDown** (*fipy.meshes.mesh2D.Mesh2D* property), 246  
**facesDown** (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 255  
**facesDown** (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 263  
**facesDown** (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 271  
**facesDown** (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 280  
**facesDown** (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 290  
**facesDown** (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 298

- facesDown (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 307
- facesDown (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 316
- facesDown (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 324
- facesDown (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 332
- facesDown (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 339
- facesDown (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 347
- facesDown (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 355
- facesDown (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 363
- facesDown (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 372
- facesDown (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 380
- facesDown (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 387
- facesDown (*fipy.meshes.tri2D.Tri2D* property), 397
- facesDown (*fipy.meshes.uniformGrid.UniformGrid* property), 404
- facesDown (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 411
- facesDown (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 418
- facesDown (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 425
- facesFront (*fipy.meshes.abstractMesh.AbstractMesh* property), 137
- facesFront (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 147
- facesFront (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 155
- facesFront (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 163
- facesFront (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 170
- facesFront (*fipy.meshes.gmshMesh.Gmsh2D* property), 185
- facesFront (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 194
- facesFront (*fipy.meshes.gmshMesh.Gmsh3D* property), 202
- facesFront (*fipy.meshes.gmshMesh.GmshGrid2D* property), 211
- facesFront (*fipy.meshes.gmshMesh.GmshGrid3D* property), 219
- facesFront (*fipy.meshes.mesh.Mesh* property), 229
- facesFront (*fipy.meshes.mesh1D.Mesh1D* property), 238
- facesFront (*fipy.meshes.mesh2D.Mesh2D* property), 246
- facesFront (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 255
- facesFront (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 264
- facesFront (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 272
- facesFront (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 280
- facesFront (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 290
- facesFront (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 299
- facesFront (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 307
- facesFront (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 316
- facesFront (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 324
- facesFront (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 332
- facesFront (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 340
- facesFront (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 347
- facesFront (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 355
- facesFront (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 363
- facesFront (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 372
- facesFront (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 380
- facesFront (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 388
- facesFront (*fipy.meshes.tri2D.Tri2D* property), 397
- facesFront (*fipy.meshes.uniformGrid.UniformGrid* property), 404
- facesFront (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 411
- facesFront (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 418
- facesFront (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 425
- facesLeft (*fipy.meshes.abstractMesh.AbstractMesh* property), 137
- facesLeft (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 147
- facesLeft (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 156
- facesLeft (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 163
- facesLeft (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 170
- facesLeft (*fipy.meshes.gmshMesh.Gmsh2D* property), 185
- facesLeft (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 194
- facesLeft (*fipy.meshes.gmshMesh.Gmsh3D* property), 202
- facesLeft (*fipy.meshes.gmshMesh.GmshGrid2D* property), 211
- facesLeft (*fipy.meshes.gmshMesh.GmshGrid3D* property), 219
- facesLeft (*fipy.meshes.mesh.Mesh* property), 229
- facesLeft (*fipy.meshes.mesh1D.Mesh1D* property), 238
- facesLeft (*fipy.meshes.mesh2D.Mesh2D* property), 247
- facesLeft (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 255
- facesLeft (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 264
- facesLeft (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 272
- facesLeft (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 281
- facesLeft (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 290
- facesLeft (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 299



- facesLeft (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 307
- facesLeft (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 317
- facesLeft (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 324
- facesLeft (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 332
- facesLeft (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 340
- facesLeft (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 348
- facesLeft (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 355
- facesLeft (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 363
- facesLeft (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 372
- facesLeft (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 381
- facesLeft (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 388
- facesLeft (*fipy.meshes.tri2D.Tri2D* property), 397
- facesLeft (*fipy.meshes.uniformGrid.UniformGrid* property), 404
- facesLeft (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 411
- facesLeft (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 418
- facesLeft (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 425
- facesRight (*fipy.meshes.abstractMesh.AbstractMesh* property), 137
- facesRight (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 147
- facesRight (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 156
- facesRight (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 163
- facesRight (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 170
- facesRight (*fipy.meshes.gmshMesh.Gmsh2D* property), 186
- facesRight (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 194
- facesRight (*fipy.meshes.gmshMesh.Gmsh3D* property), 202
- facesRight (*fipy.meshes.gmshMesh.GmshGrid2D* property), 211
- facesRight (*fipy.meshes.gmshMesh.GmshGrid3D* property), 219
- facesRight (*fipy.meshes.mesh.Mesh* property), 229
- facesRight (*fipy.meshes.mesh1D.Mesh1D* property), 238
- facesRight (*fipy.meshes.mesh2D.Mesh2D* property), 247
- facesRight (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 255
- facesRight (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 264
- facesRight (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 272
- facesRight (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 281
- facesRight (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 291
- facesRight (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 299
- facesRight (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 307
- facesRight (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 317
- facesRight (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 325
- facesRight (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 332
- facesRight (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 340
- facesRight (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 348
- facesRight (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 356
- facesRight (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 363
- facesRight (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 372
- facesRight (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 381
- facesRight (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 388
- facesRight (*fipy.meshes.tri2D.Tri2D* property), 397
- facesRight (*fipy.meshes.uniformGrid.UniformGrid* property), 404
- facesRight (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 411
- facesRight (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 418
- facesRight (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 426
- facesTop (*fipy.meshes.abstractMesh.AbstractMesh* property), 137
- facesTop (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 147
- facesTop (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 156
- facesTop (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 163
- facesTop (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 170
- facesTop (*fipy.meshes.gmshMesh.Gmsh2D* property), 186
- facesTop (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 195
- facesTop (*fipy.meshes.gmshMesh.Gmsh3D* property), 203
- facesTop (*fipy.meshes.gmshMesh.GmshGrid2D* property), 212
- facesTop (*fipy.meshes.gmshMesh.GmshGrid3D* property), 219
- facesTop (*fipy.meshes.mesh.Mesh* property), 230
- facesTop (*fipy.meshes.mesh1D.Mesh1D* property), 238
- facesTop (*fipy.meshes.mesh2D.Mesh2D* property), 247
- facesTop (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 256
- facesTop (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 264
- facesTop (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 272
- facesTop (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 281
- facesTop (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 291
- facesTop (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 299
- facesTop (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 308

[facesTop \(fipy.meshes.periodicGrid3D.PeriodicGrid3D property\)](#), 317  
[facesTop \(fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property\)](#), 325  
[facesTop \(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property\)](#), 333  
[facesTop \(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property\)](#), 340  
[facesTop \(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property\)](#), 348  
[facesTop \(fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property\)](#), 356  
[facesTop \(fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property\)](#), 364  
[facesTop \(fipy.meshes.skewedGrid2D.SkewedGrid2D property\)](#), 373  
[facesTop \(fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property\)](#), 381  
[facesTop \(fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property\)](#), 388  
[facesTop \(fipy.meshes.tri2D.Tri2D property\)](#), 398  
[facesTop \(fipy.meshes.uniformGrid.UniformGrid property\)](#), 405  
[facesTop \(fipy.meshes.uniformGrid1D.UniformGrid1D property\)](#), 412  
[facesTop \(fipy.meshes.uniformGrid2D.UniformGrid2D property\)](#), 419  
[facesTop \(fipy.meshes.uniformGrid3D.UniformGrid3D property\)](#), 426  
[facesUp \(fipy.meshes.abstractMesh.AbstractMesh property\)](#), 138  
[facesUp \(fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D property\)](#), 148  
[facesUp \(fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D property\)](#), 156  
[facesUp \(fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D property\)](#), 164  
[facesUp \(fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D property\)](#), 171  
[facesUp \(fipy.meshes.gmshMesh.Gmsh2D property\)](#), 186  
[facesUp \(fipy.meshes.gmshMesh.Gmsh2DIn3DSpace property\)](#), 195  
[facesUp \(fipy.meshes.gmshMesh.Gmsh3D property\)](#), 203  
[facesUp \(fipy.meshes.gmshMesh.GmshGrid2D property\)](#), 212  
[facesUp \(fipy.meshes.gmshMesh.GmshGrid3D property\)](#), 220  
[facesUp \(fipy.meshes.mesh.Mesh property\)](#), 230  
[facesUp \(fipy.meshes.mesh1D.Mesh1D property\)](#), 239  
[facesUp \(fipy.meshes.mesh2D.Mesh2D property\)](#), 247  
[facesUp \(fipy.meshes.nonUniformGrid1D.NonUniformGrid1D property\)](#), 256  
[facesUp \(fipy.meshes.nonUniformGrid2D.NonUniformGrid2D property\)](#), 265  
[facesUp \(fipy.meshes.nonUniformGrid3D.NonUniformGrid3D property\)](#), 273  
[facesUp \(fipy.meshes.periodicGrid1D.PeriodicGrid1D property\)](#), 281  
[facesUp \(fipy.meshes.periodicGrid2D.PeriodicGrid2D property\)](#), 291  
[facesUp \(fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight property\)](#), 300  
[facesUp \(fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom property\)](#), 308  
[facesUp \(fipy.meshes.periodicGrid3D.PeriodicGrid3D property\)](#), 317  
[facesUp \(fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack property\)](#), 325  
[facesUp \(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight property\)](#), 333  
[facesUp \(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack property\)](#), 341  
[facesUp \(fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom property\)](#), 348  
[facesUp \(fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom property\)](#), 356  
[facesUp \(fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack property\)](#), 364  
[facesUp \(fipy.meshes.skewedGrid2D.SkewedGrid2D property\)](#), 373  
[facesUp \(fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D property\)](#), 381  
[facesUp \(fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D property\)](#), 389  
[facesUp \(fipy.meshes.tri2D.Tri2D property\)](#), 398  
[facesUp \(fipy.meshes.uniformGrid.UniformGrid property\)](#), 405  
[facesUp \(fipy.meshes.uniformGrid1D.UniformGrid1D property\)](#), 412  
[facesUp \(fipy.meshes.uniformGrid2D.UniformGrid2D property\)](#), 419  
[facesUp \(fipy.meshes.uniformGrid3D.UniformGrid3D property\)](#), 426  
[faceTerm \(class in fipy.terms.faceTerm\)](#), 522  
[faceValue \(fipy.variables.betaNoiseVariable.BetaNoiseVariable property\)](#), 643  
[faceValue \(fipy.variables.cellVariable.CellVariable property\)](#), 658  
[faceValue \(fipy.variables.distanceVariable.DistanceVariable property\)](#), 677  
[faceValue \(fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable property\)](#), 692  
[faceValue \(fipy.variables.gammaNoiseVariable.GammaNoiseVariable property\)](#), 719  
[faceValue \(fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable property\)](#), 735  
[faceValue \(fipy.variables.histogramVariable.HistogramVariable property\)](#), 750  
[faceValue \(fipy.variables.modularVariable.ModularVariable property\)](#), 776  
[faceValue \(fipy.variables.noiseVariable.NoiseVariable property\)](#), 791  
[faceValue \(fipy.variables.surfactantVariable.SurfactantVariable property\)](#), 830  
[faceValue \(fipy.variables.uniformNoiseVariable.UniformNoiseVariable property\)](#), 845  
[FaceVariable](#), 990  
[FaceVariable \(class in fipy.variables.faceVariable\)](#), 698  
[fig \(examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer property\)](#), 1080  
[fig \(fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer property\)](#), 869  
[fig \(fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer property\)](#), 871  
[fig \(fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer property\)](#), 874  
[fig \(fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer property\)](#), 876  
[fig \(fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer property\)](#), 879  
[fig \(fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer property\)](#), 882

fig  
     (*fiPy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* module, 149  
     property), 885  
 fig  
     (*fiPy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer*  
     property), 889  
 fig  
     (*fiPy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer*  
     property), 893  
 finalize\_options() (*fiPy.tests.test.test* method), 575  
 FiPy, 107  
 fiPy  
     module, 125  
 fiPy.boundaryConditions  
     module, 126  
 fiPy.boundaryConditions.boundaryCondition  
     module, 127  
 fiPy.boundaryConditions.constraint  
     module, 127  
 fiPy.boundaryConditions.fixedFlux  
     module, 128  
 fiPy.boundaryConditions.fixedValue  
     module, 128  
 fiPy.boundaryConditions.nthOrderBoundaryCondition  
     module, 129  
 fiPy.boundaryConditions.test  
     module, 129  
 fiPy.matrices  
     module, 130  
 fiPy.matrices.offsetSparseMatrix  
     module, 130  
 fiPy.matrices.petscMatrix  
     module, 130  
 fiPy.matrices.pysparseMatrix  
     module, 130  
 fiPy.matrices.scipyMatrix  
     module, 130  
 fiPy.matrices.sparseMatrix  
     module, 130  
 fiPy.matrices.test  
     module, 130  
 fiPy.matrices.trilinosMatrix  
     module, 130  
 fiPy.meshes  
     module, 130  
 fiPy.meshes.abstractMesh  
     module, 132  
 fiPy.meshes.builders  
     module, 139  
 fiPy.meshes.builders.abstractGridBuilder  
     module, 140  
 fiPy.meshes.builders.grid1DBuilder  
     module, 140  
 fiPy.meshes.builders.grid2DBuilder  
     module, 140  
 fiPy.meshes.builders.grid3DBuilder  
     module, 140  
 fiPy.meshes.builders.periodicGrid1DBuilder  
     module, 140  
 fiPy.meshes.builders.utilityClasses  
     module, 140  
 fiPy.meshes.cylindricalGrid1D  
     module, 140  
 fiPy.meshes.cylindricalGrid2D  
     module, 140  
 fiPy.meshes.cylindricalNonUniformGrid1D  
     module, 140  
 fiPy.meshes.cylindricalNonUniformGrid2D  
     module, 149  
 fiPy.meshes.cylindricalUniformGrid1D  
     module, 158  
 fiPy.meshes.cylindricalUniformGrid2D  
     module, 165  
 fiPy.meshes.factoryMeshes  
     module, 172  
 fiPy.meshes.gmshMesh  
     module, 175  
 fiPy.meshes.grid1D  
     module, 223  
 fiPy.meshes.grid2D  
     module, 223  
 fiPy.meshes.grid3D  
     module, 223  
 fiPy.meshes.mesh  
     module, 223  
 fiPy.meshes.mesh1D  
     module, 232  
 fiPy.meshes.mesh2D  
     module, 240  
 fiPy.meshes.nonUniformGrid1D  
     module, 249  
 fiPy.meshes.nonUniformGrid2D  
     module, 257  
 fiPy.meshes.nonUniformGrid3D  
     module, 266  
 fiPy.meshes.periodicGrid1D  
     module, 274  
 fiPy.meshes.periodicGrid2D  
     module, 283  
 fiPy.meshes.periodicGrid3D  
     module, 309  
 fiPy.meshes.representations  
     module, 365  
 fiPy.meshes.representations.abstractRepresentation  
     module, 365  
 fiPy.meshes.representations.gridRepresentation  
     module, 365  
 fiPy.meshes.representations.meshRepresentation  
     module, 365  
 fiPy.meshes.skewedGrid2D  
     module, 365  
 fiPy.meshes.sphericalNonUniformGrid1D  
     module, 374  
 fiPy.meshes.sphericalUniformGrid1D  
     module, 383  
 fiPy.meshes.test  
     module, 390  
 fiPy.meshes.topologies  
     module, 390  
 fiPy.meshes.topologies.abstractTopology  
     module, 390  
 fiPy.meshes.topologies.gridTopology  
     module, 390  
 fiPy.meshes.topologies.meshTopology  
     module, 390  
 fiPy.meshes.tri2D  
     module, 390  
 fiPy.meshes.uniformGrid  
     module, 399  
 fiPy.meshes.uniformGrid1D  
     module, 406  
 fiPy.meshes.uniformGrid2D  
     module, 413  
 fiPy.meshes.uniformGrid3D  
     module, 420  
 fiPy.numerix

module, 1169  
 fipy.solvers  
   module, 427  
 fipy.solvers.petsc  
   module, 429  
   fipy.solvers.petsc.comms  
     module, 430  
   fipy.solvers.petsc.comms.parallelPETScCommWrapper  
     module, 430  
   fipy.solvers.petsc.comms.petscCommWrapper  
     module, 430  
   fipy.solvers.petsc.comms.serialPETScCommWrapper  
     module, 431  
   fipy.solvers.petsc.dummySolver  
     module, 431  
   fipy.solvers.petsc.linearBicgSolver  
     module, 431  
   fipy.solvers.petsc.linearCGSSolver  
     module, 432  
   fipy.solvers.petsc.linearGMRESSolver  
     module, 432  
   fipy.solvers.petsc.linearLUSolver  
     module, 433  
   fipy.solvers.petsc.linearPCGSolver  
     module, 433  
   fipy.solvers.petsc.petscKrylovSolver  
     module, 434  
   fipy.solvers.petsc.petscSolver  
     module, 435  
   fipy.solvers.pyAMG  
     module, 435  
   fipy.solvers.pyAMG.linearCGSSolver  
     module, 436  
   fipy.solvers.pyAMG.linearGeneralSolver  
     module, 437  
   fipy.solvers.pyAMG.linearGMRESSolver  
     module, 436  
   fipy.solvers.pyAMG.linearLUSolver  
     module, 438  
   fipy.solvers.pyAMG.linearPCGSolver  
     module, 438  
   fipy.solvers.pyAMG.preconditioners  
     module, 439  
   fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioners  
     module, 439  
   fipy.solvers.pyamgx  
     module, 439  
   fipy.solvers.pyamgx.aggregationAMGSolver  
     module, 440  
   fipy.solvers.pyamgx.classicalAMGSolver  
     module, 440  
   fipy.solvers.pyamgx.linearBiCGStabSolver  
     module, 441  
   fipy.solvers.pyamgx.linearCGSolver  
     module, 441  
   fipy.solvers.pyamgx.linearFGMRESSolver  
     module, 442  
   fipy.solvers.pyamgx.linearGMRESSolver  
     module, 443  
   fipy.solvers.pyamgx.linearLUSolver  
     module, 443  
   fipy.solvers.pyamgx.preconditioners  
     module, 444  
   fipy.solvers.pyamgx.preconditioners.preconditioners  
     module, 444  
   fipy.solvers.pyamgx.pyAMGXSolver  
     module, 444  
   fipy.solvers.pyamgx.smoother  
     module, 445  
   fipy.solvers.pyamgx.smoother.smoother  
     module, 445  
   fipy.solvers.pysparse  
     module, 445  
     fipy.solvers.pysparse.linearCGSSolver  
       module, 446  
     fipy.solvers.pysparse.linearGMRESSolver  
       module, 446  
     fipy.solvers.pysparse.linearJORSolver  
       module, 447  
     fipy.solvers.pysparse.linearLUSolver  
       module, 447  
     fipy.solvers.pysparse.linearPCGSolver  
       module, 448  
     fipy.solvers.pysparse.preconditioners  
       module, 448  
     fipy.solvers.pysparse.preconditioners.jacobiPreconditioner  
       module, 448  
     fipy.solvers.pysparse.preconditioners.preconditioner  
       module, 449  
     fipy.solvers.pysparse.preconditioners.sorPreconditioner  
       module, 449  
     fipy.solvers.pysparse.pysparseSolver  
       module, 449  
     fipy.solvers.pysparseMatrixSolver  
       module, 450  
   fipy.solvers.scipy  
     module, 450  
     fipy.solvers.scipy.linearBicgstabSolver  
       module, 450  
     fipy.solvers.scipy.linearCGSSolver  
       module, 451  
     fipy.solvers.scipy.linearGMRESSolver  
       module, 451  
     fipy.solvers.scipy.linearLUSolver  
       module, 452  
     fipy.solvers.scipy.linearPCGSolver  
       module, 452  
     fipy.solvers.scipy.scipyKrylovSolver  
       module, 453  
     fipy.solvers.scipy.scipySolver  
       module, 453  
   fipy.solvers.solver  
     module, 453  
   fipy.solvers.test  
     module, 458  
   fipy.solvers.trilinos  
     module, 458  
     fipy.solvers.trilinos.comms  
       module, 459  
     fipy.solvers.trilinos.comms.epetraCommWrapper  
       module, 459  
     fipy.solvers.trilinos.comms.parallelEpetraCommWrapper  
       module, 460  
     fipy.solvers.trilinos.comms.serialEpetraCommWrapper  
       module, 460  
     fipy.solvers.trilinos.linearBicgstabSolver  
       module, 461  
     fipy.solvers.trilinos.linearCGSSolver  
       module, 461  
     fipy.solvers.trilinos.linearGMRESSolver  
       module, 462  
     fipy.solvers.trilinos.linearLUSolver  
       module, 462  
     fipy.solvers.trilinos.linearPCGSolver  
       module, 463  
     fipy.solvers.trilinos.preconditioners

module, 464  
 fipy.solvers.trilinos.preconditioners.domDecompPreconditioner module, 464  
 fipy.solvers.trilinos.preconditioners.icPreconditioner module, 464  
 fipy.solvers.trilinos.preconditioners.jacobiPreconditioner module, 465  
 fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner module, 465  
 fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner module, 465  
 fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner module, 466  
 fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner module, 466  
 fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner module, 466  
 fipy.solvers.trilinos.preconditioners.multilevelSolverSmoothingPreconditioner module, 467  
 fipy.solvers.trilinos.preconditioners.preconditioner module, 467  
 fipy.solvers.trilinos.trilinosAztec00Solver module, 468  
 fipy.solvers.trilinos.trilinosMLTest module, 469  
 fipy.solvers.trilinos.trilinosNonlinearSolver module, 469  
 fipy.solvers.trilinos.trilinosSolver module, 470  
 fipy.steppers module, 471  
 fipy.steppers.pidStepper module, 473  
 fipy.steppers.pseudoRKQSStepper module, 474  
 fipy.steppers.stepper module, 474  
 fipy.terms module, 474  
 fipy.terms.abstractBinaryTerm module, 481  
 fipy.terms.abstractConvectionTerm module, 481  
 fipy.terms.abstractDiffusionTerm module, 481  
 fipy.terms.abstractUpwindConvectionTerm module, 481  
 fipy.terms.advectionTerm module, 481  
 fipy.terms.asymmetricConvectionTerm module, 488  
 fipy.terms.binaryTerm module, 488  
 fipy.terms.cellTerm module, 488  
 fipy.terms.centralDiffConvectionTerm module, 492  
 fipy.terms.coupledBinaryTerm module, 497  
 fipy.terms.diffusionTerm module, 497  
 fipy.terms.diffusionTermCorrection module, 501  
 fipy.terms.diffusionTermNoCorrection module, 504  
 fipy.terms.explicitDiffusionTerm module, 508  
 fipy.terms.explicitSourceTerm module, 511  
 fipy.terms.explicitUpwindConvectionTerm module, 511  
 fipy.terms.exponentialConvectionTerm module, 516  
 fipy.terms.faceTerm module, 522  
 fipy.terms.firstOrderAdvectionTerm module, 526  
 fipy.terms.hybridConvectionTerm module, 531  
 fipy.terms.implicitDiffusionTerm module, 536  
 fipy.terms.implicitDiffusionTerm.DiffusionTerm object, 964, 969  
 fipy.terms.implicitSourceTerm module, 536  
 fipy.terms.implicitDiffusionTerm module, 540  
 fipy.terms.powerLawConvectionTerm module, 540  
 fipy.terms.residualTerm module, 546  
 fipy.terms.sourceTerm module, 550  
 fipy.terms.term module, 554  
 fipy.terms.test module, 558  
 fipy.terms.transientTerm module, 558  
 fipy.terms.transientTerm.TransientTerm object, 964, 969  
 fipy.terms.unaryTerm module, 563  
 fipy.terms.upwindConvectionTerm module, 563  
 fipy.terms.vanLeerConvectionTerm module, 568  
 fipy.testFiPy module, 573  
 fipy.tests module, 573  
 fipy.tests.doctestPlus module, 573  
 fipy.tests.lateImportTest module, 575  
 fipy.tests.test module, 575  
 fipy.tests.testProgram module, 576  
 fipy.tools module, 576  
 fipy.tools.comms module, 593  
 fipy.tools.comms.commWrapper module, 594  
 fipy.tools.comms.dummyComm module, 594  
 fipy.tools.debug module, 594  
 fipy.tools.decorators module, 595  
 fipy.tools.dimensions module, 595  
 fipy.tools.dimensions.DictWithDefault module, 595  
 fipy.tools.dimensions.NumberDict

- module, 595
- fiPy.tools.dimensions.physicalField
  - module, 595
- fiPy.tools.dump
  - module, 621, 1122
- fiPy.tools.inline
  - module, 622
- fiPy.tools.logging
  - module, 622
- fiPy.tools.logging.environment
  - module, 622
- fiPy.tools.numerix
  - module, 623
- fiPy.tools.parser
  - module, 628, 1068, 1119
- fiPy.tools.sharedtempfile
  - module, 629
- fiPy.tools.test
  - module, 630
- fiPy.tools.timer
  - module, 630
- fiPy.tools.vector
  - module, 630
- fiPy.tools.version
  - module, 631
- fiPy.variables
  - module, 631
- fiPy.variables.addOverFacesVariable
  - module, 633
- fiPy.variables.arithmeticCellToFaceVariable
  - module, 633
- fiPy.variables.betaNoiseVariable
  - module, 633
- fiPy.variables.binaryOperatorVariable
  - module, 649
- fiPy.variables.cellToFaceVariable
  - module, 649
- fiPy.variables.cellVariable
  - module, 649
- fiPy.variables.cellVariable.CellVariable
  - object, 963, 968
- fiPy.variables.constant
  - module, 664
- fiPy.variables.constraintMask
  - module, 664
- fiPy.variables.coupledCellVariable
  - module, 664
- fiPy.variables.distanceVariable
  - module, 664
- fiPy.variables.exponentialNoiseVariable
  - module, 683
- fiPy.variables.faceGradContributionsVariable
  - module, 698
- fiPy.variables.faceGradVariable
  - module, 698
- fiPy.variables.faceVariable
  - module, 698
- fiPy.variables.gammaNoiseVariable
  - module, 709
- fiPy.variables.gaussCellGradVariable
  - module, 725
- fiPy.variables.gaussianNoiseVariable
  - module, 725
- fiPy.variables.harmonicCellToFaceVariable
  - module, 742
- fiPy.variables.histogramVariable
  - module, 742
- fiPy.variables.interfaceAreaVariable
  - module, 756
- fiPy.variables.interfaceFlagVariable
  - module, 756
- fiPy.variables.leastSquaresCellGradVariable
  - module, 756
- fiPy.variables.levelSetDiffusionVariable
  - module, 756
- fiPy.variables.meshVariable
  - module, 756
- fiPy.variables.minmodCellToFaceVariable
  - module, 767
- fiPy.variables.modCellGradVariable
  - module, 767
- fiPy.variables.modCellToFaceVariable
  - module, 767
- fiPy.variables.modFaceGradVariable
  - module, 767
- fiPy.variables.modPhysicalField
  - module, 767
- fiPy.variables.modularVariable
  - module, 767
- fiPy.variables.noiseVariable
  - module, 782
- fiPy.variables.operatorVariable
  - module, 798
- fiPy.variables.scharfetterGummelFaceVariable
  - module, 798
- fiPy.variables.surfactantConvectionVariable
  - module, 809
- fiPy.variables.surfactantVariable
  - module, 820
- fiPy.variables.test
  - module, 836
- fiPy.variables.unaryOperatorVariable
  - module, 836
- fiPy.variables.uniformNoiseVariable
  - module, 836
- fiPy.variables.variable
  - module, 851
- fiPy.viewers, 1171
  - module, 861, 932, 941, 942, 944, 948, 951, 952, 985, 1004, 1007, 1011, 1052, 1099, 1106, 1115, 1121, 1125, 1147
- fiPy.viewers.matplotlibViewer
  - module, 865
- fiPy.viewers.matplotlibViewer.abstractMatplotlib2DViewer
  - module, 867
- fiPy.viewers.matplotlibViewer.abstractMatplotlibViewer
  - module, 870
- fiPy.viewers.matplotlibViewer.matplotlib1DViewer
  - module, 872
- fiPy.viewers.matplotlibViewer.matplotlib2DContourViewer
  - module, 875
- fiPy.viewers.matplotlibViewer.matplotlib2DGridContourViewer
  - module, 877
- fiPy.viewers.matplotlibViewer.matplotlib2DGridViewer
  - module, 881
- fiPy.viewers.matplotlibViewer.matplotlib2DViewer
  - module, 883
- fiPy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer
  - module, 886
- fiPy.viewers.matplotlibViewer.matplotlibStreamViewer
  - module, 886
- fiPy.viewers.matplotlibViewer.matplotlibVectorViewer
  - module, 891
- fiPy.viewers.matplotlibViewer.test
  - module, 894
- fiPy.viewers.mayaviViewer
  - module, 894

fipy.viewers.mayaviViewer.mayaviClient module, 898  
 fipy.viewers.mayaviViewer.mayaviDaemon module, 901  
 fipy.viewers.mayaviViewer.test module, 902  
 fipy.viewers.multiViewer module, 902  
 fipy.viewers.test module, 904  
 fipy.viewers.testinteractive module, 904  
 fipy.viewers.tsvViewer module, 904  
 fipy.viewers.tsvViewer.TSVViewer object, 965, 970  
 fipy.viewers.viewer module, 906  
 fipy.viewers.vtkViewer module, 908  
 fipy.viewers.vtkViewer.test module, 912  
 fipy.viewers.vtkViewer.vtkCellViewer module, 912  
 fipy.viewers.vtkViewer.vtkFaceViewer module, 914  
 fipy.viewers.vtkViewer.vtkViewer module, 916  
 FIPY\_INCLUDE\_NUMERIX\_ALL, 1176  
 FIPY\_LOG\_CONFIG, 32, 35  
 FIPY\_SOLVERS, 25, 34  
 FirstOrderAdvectionTerm (class in *fipy.terms.firstOrderAdvectionTerm*), 526  
 FixedFlux, 1167  
 FixedFlux (class in *fipy.boundaryConditions.fixedFlux*), 128  
 FixedValue, 1167, 1170  
 FixedValue (class in *fipy.boundaryConditions.fixedValue*), 128  
 floor() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 608  
 floor() (*fipy.tools.PhysicalField* method), 586  
 fps (*fipy.viewers.mayaviViewer.MayaviClient* property), 896  
 fps (*fipy.viewers.mayaviViewer.mayaviClient.MayaviClient* property), 900

## G

GammaNoiseVariable (class in *fipy.variables.gammaNoiseVariable*), 709  
 gaussGrad (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 643  
 gaussGrad (*fipy.variables.cellVariable.CellVariable* property), 658  
 gaussGrad (*fipy.variables.distanceVariable.DistanceVariable* property), 677  
 gaussGrad (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 693  
 gaussGrad (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 720  
 gaussGrad (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 736  
 gaussGrad (*fipy.variables.histogramVariable.HistogramVariable* property), 751  
 gaussGrad (*fipy.variables.modularVariable.ModularVariable* property), 777  
 gaussGrad (*fipy.variables.noiseVariable.NoiseVariable* property), 792  
 gaussGrad (*fipy.variables.surfactantVariable.SurfactantVariable* property), 830

gaussGrad (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 846  
 GaussianNoiseVariable (class in *fipy.variables.gaussianNoiseVariable*), 725  
 GeneralSolver (in module *fipy.solvers*), 428  
 getsctype() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 608  
 getsctype() (*fipy.tools.PhysicalField* method), 586  
 getsctype() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 643  
 getsctype() (*fipy.variables.cellVariable.CellVariable* method), 659  
 getsctype() (*fipy.variables.distanceVariable.DistanceVariable* method), 677  
 getsctype() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 693  
 getsctype() (*fipy.variables.faceVariable.FaceVariable* method), 706  
 getsctype() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 720  
 getsctype() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 736  
 getsctype() (*fipy.variables.histogramVariable.HistogramVariable* method), 751  
 getsctype() (*fipy.variables.meshVariable.MeshVariable* method), 764  
 getsctype() (*fipy.variables.modularVariable.ModularVariable* method), 777  
 getsctype() (*fipy.variables.noiseVariable.NoiseVariable* method), 792  
 getsctype() (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 805  
 getsctype() (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 817  
 getsctype() (*fipy.variables.surfactantVariable.SurfactantVariable* method), 830  
 getsctype() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 846  
 getsctype() (*fipy.variables.variable.Variable* method), 858  
 getShape() (in module *fipy.tools.numerix*), 625  
 getUnit() (in module *fipy.tools.numerix*), 626  
 Gist1DViewer, 1171  
 GitHub Actions, 107  
 globalValue (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 644  
 globalValue (*fipy.variables.cellVariable.CellVariable* property), 659  
 globalValue (*fipy.variables.distanceVariable.DistanceVariable* property), 677  
 globalValue (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 693  
 globalValue (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 720  
 globalValue (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 736  
 globalValue (*fipy.variables.histogramVariable.HistogramVariable* property), 751  
 globalValue (*fipy.variables.modularVariable.ModularVariable* property), 777  
 globalValue (*fipy.variables.noiseVariable.NoiseVariable* property), 792

globalValue (*fipy.variables.surfactantVariable.SurfactantVariable* property), 830

globalValue (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 846

Gmsh, 107

gmsh module, 1066, 1074, 1085

Gmsh2D (*class in fipy.meshes.gmshMesh*), 175

Gmsh2DIn3DSpace (*class in fipy.meshes.gmshMesh*), 187

Gmsh3D (*class in fipy.meshes.gmshMesh*), 196

GmshException, 204

GmshFile (*class in fipy.meshes.gmshMesh*), 204

GmshGrid2D (*class in fipy.meshes.gmshMesh*), 205

GmshGrid3D (*class in fipy.meshes.gmshMesh*), 213

gmshVersion() (*in module fipy.meshes.gmshMesh*), 222

grad (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 644

grad (*fipy.variables.cellVariable.CellVariable* property), 659

grad (*fipy.variables.distanceVariable.DistanceVariable* property), 678

grad (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 693

grad (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 720

grad (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 736

grad (*fipy.variables.histogramVariable.HistogramVariable* property), 751

grad (*fipy.variables.modularVariable.ModularVariable* property), 777

grad (*fipy.variables.noiseVariable.NoiseVariable* property), 792

grad (*fipy.variables.surfactantVariable.SurfactantVariable* property), 830

grad (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 846

Grid1D, 940, 943, 947, 950, 1010, 1057, 1100, 1109, 1123, 1140, 1147, 1169

Grid1D() (*in module fipy.meshes.factoryMeshes*), 173

Grid2D, 930, 1003, 1006, 1050, 1060, 1062, 1069, 1097, 1119, 1167

Grid2D() (*in module fipy.meshes.factoryMeshes*), 173

Grid2DGistViewer, 1169

Grid3D() (*in module fipy.meshes.factoryMeshes*), 174

## H

harmonicFaceValue (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 644

harmonicFaceValue (*fipy.variables.cellVariable.CellVariable* property), 659

harmonicFaceValue (*fipy.variables.distanceVariable.DistanceVariable* property), 678

harmonicFaceValue (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 693

harmonicFaceValue (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 720

harmonicFaceValue (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 736

harmonicFaceValue (*fipy.variables.histogramVariable.HistogramVariable* property), 751

harmonicFaceValue (*fipy.variables.modularVariable.ModularVariable* property), 777

harmonicFaceValue (*fipy.variables.noiseVariable.NoiseVariable* property), 792

harmonicFaceValue (*fipy.variables.surfactantVariable.SurfactantVariable* property), 831

harmonicFaceValue (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 846

HistogramVariable (*class in fipy.variables.histogramVariable*), 742

HybridConvectionTerm (*class in fipy.terms.hybridConvectionTerm*), 531

## I

ICPreconditioner (*class in fipy.solvers.trilinos.preconditioners.icPreconditioner*), 464

id (*exam- ples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer* property), 1080

id (*fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2D* property), 869

id (*fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewe* property), 871

id (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 874

id (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContour* property), 876

id (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGrid* property), 879

id (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* property), 882

id (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* property), 885

id (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 889

id (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* property), 893

IllConditionedPreconditionerWarning, 453

ImplicitDiffusionTerm (*in module fipy.terms.implicitDiffusionTerm*), 536

ImplicitSourceTerm, 1099, 1120, 1124, 1151

ImplicitSourceTerm (*class in fipy.terms.implicitSourceTerm*), 536

inBaseUnits() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 608

inBaseUnits() (*fipy.tools.physicalField* method), 586

inBaseUnits() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 644

inBaseUnits() (*fipy.variables.cellVariable.CellVariable* method), 659

inBaseUnits() (*fipy.variables.distanceVariable.DistanceVariable* method), 678

inBaseUnits() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 694

inBaseUnits() (*fipy.variables.faceVariable.FaceVariable* method), 706

inBaseUnits() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 721

inBaseUnits() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 736



- method*), 737
  - `inBaseUnits()` (*fipy.variables.histogramVariable.HistogramVariable method*), 752
  - `inBaseUnits()` (*fipy.variables.meshVariable.MeshVariable method*), 764
  - `inBaseUnits()` (*fipy.variables.modularVariable.ModularVariable method*), 778
  - `inBaseUnits()` (*fipy.variables.noiseVariable.NoiseVariable method*), 793
  - `inBaseUnits()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 806
  - `inBaseUnits()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 817
  - `inBaseUnits()` (*fipy.variables.surfactantVariable.SurfactantVariable method*), 831
  - `inBaseUnits()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 847
  - `inBaseUnits()` (*fipy.variables.variable.Variable method*), 858
  - `IncorrectSolutionVariable`, 476
  - `inDimensionless()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 608
  - `inDimensionless()` (*fipy.tools.PhysicalField method*), 586
  - `initialize_options()` (*fipy.tests.test.test method*), 575
  - `inRadians()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 609
  - `inRadians()` (*fipy.tools.PhysicalField method*), 587
  - `inSIUnits()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 609
  - `inSIUnits()` (*fipy.tools.PhysicalField method*), 587
  - `interfaceVar` (*fipy.variables.surfactantVariable.SurfactantVariable property*), 832
  - `inUnitsOf()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 609
  - `inUnitsOf()` (*fipy.tools.PhysicalField method*), 587
  - `inUnitsOf()` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable method*), 644
  - `inUnitsOf()` (*fipy.variables.cellVariable.CellVariable method*), 660
  - `inUnitsOf()` (*fipy.variables.distanceVariable.DistanceVariable method*), 678
  - `inUnitsOf()` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 694
  - `inUnitsOf()` (*fipy.variables.faceVariable.FaceVariable method*), 706
  - `inUnitsOf()` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 721
  - `inUnitsOf()` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 737
  - `inUnitsOf()` (*fipy.variables.histogramVariable.HistogramVariable method*), 752
  - `inUnitsOf()` (*fipy.variables.meshVariable.MeshVariable method*), 764
  - `inUnitsOf()` (*fipy.variables.modularVariable.ModularVariable method*), 778
  - `inUnitsOf()` (*fipy.variables.noiseVariable.NoiseVariable method*), 793
  - `inUnitsOf()` (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 806
  - `inUnitsOf()` (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 817
  - `inUnitsOf()` (*fipy.variables.surfactantVariable.SurfactantVariable method*), 831
  - `inUnitsOf()` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 847
  - `inUnitsOf()` (*fipy.variables.variable.Variable method*), 859
  - `IPython`, 107
  - `isAngle()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 619
  - `isclose()` (*in module fipy.tools.numerix*), 626
  - `isCompatible()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 619
  - `isDimensionless()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 620
  - `isDimensionlessOrAngle()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 620
  - `isInverseAngle()` (*fipy.tools.dimensions.physicalField.PhysicalUnit method*), 620
  - `itemset()` (*fipy.tools.dimensions.physicalField.PhysicalField method*), 610
  - `itemset()` (*fipy.tools.PhysicalField method*), 588
  - `Iterator`, 1168
- ## J
- `JacobiPreconditioner` (*class in fipy.solvers.pysparse.preconditioners.jacobiPreconditioner*), 448
  - `JacobiPreconditioner` (*class in fipy.solvers.trilinos.preconditioners.jacobiPreconditioner*), 465
  - `JSON`, 107
  - `justErrorVector()` (*fipy.terms.advectionTerm.AdvectionTerm method*), 485
  - `justErrorVector()` (*fipy.terms.cellTerm.CellTerm method*), 489
  - `justErrorVector()` (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm method*), 494
  - `justErrorVector()` (*fipy.terms.diffusionTerm.DiffusionTerm method*), 498
  - `justErrorVector()` (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection method*), 501
  - `justErrorVector()` (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection method*), 505
  - `justErrorVector()` (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm method*), 508
  - `justErrorVector()` (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm method*), 514
  - `justErrorVector()` (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm method*), 519
  - `justErrorVector()` (*fipy.terms.faceTerm.FaceTerm method*), 523
  - `justErrorVector()` (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm method*), 528
  - `justErrorVector()` (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm method*), 533
  - `justErrorVector()` (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm method*), 538
  - `justErrorVector()`

(*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 543

`justErrorVector()` (*fipy.terms.residualTerm.ResidualTerm* method), 547

`justErrorVector()` (*fipy.terms.sourceTerm.SourceTerm* method), 551

`justErrorVector()` (*fipy.terms.term.Term* method), 555

`justErrorVector()` (*fipy.terms.transientTerm.TransientTerm* method), 560

`justErrorVector()` (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 565

`justErrorVector()` (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 570

`justResidualVector()` (*fipy.terms.advectionTerm.AdvectionTerm* method), 485

`justResidualVector()` (*fipy.terms.cellTerm.CellTerm* method), 490

`justResidualVector()` (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 495

`justResidualVector()` (*fipy.terms.diffusionTerm.DiffusionTerm* method), 499

`justResidualVector()` (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* method), 502

`justResidualVector()` (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* method), 505

`justResidualVector()` (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* method), 509

`justResidualVector()` (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 514

`justResidualVector()` (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 519

`justResidualVector()` (*fipy.terms.faceTerm.FaceTerm* method), 524

`justResidualVector()` (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 529

`justResidualVector()` (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 534

`justResidualVector()` (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 538

`justResidualVector()` (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 543

`justResidualVector()` (*fipy.terms.residualTerm.ResidualTerm* method), 547

`justResidualVector()` (*fipy.terms.sourceTerm.SourceTerm* method), 552

`justResidualVector()` (*fipy.terms.term.Term* method), 555

`justResidualVector()` (*fipy.terms.transientTerm.TransientTerm* method), 560

`justResidualVector()` (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 566

`justResidualVector()` (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 571

## K

`kwargs`

(*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 889

## L

`L1error()` (in module *fipy.steps*), 471

`L1norm()` (in module *fipy.tools.numerix*), 624

`L2error()` (in module *fipy.steps*), 471

`L2norm()` (in module *fipy.tools.numerix*), 624

`leastSquaresGrad` (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 645

`leastSquaresGrad` (*fipy.variables.cellVariable.CellVariable* property), 660

`leastSquaresGrad` (*fipy.variables.distanceVariable.DistanceVariable* property), 679

`leastSquaresGrad` (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 694

`leastSquaresGrad` (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 721

`leastSquaresGrad` (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 738

`leastSquaresGrad` (*fipy.variables.histogramVariable.HistogramVariable* property), 753

`leastSquaresGrad` (*fipy.variables.modularVariable.ModularVariable* property), 778

`leastSquaresGrad` (*fipy.variables.noiseVariable.NoiseVariable* property), 794

`leastSquaresGrad` (*fipy.variables.surfactantVariable.SurfactantVariable* property), 832

`leastSquaresGrad` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 847

`levels` (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* property), 876

`levels` (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* property), 879

`LinearBicgSolver` (class in *fipy.solvers.petsc.linearBicgSolver*), 431

`LinearBiCGStabSolver` (class in *fipy.solvers.pyamgx.linearBiCGStabSolver*), 441

`LinearBicgstabSolver` (class in *fipy.solvers.scipy.linearBicgstabSolver*), 450

`LinearBicgstabSolver` (class in *fipy.solvers.trilinos.linearBicgstabSolver*), 461

`LinearCGSolver` (class in *fipy.solvers.pyamgx.linearCGSolver*), 441

`LinearCGSSolver` (class in *fipy.solvers.petsc.linearCGSSolver*), 432

`LinearCGSSolver` (class in *fipy.solvers.pyAMG.linearCGSSolver*), 436

`LinearCGSSolver` (class in *fipy.solvers.pysparse.linearCGSSolver*), 446

`LinearCGSSolver` (class in *fipy.solvers.scipy.linearCGSSolver*), 451

`LinearCGSSolver` (class in *fipy.solvers.trilinos.linearCGSSolver*), 461

`LinearFGMRESSolver` (class in *fipy.solvers.pyamgx.linearFGMRESSolver*), 442

`LinearGeneralSolver` (class in *fipy.solvers.pyAMG.linearGeneralSolver*), 437

`LinearGMRESSolver` (class in *fipy.solvers.petsc.linearGMRESSolver*), 432

`LinearGMRESSolver` (class in *fipy.solvers.pyAMG.linearGMRESSolver*), 436

- LinearGMRESSolver (class in *fiPy.solvers.pyamgx.linearGMRESSolver*), 443
- LinearGMRESSolver (class in *fiPy.solvers.pysparse.linearGMRESSolver*), 446
- LinearGMRESSolver (class in *fiPy.solvers.scipy.linearGMRESSolver*), 451
- LinearGMRESSolver (class in *fiPy.solvers.trilinos.linearGMRESSolver*), 462
- LinearJORSolver (class in *fiPy.solvers.pysparse.linearJORSolver*), 447
- LinearLUSolver, 931, 1115, 1168
- LinearLUSolver (class in *fiPy.solvers.petsc.linearLUSolver*), 433
- LinearLUSolver (class in *fiPy.solvers.pyAMG.linearLUSolver*), 438
- LinearLUSolver (class in *fiPy.solvers.pyamgx.linearLUSolver*), 443
- LinearLUSolver (class in *fiPy.solvers.pysparse.linearLUSolver*), 447
- LinearLUSolver (class in *fiPy.solvers.scipy.linearLUSolver*), 452
- LinearLUSolver (class in *fiPy.solvers.trilinos.linearLUSolver*), 462
- LinearPCGSolver (class in *fiPy.solvers.petsc.linearPCGSolver*), 433
- LinearPCGSolver (class in *fiPy.solvers.pyAMG.linearPCGSolver*), 438
- LinearPCGSolver (class in *fiPy.solvers.pysparse.linearPCGSolver*), 448
- LinearPCGSolver (class in *fiPy.solvers.scipy.linearPCGSolver*), 452
- LinearPCGSolver (class in *fiPy.solvers.trilinos.linearPCGSolver*), 463
- LinearPCGSolver (in module *fiPy.solvers.pyamgx.linearCGSolver*), 442
- lines
  - (*fiPy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 874
- LINError() (in module *fiPy.steps*), 472
- LINFnorm() (in module *fiPy.tools.numerix*), 624
- linux, 108
- loadtxt, 1073, 1100, 1121, 1125
- log, 1105, 1114, 1143
- log (exam-
  - ples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer* property), 1080
- log
  - (*fiPy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* property), 869
- log
  - (*fiPy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* property), 871
- log
  - (*fiPy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 874
- log
  - (*fiPy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* property), 876
- log
  - (*fiPy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* property), 879
- log
  - (*fiPy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* property), 882
- log
  - (*fiPy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* property), 885
- log
  - (*fiPy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 889
- log
  - (*fiPy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* property), 893
- log() (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 610
- log() (*fiPy.tools.PhysicalField* method), 588
- log10() (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 610
- log10() (*fiPy.tools.PhysicalField* method), 588
- M**
- macOS, 108
- mag (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* property), 645
- mag (*fiPy.variables.cellVariable.CellVariable* property), 661
- mag (*fiPy.variables.distanceVariable.DistanceVariable* property), 679
- mag (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 695
- mag (*fiPy.variables.faceVariable.FaceVariable* property), 707
- mag (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 722
- mag (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 738
- mag (*fiPy.variables.histogramVariable.HistogramVariable* property), 753
- mag (*fiPy.variables.meshVariable.MeshVariable* property), 765
- mag (*fiPy.variables.modularVariable.ModularVariable* property), 779
- mag (*fiPy.variables.noiseVariable.NoiseVariable* property), 794
- mag
  - (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 806
- mag
  - (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 818
- mag
  - (*fiPy.variables.surfactantVariable.SurfactantVariable* property), 832
- mag (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 848
- mag (*fiPy.variables.variable.Variable* property), 859
- main (in module *fiPy.tests.testProgram*), 576
- main() (in module *fiPy.viewers.mayaviViewer.mayaviDaemon*), 902
- makeMapVariables() (*fiPy.meshes.gmshMesh.MSHFile* method), 221
- Matplotlib, 108
- Matplotlib1DViewer (class in *fiPy.viewers.matplotlibViewer.matplotlib1DViewer*), 872
- Matplotlib2DContourViewer (class in *fiPy.viewers.matplotlibViewer.matplotlib2DContourViewer*), 875
- Matplotlib2DGridContourViewer (class in *fiPy.viewers.matplotlibViewer.matplotlib2DGridContourViewer*), 877
- Matplotlib2DGridViewer (class in *fiPy.viewers.matplotlibViewer.matplotlib2DGridViewer*), 881
- Matplotlib2DViewer (class in *fiPy.viewers.matplotlibViewer.matplotlib2DViewer*), 883
- MatplotlibSparseMatrixViewer (class in *fiPy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer*), 886
- MatplotlibStreamViewer (class in *fiPy.viewers.matplotlibViewer.matplotlibStreamViewer*), 886
- MatplotlibSurfactantViewer (class in *exam- ples.levelSet.electroChem.matplotlibSurfactantViewer*), 1077
- MatplotlibVectorViewer (class in *fiPy.viewers.matplotlibViewer.matplotlibVectorViewer*), 891
- MatplotlibViewer() (in module *fiPy.viewers.matplotlibViewer*), 865
- matrix (*fiPy.terms.advectionTerm.AdvectionTerm* property), 486
- matrix (*fiPy.terms.cellTerm.CellTerm* property), 491
- matrix
  - (*fiPy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* property), 491

- property*), 496
- matrix** (*fiPy.terms.diffusionTerm.DiffusionTerm property*), 499
- matrix** (*fiPy.terms.diffusionTermCorrection.DiffusionTermCorrection property*), 503
- matrix** (*fiPy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection property*), 506
- matrix** (*fiPy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm property*), 510
- matrix** (*fiPy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm property*), 515
- matrix** (*fiPy.terms.exponentialConvectionTerm.ExponentialConvectionTerm property*), 520
- matrix** (*fiPy.terms.faceTerm.FaceTerm property*), 524
- matrix** (*fiPy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm property*), 530
- matrix** (*fiPy.terms.hybridConvectionTerm.HybridConvectionTerm property*), 535
- matrix** (*fiPy.terms.implicitSourceTerm.ImplicitSourceTerm property*), 539
- matrix** (*fiPy.terms.powerLawConvectionTerm.PowerLawConvectionTerm property*), 544
- matrix** (*fiPy.terms.residualTerm.ResidualTerm property*), 548
- matrix** (*fiPy.terms.sourceTerm.SourceTerm property*), 552
- matrix** (*fiPy.terms.term.Term property*), 556
- matrix** (*fiPy.terms.transientTerm.TransientTerm property*), 561
- matrix** (*fiPy.terms.upwindConvectionTerm.UpwindConvectionTerm property*), 566
- matrix** (*fiPy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm property*), 571
- MatrixIllConditionedWarning**, 454
- max()** (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable method*), 645
- max()** (*fiPy.variables.cellVariable.CellVariable method*), 661
- max()** (*fiPy.variables.distanceVariable.DistanceVariable method*), 679
- max()** (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 695
- max()** (*fiPy.variables.faceVariable.FaceVariable method*), 707
- max()** (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 722
- max()** (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 738
- max()** (*fiPy.variables.histogramVariable.HistogramVariable method*), 753
- max()** (*fiPy.variables.meshVariable.MeshVariable method*), 765
- max()** (*fiPy.variables.modularVariable.ModularVariable method*), 779
- max()** (*fiPy.variables.noiseVariable.NoiseVariable method*), 794
- max()** (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 806
- max()** (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 818
- max()** (*fiPy.variables.surfactantVariable.SurfactantVariable method*), 832
- max()** (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 848
- max()** (*fiPy.variables.variable.Variable method*), 859
- MaxAll()** (*fiPy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper method*), 460
- MaximumIterationWarning**, 454
- MayaVi**, 108
- Mayavi**, 108
- MayaviClient** (*class in fiPy.viewers.mayaViViewer*), 894
- MayaviClient** (*class in fiPy.viewers.mayaViViewer.mayaViClient*), 898
- MayaviDaemon** (*class in fiPy.viewers.mayaViViewer.mayaViDaemon*), 901
- MayaviSurfactantViewer**, 1072
- MayaviSurfactantViewer** (*class in examples.levelSet.electroChem.mayaViSurfactantViewer*), 1081
- Mesh** (*class in fiPy.meshes.mesh*), 223
- Mesh1D** (*class in fiPy.meshes.mesh1D*), 232
- Mesh2D** (*class in fiPy.meshes.mesh2D*), 240
- MeshAdditionError**, 139, 231
- MeshDimensionError**, 863
- MeshExportError**, 221
- MeshVariable** (*class in fiPy.variables.meshVariable*), 757
- method1()** (*package.subpackage.base.Base method*), 123
- method1()** (*package.subpackage.object.Object method*), 124
- method2()** (*package.subpackage.base.Base method*), 123
- method2()** (*package.subpackage.object.Object method*), 124
- min()** (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable method*), 646
- min()** (*fiPy.variables.cellVariable.CellVariable method*), 661
- min()** (*fiPy.variables.distanceVariable.DistanceVariable method*), 679
- min()** (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method*), 695
- min()** (*fiPy.variables.faceVariable.FaceVariable method*), 707
- min()** (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable method*), 722
- min()** (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable method*), 738
- min()** (*fiPy.variables.histogramVariable.HistogramVariable method*), 753
- min()** (*fiPy.variables.meshVariable.MeshVariable method*), 765
- min()** (*fiPy.variables.modularVariable.ModularVariable method*), 779
- min()** (*fiPy.variables.noiseVariable.NoiseVariable method*), 794
- min()** (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method*), 806
- min()** (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method*), 818
- min()** (*fiPy.variables.surfactantVariable.SurfactantVariable method*), 832
- min()** (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable method*), 848
- min()** (*fiPy.variables.variable.Variable method*), 859
- MinAll()** (*fiPy.solvers.trilinos.comms.parallelEpetraCommWrapper.ParallelEpetraCommWrapper method*), 460
- minmodFaceValue** (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable property*), 646
- minmodFaceValue** (*fiPy.variables.cellVariable.CellVariable property*), 661
- minmodFaceValue** (*fiPy.variables.distanceVariable.DistanceVariable property*), 679
- minmodFaceValue** (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable property*), 695
- minmodFaceValue** (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable property*), 722
- minmodFaceValue** (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable property*), 738
- minmodFaceValue** (*fiPy.variables.histogramVariable.HistogramVariable*

- property*), 753
- `minmodFaceValue` (*fipy.variables.modularVariable.ModularVariable* *property*), 779
- `minmodFaceValue` (*fipy.variables.noiseVariable.NoiseVariable* *property*), 794
- `minmodFaceValue` (*fipy.variables.surfactantVariable.SurfactantVariable* *property*), 832
- `minmodFaceValue` (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* *property*), 848
- `ModularVariable`, 1120
- `ModularVariable` (*class in fipy.variables.modularVariable*), 767
- `module`
  - `examples`, 919
  - `examples.benchmarking`, 920
  - `examples.benchmarking.benchmarkmarker`, 920
  - `examples.benchmarking.size`, 920
  - `examples.benchmarking.steps`, 920
  - `examples.benchmarking.utils`, 920
  - `examples.benchmarking.versions`, 920
  - `examples.cahnHilliard`, 920
  - `examples.cahnHilliard.mesh2D`, 921
  - `examples.cahnHilliard.mesh2DCoupled`, 923
  - `examples.cahnHilliard.mesh3D`, 926
  - `examples.cahnHilliard.sphere`, 928
  - `examples.cahnHilliard.sphereDaemon`, 930
  - `examples.cahnHilliard.tanh1D`, 930
  - `examples.cahnHilliard.test`, 932
  - `examples.chemotaxis`, 932
  - `examples.chemotaxis.input`, 933
  - `examples.chemotaxis.input2D`, 935
  - `examples.chemotaxis.parameters`, 938
  - `examples.chemotaxis.test`, 938
  - `examples.convection`, 938
  - `examples.convection.advection`, 938
  - `examples.convection.advection.explicitUpwind`, 939
  - `examples.convection.advection.implicitUpwind`, 939
  - `examples.convection.advection.vanLeerUpwind`, 939
  - `examples.convection.exponential1D`, 939
  - `examples.convection.exponential1D.cylindricalMesh1D`, 940
  - `examples.convection.exponential1D.cylindricalMesh1DNonUniform`, 941
  - `examples.convection.exponential1D.mesh1D`, 943
  - `examples.convection.exponential1D.tri2D`, 944
  - `examples.convection.exponential1DBack`, 945
  - `examples.convection.exponential1DBack.mesh1D`, 945
  - `examples.convection.exponential1DSource`, 946
  - `examples.convection.exponential1DSource.mesh1D`, 947
  - `examples.convection.exponential1DSource.tri2D`, 948
  - `examples.convection.exponential2D`, 949
  - `examples.convection.exponential2D.cylindricalMesh2D`, 949
  - `examples.convection.exponential2D.cylindricalMesh2DNonUniform`, 951
  - `examples.convection.exponential2D.mesh2D`, 952
  - `examples.convection.exponential2D.tri2D`, 953
  - `examples.convection.pecllet`, 954
  - `examples.convection.powerLaw1D`, 955
  - `examples.convection.powerLaw1D.mesh1D`, 956
  - `examples.convection.powerLaw1D.tri2D`, 957
  - `examples.convection.robin`, 958
  - `examples.convection.source`, 960
  - `examples.convection.test`, 961
  - `examples.diffusion`, 961
  - `examples.diffusion.anisotropy`, 961
  - `examples.diffusion.circle`, 963
  - `examples.diffusion.circleQuad`, 968
  - `examples.diffusion.coupled`, 973
  - `examples.diffusion.electrostatics`, 975
  - `examples.diffusion.explicit`, 979
  - `examples.diffusion.explicit.mesh1D`, 980
  - `examples.diffusion.explicit.mixedelement`, 981
  - `examples.diffusion.explicit.test`, 982
  - `examples.diffusion.explicit.tri2D`, 982
  - `examples.diffusion.mesh1D`, 983
  - `examples.diffusion.mesh20x20`, 1003
  - `examples.diffusion.mesh20x20Coupled`, 1006
  - `examples.diffusion.nthOrder`, 1009
  - `examples.diffusion.nthOrder.input4thOrder1D`, 1010
  - `examples.diffusion.nthOrder.input4thOrder_line`, 1012
  - `examples.diffusion.nthOrder.test`, 1012
  - `examples.diffusion.steadyState`, 1012
  - `examples.diffusion.steadyState.mesh1D`, 1012
  - `examples.diffusion.steadyState.mesh1D.inputPeriodic`, 1013
  - `examples.diffusion.steadyState.mesh1D.tri2Dinput`, 1013
  - `examples.diffusion.steadyState.mesh20x20`, 1014
  - `examples.diffusion.steadyState.mesh20x20.gmshinput`, 1014
  - `examples.diffusion.steadyState.mesh20x20.isotropy`, 1014
  - `examples.diffusion.steadyState.mesh20x20.modifiedMeshInput`, 1015
  - `examples.diffusion.steadyState.mesh20x20.orthoerror`, 1016
  - `examples.diffusion.steadyState.mesh20x20.tri2Dinput`, 1017
  - `examples.diffusion.steadyState.mesh50x50`, 1017
  - `examples.diffusion.steadyState.mesh50x50.input`, 1017
  - `examples.diffusion.steadyState.mesh50x50.tri2Dinput`, 1017
  - `examples.diffusion.steadyState.otherMeshes`, 1018
  - `examples.diffusion.steadyState.otherMeshes.cubicalProblem`, 1018
  - `examples.diffusion.steadyState.otherMeshes.grid3Dinput`, 1018
  - `examples.diffusion.steadyState.otherMeshes.prism`, 1018
  - `examples.diffusion.steadyState.test`, 1019
  - `examples.diffusion.test`, 1019
  - `examples.diffusion.variable`, 1019
  - `examples.elphf`, 1020
  - `examples.elphf.diffusion`, 1021
  - `examples.elphf.diffusion.mesh1D`, 1021
  - `examples.elphf.diffusion.mesh1Ddimensional`, 1024
  - `examples.elphf.diffusion.mesh2D`, 1026
  - `examples.elphf.input`, 1029
  - `examples.elphf.phase`, 1035
  - `examples.elphf.phaseDiffusion`, 1038
  - `examples.elphf.poisson`, 1046
  - `examples.elphf.test`, 1049
  - `examples.flow`, 1049
  - `examples.flow.stokesCavity`, 1050
  - `examples.flow.test`, 1054
  - `examples.levelSet`, 1054
  - `examples.levelSet.advection`, 1055
  - `examples.levelSet.advection.circle`, 1055
  - `examples.levelSet.advection.mesh1D`, 1057
  - `examples.levelSet.advection.test`, 1058
  - `examples.levelSet.advection.trench`, 1058
  - `examples.levelSet.distanceFunction`, 1060

examples.levelSet.distanceFunction.circle, 1060  
 examples.levelSet.distanceFunction.interior, 1061  
 examples.levelSet.distanceFunction.mesh1D, 1062  
 examples.levelSet.distanceFunction.square, 1063  
 examples.levelSet.distanceFunction.test, 1064  
 examples.levelSet.electroChem, 1064  
 examples.levelSet.electroChem.adsorbingSurfactantEquation, 1064  
 examples.levelSet.electroChem.adsorption, 1064  
 examples.levelSet.electroChem.gapFillDistanceVariable, 1065  
 examples.levelSet.electroChem.gapFillMesh, 1065  
 examples.levelSet.electroChem.gold, 1066  
 examples.levelSet.electroChem.howToWriteAScript, 1067  
 examples.levelSet.electroChem.leveler, 1073  
 examples.levelSet.electroChem.lines, 1077  
 examples.levelSet.electroChem.matplotlibSurfactantViewer, 1077  
 examples.levelSet.electroChem.mayaviSurfactantViewer, 1081  
 examples.levelSet.electroChem.metalIonDiffusionEquation, 1084  
 examples.levelSet.electroChem.simpleTrenchSystem, 1084  
 examples.levelSet.electroChem.surfactantBulkDiffusionEquation, 1087  
 examples.levelSet.electroChem.test, 1087  
 examples.levelSet.electroChem.trenchMesh, 1087  
 examples.levelSet.surfactant, 1087  
 examples.levelSet.surfactant.circle, 1088  
 examples.levelSet.surfactant.expandingCircle, 1088  
 examples.levelSet.surfactant.square, 1089  
 examples.levelSet.surfactant.test, 1090  
 examples.levelSet.test, 1090  
 examples.meshing, 1090  
 examples.meshing.gmshRefinement, 1090  
 examples.meshing.inputGrid2D, 1090  
 examples.meshing.sphere, 1091  
 examples.meshing.test, 1092  
 examples.parallel, 1092  
 examples.phase, 1092  
 examples.phase.anisotropy, 1093  
 examples.phase.anisotropyOLD, 1097  
 examples.phase.binary, 1100  
 examples.phase.binaryCoupled, 1109  
 examples.phase.impingement, 1119  
 examples.phase.impingement.mesh20x20, 1119  
 examples.phase.impingement.mesh40x1, 1123  
 examples.phase.impingement.test, 1126  
 examples.phase.missOrientation, 1126  
 examples.phase.missOrientation.circle, 1126  
 examples.phase.missOrientation.mesh1D, 1127  
 examples.phase.missOrientation.modCircle, 1128  
 examples.phase.missOrientation.test, 1128  
 examples.phase.polyxtal, 1128  
 examples.phase.polyxtalCoupled, 1134  
 examples.phase.quaternary, 1140  
 examples.phase.simple, 1147  
 examples.phase.symmetry, 1155  
 examples.phase.test, 1157  
 examples.reactiveWetting, 1157  
 examples.reactiveWetting.liquidVapor1D, 1157  
 examples.reactiveWetting.liquidVapor2D, 1162  
 examples.reactiveWetting.test, 1166  
 examples.riemann, 1166  
 examples.riemann.acoustics, 1166  
 examples.riemann.test, 1167  
 examples.test, 1167  
 examples.updating, 1167  
 examples.updating.update0\_1to1\_0, 1167  
 examples.updating.update1\_0to2\_0, 1171  
 examples.updating.update2\_0to3\_0, 1175  
 fipy, 125  
 fipy.boundaryConditions, 126  
 fipy.boundaryConditions.boundaryCondition, 127  
 fipy.boundaryConditions.constraint, 127  
 fipy.boundaryConditions.fixedFlux, 128  
 fipy.boundaryConditions.fixedValue, 128  
 fipy.boundaryConditions.nthOrderBoundaryCondition, 129  
 fipy.boundaryConditions.test, 129  
 fipy.matrices, 130  
 fipy.matrices.offsetSparseMatrix, 130  
 fipy.matrices.petscMatrix, 130  
 fipy.matrices.pysparseMatrix, 130  
 fipy.matrices.scipyMatrix, 130  
 fipy.matrices.sparseMatrix, 130  
 fipy.matrices.test, 130  
 fipy.matrices.trilinosMatrix, 130  
 fipy.meshes, 130  
 fipy.meshes.abstractMesh, 132  
 fipy.meshes.builders, 139  
 fipy.meshes.builders.abstractGridBuilder, 140  
 fipy.meshes.builders.grid1DBuilder, 140  
 fipy.meshes.builders.grid2DBuilder, 140  
 fipy.meshes.builders.grid3DBuilder, 140  
 fipy.meshes.builders.periodicGrid1DBuilder, 140  
 fipy.meshes.builders.utilityClasses, 140  
 fipy.meshes.cylindricalGrid1D, 140  
 fipy.meshes.cylindricalGrid2D, 140  
 fipy.meshes.cylindricalNonUniformGrid1D, 140  
 fipy.meshes.cylindricalNonUniformGrid2D, 149  
 fipy.meshes.cylindricalUniformGrid1D, 158  
 fipy.meshes.cylindricalUniformGrid2D, 165  
 fipy.meshes.factoryMeshes, 172  
 fipy.meshes.gmshMesh, 175  
 fipy.meshes.grid1D, 223  
 fipy.meshes.grid2D, 223  
 fipy.meshes.grid3D, 223  
 fipy.meshes.mesh, 223  
 fipy.meshes.mesh1D, 232  
 fipy.meshes.mesh2D, 240  
 fipy.meshes.nonUniformGrid1D, 249  
 fipy.meshes.nonUniformGrid2D, 257  
 fipy.meshes.nonUniformGrid3D, 266  
 fipy.meshes.periodicGrid1D, 274  
 fipy.meshes.periodicGrid2D, 283  
 fipy.meshes.periodicGrid3D, 309  
 fipy.meshes.representations, 365  
 fipy.meshes.representations.abstractRepresentation, 365  
 fipy.meshes.representations.gridRepresentation, 365  
 fipy.meshes.representations.meshRepresentation, 365  
 fipy.meshes.skewedGrid2D, 365  
 fipy.meshes.sphericalNonUniformGrid1D, 374  
 fipy.meshes.sphericalUniformGrid1D, 383  
 fipy.meshes.test, 390  
 fipy.meshes.topologies, 390  
 fipy.meshes.topologies.abstractTopology, 390  
 fipy.meshes.topologies.gridTopology, 390  
 fipy.meshes.topologies.meshTopology, 390  
 fipy.meshes.tri2D, 390  
 fipy.meshes.uniformGrid, 399  
 fipy.meshes.uniformGrid1D, 406  
 fipy.meshes.uniformGrid2D, 413

fipy.meshes.uniformGrid3D, 420  
 fipy.numerix, 1169  
 fipy.solvers, 427  
 fipy.solvers.petsc, 429  
 fipy.solvers.petsc.comms, 430  
 fipy.solvers.petsc.comms.parallelPETScCommWrapper, 430  
 fipy.solvers.petsc.comms.petscCommWrapper, 430  
 fipy.solvers.petsc.comms.serialPETScCommWrapper, 431  
 fipy.solvers.petsc.dummySolver, 431  
 fipy.solvers.petsc.linearBicgSolver, 431  
 fipy.solvers.petsc.linearCGSSolver, 432  
 fipy.solvers.petsc.linearGMRESSolver, 432  
 fipy.solvers.petsc.linearLUSolver, 433  
 fipy.solvers.petsc.linearPCGSolver, 433  
 fipy.solvers.petsc.petscKrylovSolver, 434  
 fipy.solvers.petsc.petscSolver, 435  
 fipy.solvers.pyAMG, 435  
 fipy.solvers.pyAMG.linearCGSSolver, 436  
 fipy.solvers.pyAMG.linearGeneralSolver, 437  
 fipy.solvers.pyAMG.linearGMRESSolver, 436  
 fipy.solvers.pyAMG.linearLUSolver, 438  
 fipy.solvers.pyAMG.linearPCGSolver, 438  
 fipy.solvers.pyAMG.preconditioners, 439  
 fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner, 439  
 fipy.solvers.pyamgx, 439  
 fipy.solvers.pyamgx.aggregationAMGSolver, 440  
 fipy.solvers.pyamgx.classicalAMGSolver, 440  
 fipy.solvers.pyamgx.linearBiCGStabSolver, 441  
 fipy.solvers.pyamgx.linearCGSolver, 441  
 fipy.solvers.pyamgx.linearFGMRESSolver, 442  
 fipy.solvers.pyamgx.linearGMRESSolver, 443  
 fipy.solvers.pyamgx.linearLUSolver, 443  
 fipy.solvers.pyamgx.preconditioners, 444  
 fipy.solvers.pyamgx.preconditioners.preconditioners, 444  
 fipy.solvers.pyamgx.pyAMGXSolver, 444  
 fipy.solvers.pyamgx.smoothers, 445  
 fipy.solvers.pyamgx.smoothers.smoothers, 445  
 fipy.solvers.pysparse, 445  
 fipy.solvers.pysparse.linearCGSSolver, 446  
 fipy.solvers.pysparse.linearGMRESSolver, 446  
 fipy.solvers.pysparse.linearJORSolver, 447  
 fipy.solvers.pysparse.linearLUSolver, 447  
 fipy.solvers.pysparse.linearPCGSolver, 448  
 fipy.solvers.pysparse.preconditioners, 448  
 fipy.solvers.pysparse.preconditioners.jacobiPreconditioner, 448  
 fipy.solvers.pysparse.preconditioners.preconditioner, 449  
 fipy.solvers.pysparse.preconditioners.ssorPreconditioner, 449  
 fipy.solvers.pysparse.pysparseSolver, 449  
 fipy.solvers.pysparse.pysparseMatrixSolver, 450  
 fipy.solvers.scipy, 450  
 fipy.solvers.scipy.linearBicgstabSolver, 450  
 fipy.solvers.scipy.linearCGSSolver, 451  
 fipy.solvers.scipy.linearGMRESSolver, 451  
 fipy.solvers.scipy.linearLUSolver, 452  
 fipy.solvers.scipy.linearPCGSolver, 452  
 fipy.solvers.scipy.scipyKrylovSolver, 453  
 fipy.solvers.scipy.scipySolver, 453  
 fipy.solvers.solver, 453  
 fipy.solvers.test, 458  
 fipy.solvers.trilinos, 458  
 fipy.solvers.trilinos.comms, 459  
 fipy.solvers.trilinos.comms.epetraCommWrapper, 459  
 fipy.solvers.trilinos.comms.parallelEpetraCommWrapper, 460  
 fipy.solvers.trilinos.comms.serialEpetraCommWrapper, 460  
 fipy.solvers.trilinos.linearBicgstabSolver, 461  
 fipy.solvers.trilinos.linearCGSSolver, 461  
 fipy.solvers.trilinos.linearGMRESSolver, 462  
 fipy.solvers.trilinos.linearLUSolver, 462  
 fipy.solvers.trilinos.linearPCGSolver, 463  
 fipy.solvers.trilinos.preconditioners, 464  
 fipy.solvers.trilinos.preconditioners.domDecompPreconditioner, 464  
 fipy.solvers.trilinos.preconditioners.icPreconditioner, 464  
 fipy.solvers.trilinos.preconditioners.jacobiPreconditioner, 465  
 fipy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner, 465  
 fipy.solvers.trilinos.preconditioners.multilevelDDPreconditioner, 465  
 fipy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner, 466  
 fipy.solvers.trilinos.preconditioners.multilevelSAPreconditioner, 466  
 fipy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner, 466  
 fipy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner, 467  
 fipy.solvers.trilinos.preconditioners.preconditioner, 467  
 fipy.solvers.trilinos.trilinosAztec00Solver, 468  
 fipy.solvers.trilinos.trilinosMLTest, 469  
 fipy.solvers.trilinos.trilinosNonlinearSolver, 469  
 fipy.solvers.trilinos.trilinosSolver, 470  
 fipy.steps, 471  
 fipy.steps.pidStepper, 473  
 fipy.steps.pseudoRKQSStepper, 474  
 fipy.steps.stepper, 474  
 fipy.terms, 474  
 fipy.terms.abstractBinaryTerm, 481  
 fipy.terms.abstractConvectionTerm, 481  
 fipy.terms.abstractDiffusionTerm, 481  
 fipy.terms.abstractUpwindConvectionTerm, 481  
 fipy.terms.advectionTerm, 481  
 fipy.terms.asymmetricConvectionTerm, 488  
 fipy.terms.binaryTerm, 488  
 fipy.terms.cellTerm, 488  
 fipy.terms.centralDiffConvectionTerm, 492  
 fipy.terms.coupledBinaryTerm, 497  
 fipy.terms.diffusionTerm, 497  
 fipy.terms.diffusionTermCorrection, 501  
 fipy.terms.diffusionTermNoCorrection, 504  
 fipy.terms.explicitDiffusionTerm, 508  
 fipy.terms.explicitSourceTerm, 511  
 fipy.terms.explicitUpwindConvectionTerm, 511  
 fipy.terms.exponentialConvectionTerm, 516  
 fipy.terms.faceTerm, 522  
 fipy.terms.firstOrderAdvectionTerm, 526  
 fipy.terms.hybridConvectionTerm, 531  
 fipy.terms.implicitDiffusionTerm, 536  
 fipy.terms.implicitSourceTerm, 536  
 fipy.terms.nonDiffusionTerm, 540  
 fipy.terms.powerLawConvectionTerm, 540  
 fipy.terms.residualTerm, 546  
 fipy.terms.sourceTerm, 550  
 fipy.terms.term, 554  
 fipy.terms.test, 558

- fiPy.terms.transientTerm, 558
- fiPy.terms.unaryTerm, 563
- fiPy.terms.upwindConvectionTerm, 563
- fiPy.terms.vanLeerConvectionTerm, 568
- fiPy.testFiPy, 573
- fiPy.tests, 573
- fiPy.tests.doctestPlus, 573
- fiPy.tests.lateImportTest, 575
- fiPy.tests.test, 575
- fiPy.tests.testProgram, 576
- fiPy.tools, 576
- fiPy.tools.comms, 593
- fiPy.tools.comms.commWrapper, 594
- fiPy.tools.comms.dummyComm, 594
- fiPy.tools.debug, 594
- fiPy.tools.decorators, 595
- fiPy.tools.dimensions, 595
- fiPy.tools.dimensions.DictWithDefault, 595
- fiPy.tools.dimensions.NumberDict, 595
- fiPy.tools.dimensions.physicalField, 595
- fiPy.tools.dump, 621, 1122
- fiPy.tools.inline, 622
- fiPy.tools.logging, 622
- fiPy.tools.logging.environment, 622
- fiPy.tools.numerix, 623
- fiPy.tools.parser, 628, 1068, 1119
- fiPy.tools.sharedtempfile, 629
- fiPy.tools.test, 630
- fiPy.tools.timer, 630
- fiPy.tools.vector, 630
- fiPy.tools.version, 631
- fiPy.variables, 631
- fiPy.variables.addOverFacesVariable, 633
- fiPy.variables.arithmeticCellToFaceVariable, 633
- fiPy.variables.betaNoiseVariable, 633
- fiPy.variables.binaryOperatorVariable, 649
- fiPy.variables.cellToFaceVariable, 649
- fiPy.variables.cellVariable, 649
- fiPy.variables.constant, 664
- fiPy.variables.constraintMask, 664
- fiPy.variables.coupledCellVariable, 664
- fiPy.variables.distanceVariable, 664
- fiPy.variables.exponentialNoiseVariable, 683
- fiPy.variables.faceGradContributionsVariable, 698
- fiPy.variables.faceGradVariable, 698
- fiPy.variables.faceVariable, 698
- fiPy.variables.gammaNoiseVariable, 709
- fiPy.variables.minmodCellGradVariable, 725
- fiPy.variables.gaussianNoiseVariable, 725
- fiPy.variables.harmonicCellToFaceVariable, 742
- fiPy.variables.histogramVariable, 742
- fiPy.variables.interfaceAreaVariable, 756
- fiPy.variables.interfaceFlagVariable, 756
- fiPy.variables.leastSquaresCellGradVariable, 756
- fiPy.variables.levelSetDiffusionVariable, 756
- fiPy.variables.meshVariable, 756
- fiPy.variables.minmodCellToFaceVariable, 767
- fiPy.variables.modCellGradVariable, 767
- fiPy.variables.modCellToFaceVariable, 767
- fiPy.variables.modFaceGradVariable, 767
- fiPy.variables.modPhysicalField, 767
- fiPy.variables.modularVariable, 767
- fiPy.variables.noiseVariable, 782
- fiPy.variables.operatorVariable, 798
- fiPy.variables.scharfetterGummelFaceVariable, 798
- fiPy.variables.surfactantConvectionVariable, 809
- fiPy.variables.surfactantVariable, 820
- fiPy.variables.test, 836
- fiPy.variables.unaryOperatorVariable, 836
- fiPy.variables.uniformNoiseVariable, 836
- fiPy.variables.variable, 851
- fiPy.viewers, 861, 932, 941, 942, 944, 948, 951, 952, 985, 1004, 1007, 1011, 1052, 1099, 1106, 1115, 1121, 1125, 1147
- fiPy.viewers.matplotlibViewer, 865
- fiPy.viewers.matplotlibViewer.abstractMatplotlib2DViewer, 867
- fiPy.viewers.matplotlibViewer.abstractMatplotlibViewer, 870
- fiPy.viewers.matplotlibViewer.matplotlib1DViewer, 872
- fiPy.viewers.matplotlibViewer.matplotlib2DContourViewer, 875
- fiPy.viewers.matplotlibViewer.matplotlib2DGridContourViewer, 877
- fiPy.viewers.matplotlibViewer.matplotlib2DGridViewer, 881
- fiPy.viewers.matplotlibViewer.matplotlib2DViewer, 883
- fiPy.viewers.matplotlibViewer.matplotlibSparseMatrixViewer, 886
- fiPy.viewers.matplotlibViewer.matplotlibStreamViewer, 886
- fiPy.viewers.matplotlibViewer.matplotlibVectorViewer, 891
- fiPy.viewers.matplotlibViewer.test, 894
- fiPy.viewers.mayaviViewer, 894
- fiPy.viewers.mayaviViewer.mayaviClient, 898
- fiPy.viewers.mayaviViewer.mayaviDaemon, 901
- fiPy.viewers.mayaviViewer.test, 902
- fiPy.viewers.multiViewer, 902
- fiPy.viewers.test, 904
- fiPy.viewers.testinteractive, 904
- fiPy.viewers.tsvViewer, 904
- fiPy.viewers.viewer, 906
- fiPy.viewers.vtkViewer, 908
- fiPy.viewers.vtkViewer.test, 912
- fiPy.viewers.vtkViewer.vtkCellViewer, 912
- fiPy.viewers.vtkViewer.vtkFaceViewer, 914
- fiPy.viewers.vtkViewer.vtkViewer, 916
- gmsH, 1066, 1074, 1085
- package, 121
- package.subpackage, 121
- package.subpackage.base, 122
- package.subpackage.object, 123
- scipy, 966, 971, 1115, 1154
- viewers, 1145
- MPI, 108
- mpi4py, 108
- MSHFile (class in *fiPy.meshes.gmsHMesh*), 221
- MultilevelDDMLPreconditioner (class in *fiPy.solvers.trilinos.preconditioners.multilevelDDMLPreconditioner*), 465
- MultilevelDDPreconditioner (class in *fiPy.solvers.trilinos.preconditioners.multilevelDDPreconditioner*), 465
- MultilevelNSSAPreconditioner (class in *fiPy.solvers.trilinos.preconditioners.multilevelNSSAPreconditioner*), 466
- MultilevelSAPreconditioner (class in *fiPy.solvers.trilinos.preconditioners.multilevelSAPreconditioner*), 466
- MultilevelSGSPreconditioner (class in *fiPy.solvers.trilinos.preconditioners.multilevelSGSPreconditioner*), 466



**MultilevelSolverSmootherPreconditioner** (class in *fiPy.solvers.trilinos.preconditioners.multilevelSolverSmootherPreconditioner*), 467

**multiply()** (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 610

**multiply()** (*fiPy.tools.PhysicalField* method), 588

**MultiViewer** (class in *fiPy.viewers.multiViewer*), 902

## N

**name()** (*fiPy.tools.dimensions.physicalField.PhysicalUnit* method), 620

**nearest()** (in module *fiPy.tools.numerix*), 626

**nix\_info()** (in module *fiPy.tools.logging.environment*), 622

**NoiseVariable** (class in *fiPy.variables.noiseVariable*), 783

**NonUniformGrid1D** (class in *fiPy.meshes.nonUniformGrid1D*), 249

**NonUniformGrid2D** (class in *fiPy.meshes.nonUniformGrid2D*), 257

**NonUniformGrid3D** (class in *fiPy.meshes.nonUniformGrid3D*), 266

**NthOrderBoundaryCondition**, 931, 1010

**NthOrderBoundaryCondition** (class in *fiPy.boundaryConditions.nthOrderBoundaryCondition*), 129

**numarray**, 108

**Numeric**, 108

**numericValue** (*fiPy.tools.dimensions.physicalField.PhysicalField* property), 611

**numericValue** (*fiPy.tools.PhysicalField* property), 589

**NumPy**, 108

## O

**object**

- fiPy.terms.implicitDiffusionTerm.DiffusionTerm*, 964, 969
- fiPy.terms.transientTerm.TransientTerm*, 964, 969
- fiPy.variables.cellVariable.CellVariable*, 963, 968
- fiPy.viewers.tsvViewer.TSVViewer*, 965, 970

**Object** (class in package *subpackage.object*), 123

**OffsetSparseMatrix()** (in module *fiPy.matrices.offsetSparseMatrix*), 130

**old** (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* property), 646

**old** (*fiPy.variables.cellVariable.CellVariable* property), 661

**old** (*fiPy.variables.distanceVariable.DistanceVariable* property), 680

**old** (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 695

**old** (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 722

**old** (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 739

**old** (*fiPy.variables.histogramVariable.HistogramVariable* property), 754

**old** (*fiPy.variables.modularVariable.ModularVariable* property), 779

**old** (*fiPy.variables.noiseVariable.NoiseVariable* property), 795

**old** (*fiPy.variables.surfactantVariable.SurfactantVariable* property), 833

**old** (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 848

**OpenMP**, 108

**openMshFile()** (in module *fiPy.meshes.gmshMesh*), 222

**openPosFile()** (in module *fiPy.meshes.gmshMesh*), 223

## P

**package**

- module, 121

**package.subpackage**

- module, 121

**package.subpackage.base**

- module, 122

**package.subpackage.object**

module, 123

**package\_info()** (in module *fiPy.tools.logging.environment*), 622

**pandas**, 108

**parallelComm** (in module *fiPy.tools*), 576

**ParallelEpetraCommWrapper** (class in *fiPy.solvers.trilinos.comms.parallelEpetraCommWrapper*), 460

**ParallelPETScCommWrapper** (class in *fiPy.solvers.petsc.comms.parallelPETScCommWrapper*), 430

**parse()** (in module *fiPy.tools.parser*), 628

**parse\_command\_line()** (*fiPy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 902

**PeriodicGrid1D** (class in *fiPy.meshes.periodicGrid1D*), 274

**PeriodicGrid2D** (class in *fiPy.meshes.periodicGrid2D*), 283

**PeriodicGrid2DLeftRight** (class in *fiPy.meshes.periodicGrid2D*), 292

**PeriodicGrid2DTopBottom** (class in *fiPy.meshes.periodicGrid2D*), 301

**PeriodicGrid3D** (class in *fiPy.meshes.periodicGrid3D*), 309

**PeriodicGrid3DFrontBack** (class in *fiPy.meshes.periodicGrid3D*), 318

**PeriodicGrid3DLeftRight** (class in *fiPy.meshes.periodicGrid3D*), 326

**PeriodicGrid3DLeftRightFrontBack** (class in *fiPy.meshes.periodicGrid3D*), 334

**PeriodicGrid3DLeftRightTopBottom** (class in *fiPy.meshes.periodicGrid3D*), 342

**PeriodicGrid3DTopBottom** (class in *fiPy.meshes.periodicGrid3D*), 349

**PeriodicGrid3DTopBottomFrontBack** (class in *fiPy.meshes.periodicGrid3D*), 357

**PETSc**, 108

**petsc4py**, 108

**PETScCommWrapper** (class in *fiPy.solvers.petsc.comms.petscCommWrapper*), 430

**PETScKrylovSolver** (class in *fiPy.solvers.petsc.petscKrylovSolver*), 434

**PETScSolver** (class in *fiPy.solvers.petsc.petscSolver*), 435

**PhysicalField** (class in *fiPy.tools*), 576

**PhysicalField** (class in *fiPy.tools.dimensions.physicalField*), 598

**physicalShape** (*fiPy.meshes.skewedGrid2D.SkewedGrid2D* property), 373

**physicalShape** (*fiPy.meshes.tri2D.Tri2D* property), 398

**PhysicalUnit** (class in *fiPy.tools.dimensions.physicalField*), 614

**pi**, 1094, 1099, 1120, 1125

**PIDStepper** (class in *fiPy.steps.steps.pidStepper*), 473

**pip**, 108

**pip\_info()** (in module *fiPy.tools.logging.environment*), 622

**platform\_info()** (in module *fiPy.tools.logging.environment*), 623

**plot()** (example *ples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer* method), 1080

**plot()** (*fiPy.viewers.DummyViewer* method), 862

**plot()** (*fiPy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* method), 869

**plot()** (*fiPy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* method), 871

**plot()** (*fiPy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* method), 874

**plot()** (*fiPy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* method), 876

- plot() (fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.matplotlib2DGridContourViewer method), 879
  - plot() (fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer method), 882
  - plot() (fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer method), 885
  - plot() (fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer method), 889
  - plot() (fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer method), 893
  - plot() (fipy.viewers.mayaviViewer.MayaviClient method), 896
  - plot() (fipy.viewers.mayaviViewer.mayaviClient.MayaviClient method), 900
  - plot() (fipy.viewers.multiViewer.MultiViewer method), 902
  - plot() (fipy.viewers.tsvViewer.TSVViewer method), 905
  - plot() (fipy.viewers.viewer.AbstractViewer method), 907
  - plot() (fipy.viewers.vtkViewer.VTKCellViewer method), 909
  - plot() (fipy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer method), 913
  - plot() (fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer method), 910
  - plotMesh() (exam-  
ples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer method), 1080
  - plotMesh() (exam-  
ples.levelSet.electroChem.mayaviSurfactantViewer.MayaviSurfactantViewer method), 1083
  - plotMesh() (fipy.viewers.DummyViewer method), 862
  - plotMesh() (fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer method), 869
  - plotMesh() (fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer method), 871
  - plotMesh() (fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer method), 874
  - plotMesh() (fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer method), 876
  - plotMesh() (fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer method), 879
  - plotMesh() (fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer method), 882
  - plotMesh() (fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer method), 885
  - plotMesh() (fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer method), 890
  - plotMesh() (fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer method), 893
  - plotMesh() (fipy.viewers.mayaviViewer.MayaviClient method), 896
  - plotMesh() (fipy.viewers.mayaviViewer.mayaviClient.MayaviClient method), 900
  - plotMesh() (fipy.viewers.multiViewer.MultiViewer method), 902
  - plotMesh() (fipy.viewers.tsvViewer.TSVViewer method), 905
  - plotMesh() (fipy.viewers.viewer.AbstractViewer method), 907
  - plotMesh() (fipy.viewers.vtkViewer.VTKCellViewer method), 909
  - plotMesh() (fipy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer method), 913
  - plotMesh() (fipy.viewers.vtkViewer.VTKFaceViewer method), 910
  - plotMesh() (fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer method), 915
  - plotMesh() (fipy.viewers.vtkViewer.vtkViewer.VTKViewer method), 916
  - POSFile (class in fipy.meshes.gmshMesh), 222
  - PowerLawConvectionTerm, 1104, 1144
  - PowerLawConvectionTerm (class in fipy.terms.powerLawConvectionTerm), 540
  - Preconditioner (class in fipy.solvers.pyamgx.preconditioners.preconditioners), 444
  - Preconditioner (class in fipy.solvers.pysparse.preconditioners.preconditioner), 449
  - Preconditioner (class in fipy.solvers.trilinos.preconditioners.preconditioner), 467
  - PreconditionerNotPositiveDefiniteWarning, 455
  - PreconditionerWarning, 456
  - PRINT() (in module fipy.tools.debug), 594
  - prune() (in module fipy.tools.vector), 631
  - PseudoRKQSStepper (class in fipy.steps.pseudoRKQSStepper), 474
  - put() (fipy.tools.dimensions.physicalField.PhysicalField method), 611
  - put() (fipy.tools.PhysicalField method), 589
  - put() (in module fipy.tools.numerix), 626
  - putAdd() (in module fipy.tools.vector), 631
  - PyAMG, 108
  - pyamgx, 109
  - PyAMGXSolver (class in fipy.solvers.pyamgx.pyAMGXSolver), 444
  - PyPI, 109
  - Pyrex, 109
  - PySparse, 449
  - PysparseSolver (class in fipy.solvers.pysparse.pysparseSolver), 449
  - Python, 109
  - Python 3, 109
  - Python Enhancement Proposals PEP 3000, 109
  - PyTrilinos, 109
  - PyVTK, 109
- ## R
- rank() (in module fipy.tools.numerix), 627
  - rdot() (fipy.variables.betaNoiseVariable.BetaNoiseVariable method), 646
  - rdot() (fipy.variables.cellVariable.CellVariable method), 662
  - rdot() (fipy.variables.distanceVariable.DistanceVariable method), 680
  - rdot() (fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable method), 696
  - rdot() (fipy.variables.faceVariable.FaceVariable method), 707
  - rdot() (fipy.variables.gammaNoiseVariable.GammaNoiseVariable method), 723
  - rdot() (fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable method), 739
  - rdot() (fipy.variables.histogramVariable.HistogramVariable method), 754
  - rdot() (fipy.variables.meshVariable.MeshVariable method), 765
  - rdot() (fipy.variables.modularVariable.ModularVariable method), 780
  - rdot() (fipy.variables.noiseVariable.NoiseVariable method), 795
  - rdot() (fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable method), 806
  - rdot() (fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable method), 818

**rdot()** (*fiPy.variables.surfactantVariable.SurfactantVariable* method), 833  
**rdot()** (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 849  
**read()** (*fiPy.meshes.gmshMesh.MSHFile* method), 221  
**read()** (in module *fiPy.tools.dump*), 621  
**register\_skipper()** (in module *fiPy.tests.doctestPlus*), 573  
**release()** (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* method), 647  
**release()** (*fiPy.variables.cellVariable.CellVariable* method), 662  
**release()** (*fiPy.variables.distanceVariable.DistanceVariable* method), 680  
**release()** (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 696  
**release()** (*fiPy.variables.faceVariable.FaceVariable* method), 707  
**release()** (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 723  
**release()** (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 739  
**release()** (*fiPy.variables.histogramVariable.HistogramVariable* method), 754  
**release()** (*fiPy.variables.meshVariable.MeshVariable* method), 765  
**release()** (*fiPy.variables.modularVariable.ModularVariable* method), 780  
**release()** (*fiPy.variables.noiseVariable.NoiseVariable* method), 795  
**release()** (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 807  
**release()** (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 818  
**release()** (*fiPy.variables.surfactantVariable.SurfactantVariable* method), 833  
**release()** (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 849  
**release()** (*fiPy.variables.variable.Variable* method), 859  
**report\_skips()** (in module *fiPy.tests.doctestPlus*), 573  
**reshape()** (*fiPy.tools.dimensions.physicalField.PhysicalField* method), 611  
**reshape()** (*fiPy.tools.physicalField.PhysicalField* method), 589  
**reshape()** (in module *fiPy.tools.numerix*), 627  
**residual()** (in module *fiPy.steps*), 472  
**ResidualTerm** (class in *fiPy.terms.residualTerm*), 546  
**residualVectorAndNorm()** (*fiPy.terms.advectionTerm.AdvectionTerm* method), 486  
**residualVectorAndNorm()** (*fiPy.terms.cellTerm.CellTerm* method), 491  
**residualVectorAndNorm()** (*fiPy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 496  
**residualVectorAndNorm()** (*fiPy.terms.diffusionTerm.DiffusionTerm* method), 500  
**residualVectorAndNorm()** (*fiPy.terms.diffusionTermCorrection.DiffusionTermCorrection* method), 503  
**residualVectorAndNorm()** (*fiPy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* method), 506  
**residualVectorAndNorm()** (*fiPy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* method), 510  
**residualVectorAndNorm()** (*fiPy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 515  
**residualVectorAndNorm()** (*fiPy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 520  
**residualVectorAndNorm()** (*fiPy.terms.faceTerm.FaceTerm* method), 525  
**residualVectorAndNorm()** (*fiPy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 530  
**residualVectorAndNorm()** (*fiPy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 535  
**residualVectorAndNorm()** (*fiPy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 539  
**residualVectorAndNorm()** (*fiPy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 544  
**residualVectorAndNorm()** (*fiPy.terms.residualTerm.ResidualTerm* method), 548  
**residualVectorAndNorm()** (*fiPy.terms.sourceTerm.SourceTerm* method), 553  
**residualVectorAndNorm()** (*fiPy.terms.term.Term* method), 556  
**residualVectorAndNorm()** (*fiPy.terms.transientTerm.TransientTerm* method), 561  
**residualVectorAndNorm()** (*fiPy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 566  
**residualVectorAndNorm()** (*fiPy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 571  
**RHSvector**, 1052  
**RHSvector** (*fiPy.terms.advectionTerm.AdvectionTerm* property), 484  
**RHSvector** (*fiPy.terms.cellTerm.CellTerm* property), 488  
**RHSvector** (*fiPy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* property), 493  
**RHSvector** (*fiPy.terms.diffusionTerm.DiffusionTerm* property), 498  
**RHSvector** (*fiPy.terms.diffusionTermCorrection.DiffusionTermCorrection* property), 501  
**RHSvector** (*fiPy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* property), 504  
**RHSvector** (*fiPy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* property), 508  
**RHSvector** (*fiPy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* property), 513  
**RHSvector** (*fiPy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* property), 518  
**RHSvector** (*fiPy.terms.faceTerm.FaceTerm* property), 522  
**RHSvector** (*fiPy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* property), 527  
**RHSvector** (*fiPy.terms.hybridConvectionTerm.HybridConvectionTerm* property), 532  
**RHSvector** (*fiPy.terms.implicitSourceTerm.ImplicitSourceTerm* property), 537  
**RHSvector** (*fiPy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* property), 542  
**RHSvector** (*fiPy.terms.residualTerm.ResidualTerm* property), 546  
**RHSvector** (*fiPy.terms.sourceTerm.SourceTerm* property), 550  
**RHSvector** (*fiPy.terms.term.Term* property), 555  
**RHSvector** (*fiPy.terms.transientTerm.TransientTerm* property), 559  
**RHSvector** (*fiPy.terms.upwindConvectionTerm.UpwindConvectionTerm* property), 564

RHSvector  
 (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* property), 569

runGold, 1066

runLeveler, 1073

runSimpleTrenchSystem, 1084

## S

ScalarQuantityOutOfRangeWarning, 456

ScharfetterGummelFaceVariable (class in *fipy.variables.scharfetterGummelFaceVariable*), 798

ScientificPython, 109

SciPy, 109, 1106

scipy  
 module, 966, 971, 1115, 1154

scramble() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 647

scramble() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 696

scramble() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 723

scramble() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 739

scramble() (*fipy.variables.noiseVariable.NoiseVariable* method), 795

scramble() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 849

serialComm (in module *fipy.tools*), 576

SerialEpetraCommWrapper (class in *fipy.solvers.trilinos.comms.serialEpetraCommWrapper*), 460

SerialPETScCommWrapper (class in *fipy.solvers.petsc.comms.serialPETScCommWrapper*), 431

SerialSolverError, 428

setLimits() (examples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer method), 1080

setLimits() (examples.levelSet.electroChem.mayaviSurfactantViewer.MayaviSurfactantViewer method), 1083

setLimits() (*fipy.viewers.DummyViewer* method), 862

setLimits() (*fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* method), 869

setLimits() (*fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* method), 871

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* method), 874

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* method), 876

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* method), 880

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* method), 882

setLimits() (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* method), 885

setLimits() (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* method), 890

setLimits() (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* method), 893

setLimits() (*fipy.viewers.mayaviViewer.MayaviClient* method), 897

setLimits() (*fipy.viewers.mayaviViewer.mayaviClient.MayaviClient* method), 900

setLimits() (*fipy.viewers.multiViewer.MultiViewer* method), 903

setLimits() (*fipy.viewers.tsvViewer.TSVViewer* method), 905

setLimits() (*fipy.viewers.viewer.AbstractViewer* method), 907

setLimits() (*fipy.viewers.vtkViewer.VTKCellViewer* method), 909

setLimits() (*fipy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer* method), 913

setLimits() (*fipy.viewers.vtkViewer.VTKFaceViewer* method), 910

setLimits() (*fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer* method), 915

setLimits() (*fipy.viewers.vtkViewer.vtkViewer.VTKViewer* method), 916

setName() (*fipy.tools.dimensions.physicalField.PhysicalUnit* method), 620

setup\_source() (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 902

setValue() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 647

setValue() (*fipy.variables.cellVariable.CellVariable* method), 662

setValue() (*fipy.variables.distanceVariable.DistanceVariable* method), 681

setValue() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 696

setValue() (*fipy.variables.faceVariable.FaceVariable* method), 707

setValue() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 723

setValue() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 740

setValue() (*fipy.variables.histogramVariable.HistogramVariable* method), 754

setValue() (*fipy.variables.meshVariable.MeshVariable* method), 765

setValue() (*fipy.variables.modularVariable.ModularVariable* method), 780

setValue() (*fipy.variables.noiseVariable.NoiseVariable* method), 796

setValue() (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 807

setValue() (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 819

setValue() (*fipy.variables.surfactantVariable.SurfactantVariable* method), 834

setValue() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 849

setValue() (*fipy.variables.variable.Variable* method), 859

shape (*fipy.tools.dimensions.physicalField.PhysicalField* property), 611

shape (*fipy.tools.PhysicalField* property), 589

shape (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 647

shape (*fipy.variables.cellVariable.CellVariable* property), 663

shape (*fipy.variables.distanceVariable.DistanceVariable* property), 681

shape (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* property), 697

shape (*fipy.variables.faceVariable.FaceVariable* property), 708

shape (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* property), 724

- shape (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* property), 740
- shape (*fipy.variables.histogramVariable.HistogramVariable* property), 755
- shape (*fipy.variables.meshVariable.MeshVariable* property), 766
- shape (*fipy.variables.modularVariable.ModularVariable* property), 781
- shape (*fipy.variables.noiseVariable.NoiseVariable* property), 796
- shape (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* property), 807
- shape (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* property), 819
- shape (*fipy.variables.surfactantVariable.SurfactantVariable* property), 834
- shape (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* property), 850
- shape (*fipy.variables.variable.Variable* property), 860
- SharedTemporaryFile() (in module *fipy.tools*), 592
- SharedTemporaryFile() (in module *fipy.tools.sharedtempfile*), 629
- sign() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 611
- sign() (*fipy.tools.PhysicalField* method), 589
- sin() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 612
- sin() (*fipy.tools.PhysicalField* method), 590
- sinh() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 612
- sinh() (*fipy.tools.PhysicalField* method), 590
- SkewedGrid2D (class in *fipy.meshes.skewedGrid2D*), 365
- SmoothedAggregationPreconditioner (class in *fipy.solvers.pyAMG.preconditioners.smoothedAggregationPreconditioner*), 439
- Smoother (class in *fipy.solvers.pyamg.smoothers.smoothers*), 445
- SolutionVariableNumberError, 477
- SolutionVariableRequiredError, 477
- solve, 1115
- solve() (*fipy.terms.advectionTerm.AdvectionTerm* method), 487
- solve() (*fipy.terms.cellTerm.CellTerm* method), 491
- solve() (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 496
- solve() (*fipy.terms.diffusionTerm.DiffusionTerm* method), 500
- solve() (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* method), 503
- solve() (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* method), 507
- solve() (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* method), 510
- solve() (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 515
- solve() (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 520
- solve() (*fipy.terms.faceTerm.FaceTerm* method), 525
- solve() (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 530
- solve() (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 535
- solve() (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 539
- solve() (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 545
- solve() (*fipy.terms.residualTerm.ResidualTerm* method), 549
- solve() (*fipy.terms.sourceTerm.SourceTerm* method), 553
- solve() (*fipy.terms.term.Term* method), 557
- solve() (*fipy.terms.transientTerm.TransientTerm* method), 562
- solve() (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 567
- solve() (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 572
- Solver (class in *fipy.solvers.solver*), 457
- SolverConvergenceWarning, 457
- SourceTerm (class in *fipy.terms.sourceTerm*), 550
- SphericalGrid1D() (in module *fipy.meshes.factory.Meshes*), 174
- SphericalNonUniformGrid1D (class in *fipy.meshes.sphericalNonUniformGrid1D*), 374
- SphericalUniformGrid1D (class in *fipy.meshes.sphericalUniformGrid1D*), 383
- Sphinx, 109
- sqrt, 932, 1068, 1119, 1148
  - arcsin; cos, 966, 971
- sqrt() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 612
- sqrt() (*fipy.tools.PhysicalField* method), 590
- sqrtDot() (in module *fipy.tools.numerix*), 627
- SsorPreconditioner (class in *fipy.solvers.pysparse.preconditioners.ssorPreconditioner*), 449
- StagnatedSolverWarning, 458
- std() (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* method), 648
- std() (*fipy.variables.cellVariable.CellVariable* method), 663
- std() (*fipy.variables.distanceVariable.DistanceVariable* method), 682
- std() (*fipy.variables.exponentialNoiseVariable.ExponentialNoiseVariable* method), 697
- std() (*fipy.variables.faceVariable.FaceVariable* method), 708
- std() (*fipy.variables.gammaNoiseVariable.GammaNoiseVariable* method), 724
- std() (*fipy.variables.gaussianNoiseVariable.GaussianNoiseVariable* method), 741
- std() (*fipy.variables.histogramVariable.HistogramVariable* method), 755
- std() (*fipy.variables.meshVariable.MeshVariable* method), 766
- std() (*fipy.variables.modularVariable.ModularVariable* method), 781
- std() (*fipy.variables.noiseVariable.NoiseVariable* method), 797
- std() (*fipy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable* method), 808
- std() (*fipy.variables.surfactantConvectionVariable.SurfactantConvectionVariable* method), 820
- std() (*fipy.variables.surfactantVariable.SurfactantVariable* method), 835
- std() (*fipy.variables.uniformNoiseVariable.UniformNoiseVariable* method), 850
- std() (*fipy.variables.variable.Variable* method), 860
- SteadyConvectionDiffusionScEquation, 1168
- Stepper (class in *fipy.steps.steps*), 474
- subtract() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 612
- subtract() (*fipy.tools.PhysicalField* method), 590
- sum() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 613
- sum() (*fipy.tools.PhysicalField* method), 591
- sum() (in module *fipy.tools.numerix*), 627
- SurfactantConvectionVariable (class in *fipy.variables.surfactantConvectionVariable*), 809
- SurfactantVariable, 1070
- SurfactantVariable (class in *fipy.variables.surfactantVariable*), 820
- sweep, 1052, 1115, 1153
- sweep() (*fipy.terms.advectionTerm.AdvectionTerm* method), 487

sweep() (*fipy.terms.cellTerm.CellTerm* method), 491

sweep() (*fipy.terms.centralDiffConvectionTerm.CentralDifferenceConvectionTerm* method), 496

sweep() (*fipy.terms.diffusionTerm.DiffusionTerm* method), 500

sweep() (*fipy.terms.diffusionTermCorrection.DiffusionTermCorrection* method), 504

sweep() (*fipy.terms.diffusionTermNoCorrection.DiffusionTermNoCorrection* method), 507

sweep() (*fipy.terms.explicitDiffusionTerm.ExplicitDiffusionTerm* method), 510

sweep() (*fipy.terms.explicitUpwindConvectionTerm.ExplicitUpwindConvectionTerm* method), 516

sweep() (*fipy.terms.exponentialConvectionTerm.ExponentialConvectionTerm* method), 521

sweep() (*fipy.terms.faceTerm.FaceTerm* method), 525

sweep() (*fipy.terms.firstOrderAdvectionTerm.FirstOrderAdvectionTerm* method), 530

sweep() (*fipy.terms.hybridConvectionTerm.HybridConvectionTerm* method), 535

sweep() (*fipy.terms.implicitSourceTerm.ImplicitSourceTerm* method), 540

sweep() (*fipy.terms.powerLawConvectionTerm.PowerLawConvectionTerm* method), 545

sweep() (*fipy.terms.residualTerm.ResidualTerm* method), 549

sweep() (*fipy.terms.sourceTerm.SourceTerm* method), 553

sweep() (*fipy.terms.term.Term* method), 557

sweep() (*fipy.terms.transientTerm.TransientTerm* method), 562

sweep() (*fipy.terms.upwindConvectionTerm.UpwindConvectionTerm* method), 567

sweep() (*fipy.terms.vanLeerConvectionTerm.VanLeerConvectionTerm* method), 572

sweepMonotonic() (in module *fipy.steps*), 472

## T

take() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 613

take() (*fipy.tools.PhysicalField* method), 591

take() (in module *fipy.tools.numerix*), 627

tan, 1094, 1099

tan() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 613

tan() (*fipy.tools.PhysicalField* method), 591

tanh, 1148

tanh() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 613

tanh() (*fipy.tools.PhysicalField* method), 591

Term (class in *fipy.terms.term*), 554

TermMultiplyError, 478

test (class in *fipy.tests.test*), 575

test() (in module *fipy*), 125

testmod() (in module *fipy.tests.doctestPlus*), 573

TestProgram (class in *fipy.tests.testProgram*), 576

Timer (class in *fipy.tools.timer*), 630

title (example-  
*fipy.viewers.matplotlibViewer.matplotlibSurfactantViewer.MatplotlibSurfactantViewer* property), 1081

title (example-  
*fipy.viewers.matplotlibViewer.matplotlibSurfactantViewer.MatplotlibSurfactantViewer* property), 1084

title (*fipy.viewers.DummyViewer* property), 863

title (*fipy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2DViewer* property), 870

title (*fipy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewer* property), 872

title (*fipy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer* property), 875

title (*fipy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContourViewer* property), 877

title (*fipy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGridContourViewer* property), 880

title (*fipy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewer* property), 883

title (*fipy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer* property), 886

title (*fipy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewer* property), 890

title (*fipy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewer* property), 894

title (*fipy.viewers.mayaviViewer.MayaviClient* property), 897

title (*fipy.viewers.mayaviViewer.mayaviClient.MayaviClient* property), 901

title (*fipy.viewers.multiViewer.MultiViewer* property), 903

title (*fipy.viewers.tsvViewer.TSVViewer* property), 906

title (*fipy.viewers.viewer.AbstractViewer* property), 908

title (*fipy.viewers.vtkViewer.VTKCellViewer* property), 909

title (*fipy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer* property), 914

title (*fipy.viewers.vtkViewer.VTKFaceViewer* property), 911

title (*fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer* property), 915

title (*fipy.viewers.vtkViewer.vtkViewer.VTKViewer* property), 917

tostring() (*fipy.tools.dimensions.physicalField.PhysicalField* method), 614

tostring() (*fipy.tools.PhysicalField* method), 592

tostring() (in module *fipy.tools.numerix*), 627

TransientTerm, 984, 1099, 1120, 1124, 1149

TransientTerm (class in *fipy.terms.transientTerm*), 558

TransientTermError, 478

TravisCI, 109

Tri2D (class in *fipy.meshes.tri2D*), 390

Trilinos, 109

TrilinosAzttecOOSolver (class in *fipy.solvers.trilinos.trilinosAzttecOOSolver*), 468

TrilinosMLTest (class in *fipy.solvers.trilinos.trilinosMLTest*), 469

TrilinosNonlinearSolver (class in *fipy.solvers.trilinos.trilinosNonlinearSolver*), 469

TrilinosSolver (class in *fipy.solvers.trilinos.trilinosSolver*), 470

TSVViewer (class in *fipy.viewers.tsvViewer*), 904

## U

UniformGrid (class in *fipy.meshes.uniformGrid*), 399

UniformGrid1D (class in *fipy.meshes.uniformGrid1D*), 406

UniformGrid2D (class in *fipy.meshes.uniformGrid2D*), 413

UniformGrid3D (class in *fipy.meshes.uniformGrid3D*), 420

UniformNoiseVariable (class in *fipy.variables.uniformNoiseVariable*), 836

unit (*fipy.tools.dimensions.physicalField.PhysicalField* property), 614

unit (*fipy.tools.PhysicalField* property), 592

unit (*fipy.variables.betaNoiseVariable.BetaNoiseVariable* property), 836

**unit** (*fiPy.variables.cellVariable.CellVariable* property), 663  
**unit** (*fiPy.variables.distanceVariable.DistanceVariable* property), 682  
**unit**  
     (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable*  
     property), 698  
**unit** (*fiPy.variables.faceVariable.FaceVariable* property), 709  
**unit** (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable*  
     property), 725  
**unit** (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable*  
     property), 741  
**unit** (*fiPy.variables.histogramVariable.HistogramVariable* property),  
     756  
**unit** (*fiPy.variables.meshVariable.MeshVariable* property), 767  
**unit** (*fiPy.variables.modularVariable.ModularVariable* property), 781  
**unit** (*fiPy.variables.noiseVariable.NoiseVariable* property), 797  
**unit**  
     (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable*  
     property), 808  
**unit**  
     (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable*  
     property), 820  
**unit** (*fiPy.variables.surfactantVariable.SurfactantVariable* property),  
     835  
**unit** (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable*  
     property), 851  
**unit** (*fiPy.variables.variable.Variable* property), 860  
**update\_pipeline()**  
     (*fiPy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon*  
     method), 902  
**updateOld()** (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable*  
     method), 648  
**updateOld()** (*fiPy.variables.cellVariable.CellVariable* method), 663  
**updateOld()** (*fiPy.variables.distanceVariable.DistanceVariable*  
     method), 682  
**updateOld()**  
     (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable*  
     method), 698  
**updateOld()**  
     (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable*  
     method), 725  
**updateOld()**  
     (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable*  
     method), 741  
**updateOld()** (*fiPy.variables.histogramVariable.HistogramVariable*  
     method), 756  
**updateOld()** (*fiPy.variables.modularVariable.ModularVariable*  
     method), 782  
**updateOld()** (*fiPy.variables.noiseVariable.NoiseVariable* method),  
     797  
**updateOld()** (*fiPy.variables.surfactantVariable.SurfactantVariable*  
     method), 835  
**updateOld()**  
     (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable*  
     method), 851  
**UpwindConvectionTerm** (class in *fiPy.terms.upwindConvectionTerm*),  
     563

**V**

**value** (*fiPy.variables.betaNoiseVariable.BetaNoiseVariable* property),  
     648  
**value** (*fiPy.variables.cellVariable.CellVariable* property), 664  
**value** (*fiPy.variables.distanceVariable.DistanceVariable* property), 682  
**value**  
     (*fiPy.variables.exponentialNoiseVariable.ExponentialNoiseVariable*  
     property), 698  
**value** (*fiPy.variables.faceVariable.FaceVariable* property), 709  
**value** (*fiPy.variables.gammaNoiseVariable.GammaNoiseVariable*  
     property), 725  
**value** (*fiPy.variables.gaussianNoiseVariable.GaussianNoiseVariable*  
     property), 741  
**value** (*fiPy.variables.histogramVariable.HistogramVariable* property),  
     756  
**value** (*fiPy.variables.meshVariable.MeshVariable* property), 767  
**value** (*fiPy.variables.modularVariable.ModularVariable* property),  
     782  
**value** (*fiPy.variables.noiseVariable.NoiseVariable* property), 797  
**value**  
     (*fiPy.variables.scharfetterGummelFaceVariable.ScharfetterGummelFaceVariable*  
     property), 808  
**value**  
     (*fiPy.variables.surfactantConvectionVariable.SurfactantConvectionVariable*  
     property), 820  
**value** (*fiPy.variables.surfactantVariable.SurfactantVariable* property),  
     835  
**value** (*fiPy.variables.uniformNoiseVariable.UniformNoiseVariable*  
     property), 851  
**value** (*fiPy.variables.variable.Variable* property), 861  
**vanLeerConvectionTerm** (class in  
     *fiPy.terms.vanLeerConvectionTerm*), 568  
**Variable**, 1101, 1110, 1152  
**Variable** (class in *fiPy.variables.variable*), 851  
**vars** (exam-  
     ples.levelSet.electroChem.matplotlibSurfactantViewer.MatplotlibSurfactantViewer  
     property), 1081  
**vars** (exam-  
     ples.levelSet.electroChem.mayaviSurfactantViewer.MayaviSurfactantViewer  
     property), 1084  
**vars** (*fiPy.viewers.DummyViewer* property), 863  
**vars**  
     (*fiPy.viewers.matplotlibViewer.abstractMatplotlib2DViewer.AbstractMatplotlib2D*  
     property), 870  
**vars**  
     (*fiPy.viewers.matplotlibViewer.abstractMatplotlibViewer.AbstractMatplotlibViewe*  
     property), 872  
**vars**  
     (*fiPy.viewers.matplotlibViewer.matplotlib1DViewer.Matplotlib1DViewer*  
     property), 875  
**vars**  
     (*fiPy.viewers.matplotlibViewer.matplotlib2DContourViewer.Matplotlib2DContour*  
     property), 877  
**vars**  
     (*fiPy.viewers.matplotlibViewer.matplotlib2DGridContourViewer.Matplotlib2DGri*  
     property), 880  
**vars**  
     (*fiPy.viewers.matplotlibViewer.matplotlib2DGridViewer.Matplotlib2DGridViewe*  
     property), 883  
**vars**  
     (*fiPy.viewers.matplotlibViewer.matplotlib2DViewer.Matplotlib2DViewer*  
     property), 886  
**vars**  
     (*fiPy.viewers.matplotlibViewer.matplotlibStreamViewer.MatplotlibStreamViewe*  
     property), 890  
**vars**  
     (*fiPy.viewers.matplotlibViewer.matplotlibVectorViewer.MatplotlibVectorViewe*  
     property), 894  
**vars** (*fiPy.viewers.mayaviViewer.MayaviClient* property), 897  
**vars** (*fiPy.viewers.mayaviViewer.mayaviClient.MayaviClient* property),  
     901  
**vars** (*fiPy.viewers.multiViewer.MultiViewer* property), 903  
**vars** (*fiPy.viewers.tsvViewer.TSVViewer* property), 906  
**vars** (*fiPy.viewers.viewer.AbstractViewer* property), 908  
**vars** (*fiPy.viewers.vtkViewer.VTKCellViewer* property), 909  
**vars** (*fiPy.viewers.vtkViewer.vtkCellViewer.VTKCellViewer* property),  
     914  
**vars** (*fiPy.viewers.vtkViewer.VTKFaceViewer* property), 911

vars (*fipy.viewers.vtkViewer.vtkFaceViewer.VTKFaceViewer* property), 915

vars (*fipy.viewers.vtkViewer.vtkViewer.VTKViewer* property), 917

VectorCoefficientError, 479

view\_data() (*fipy.viewers.mayaviViewer.mayaviDaemon.MayaviDaemon* method), 902

Viewer() (*in module fipy.viewers*), 863

viewers module, 1145

VTKCellDataSet (*fipy.meshes.abstractMesh.AbstractMesh* property), 132

VTKCellDataSet (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 141

VTKCellDataSet (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 149

VTKCellDataSet (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 158

VTKCellDataSet (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 165

VTKCellDataSet (*fipy.meshes.gmshMesh.Gmsh2D* property), 179

VTKCellDataSet (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 188

VTKCellDataSet (*fipy.meshes.gmshMesh.Gmsh3D* property), 196

VTKCellDataSet (*fipy.meshes.gmshMesh.GmshGrid2D* property), 205

VTKCellDataSet (*fipy.meshes.gmshMesh.GmshGrid3D* property), 213

VTKCellDataSet (*fipy.meshes.mesh.Mesh* property), 223

VTKCellDataSet (*fipy.meshes.mesh1D.Mesh1D* property), 232

VTKCellDataSet (*fipy.meshes.mesh2D.Mesh2D* property), 240

VTKCellDataSet (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 249

VTKCellDataSet (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 257

VTKCellDataSet (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 266

VTKCellDataSet (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 274

VTKCellDataSet (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 284

VTKCellDataSet (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 292

VTKCellDataSet (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 301

VTKCellDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 311

VTKCellDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 318

VTKCellDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 326

VTKCellDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 334

VTKCellDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 342

VTKCellDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 349

VTKCellDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 357

VTKCellDataSet (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 366

VTKCellDataSet (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 375

VTKCellDataSet (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 383

VTKCellDataSet (*fipy.meshes.tri2D.Tri2D* property), 391

VTKCellDataSet (*fipy.meshes.uniformGrid.UniformGrid* property), 399

VTKCellDataSet (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 406

VTKCellDataSet (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 413

VTKCellDataSet (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 420

VTKCellViewer (*class in fipy.viewers.vtkViewer*), 908

VTKCellViewer (*class in fipy.viewers.vtkViewer.vtkCellViewer*), 912

VTKFaceDataSet (*fipy.meshes.abstractMesh.AbstractMesh* property), 132

VTKFaceDataSet (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 141

VTKFaceDataSet (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 149

VTKFaceDataSet (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 158

VTKFaceDataSet (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 165

VTKFaceDataSet (*fipy.meshes.gmshMesh.Gmsh2D* property), 179

VTKFaceDataSet (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 188

VTKFaceDataSet (*fipy.meshes.gmshMesh.Gmsh3D* property), 196

VTKFaceDataSet (*fipy.meshes.gmshMesh.GmshGrid2D* property), 205

VTKFaceDataSet (*fipy.meshes.gmshMesh.GmshGrid3D* property), 213

VTKFaceDataSet (*fipy.meshes.mesh.Mesh* property), 223

VTKFaceDataSet (*fipy.meshes.mesh1D.Mesh1D* property), 232

VTKFaceDataSet (*fipy.meshes.mesh2D.Mesh2D* property), 240

VTKFaceDataSet (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 249

VTKFaceDataSet (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 257

VTKFaceDataSet (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 266

VTKFaceDataSet (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 275

VTKFaceDataSet (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 284

VTKFaceDataSet (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 292

VTKFaceDataSet (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 301

VTKFaceDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 311



- (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 318
- VTKFaceDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 326
- VTKFaceDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 334
- VTKFaceDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 342
- VTKFaceDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 349
- VTKFaceDataSet (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 357
- VTKFaceDataSet (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 366
- VTKFaceDataSet (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 375
- VTKFaceDataSet (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 383
- VTKFaceDataSet (*fipy.meshes.tri2D.Tri2D* property), 391
- VTKFaceDataSet (*fipy.meshes.uniformGrid.UniformGrid* property), 399
- VTKFaceDataSet (*fipy.meshes.uniformGrid1D.UniformGrid1D* property), 406
- VTKFaceDataSet (*fipy.meshes.uniformGrid2D.UniformGrid2D* property), 413
- VTKFaceDataSet (*fipy.meshes.uniformGrid3D.UniformGrid3D* property), 420
- VTKFaceViewer (class in *fipy.viewers.vtkViewer*), 910
- VTKFaceViewer (class in *fipy.viewers.vtkViewer.vtkFaceViewer*), 914
- VTKViewer (class in *fipy.viewers.vtkViewer.vtkViewer*), 916
- VTKViewer() (in module *fipy.viewers.vtkViewer*), 911
- W**
- Weave, 109
- Windows, 109
- with\_traceback() (*fipy.meshes.abstractMesh.MeshAdditionError* method), 139
- with\_traceback() (*fipy.meshes.gmshMesh.GmshException* method), 204
- with\_traceback() (*fipy.meshes.gmshMesh.MeshExportError* method), 222
- with\_traceback() (*fipy.meshes.mesh.MeshAdditionError* method), 231
- with\_traceback() (*fipy.solvers.SerialSolverError* method), 428
- with\_traceback() (*fipy.solvers.solver.IllConditionedPreconditionerWarning* method), 454
- with\_traceback() (*fipy.solvers.solver.MatrixIllConditionedWarning* method), 454
- with\_traceback() (*fipy.solvers.solver.MaximumIterationWarning* method), 455
- with\_traceback() (*fipy.solvers.solver.PreconditionerNotPositiveDefiniteWarning* method), 456
- with\_traceback() (*fipy.solvers.solver.PreconditionerWarning* method), 456
- with\_traceback() (*fipy.solvers.solver.ScalarQuantityOutOfRangeWarning* method), 457
- with\_traceback() (*fipy.solvers.solver.SolverConvergenceWarning* method), 458
- with\_traceback() (*fipy.solvers.solver.StagnatedSolverWarning* method), 458
- with\_traceback() (*fipy.terms.AbstractBaseClassError* method), 475
- with\_traceback() (*fipy.terms.ExplicitVariableError* method), 476
- with\_traceback() (*fipy.terms.IncorrectSolutionVariable* method), 477
- with\_traceback() (*fipy.terms.SolutionVariableNumberError* method), 477
- with\_traceback() (*fipy.terms.SolutionVariableRequiredError* method), 478
- with\_traceback() (*fipy.terms.TermMultiplyError* method), 478
- with\_traceback() (*fipy.terms.TransientTermError* method), 479
- with\_traceback() (*fipy.terms.VectorCoeffError* method), 479
- with\_traceback() (*fipy.viewers.MeshDimensionError* method), 863
- write() (in module *fipy.tools.dump*), 621
- X**
- x (*fipy.meshes.abstractMesh.AbstractMesh* property), 138
- x (*fipy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 148
- x (*fipy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 157
- x (*fipy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 164
- x (*fipy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 171
- x (*fipy.meshes.gmshMesh.Gmsh2D* property), 187
- x (*fipy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 195
- x (*fipy.meshes.gmshMesh.Gmsh3D* property), 203
- x (*fipy.meshes.gmshMesh.GmshGrid2D* property), 212
- x (*fipy.meshes.gmshMesh.GmshGrid3D* property), 220
- x (*fipy.meshes.mesh.Mesh* property), 230
- x (*fipy.meshes.mesh1D.Mesh1D* property), 239
- x (*fipy.meshes.mesh2D.Mesh2D* property), 248
- x (*fipy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 256
- x (*fipy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 265
- x (*fipy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 273
- x (*fipy.meshes.periodicGrid1D.PeriodicGrid1D* property), 282
- x (*fipy.meshes.periodicGrid2D.PeriodicGrid2D* property), 292
- x (*fipy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 300
- x (*fipy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 308
- x (*fipy.meshes.periodicGrid3D.PeriodicGrid3D* property), 318
- x (*fipy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 325
- x (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 333
- x (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 341
- x (*fipy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 349
- x (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 356
- x (*fipy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 364
- x (*fipy.meshes.skewedGrid2D.SkewedGrid2D* property), 373
- x (*fipy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 382
- x (*fipy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 389
- x (*fipy.meshes.tri2D.Tri2D* property), 398

x (*fiPy.meshes.uniformGrid.UniformGrid* property), 405  
x (*fiPy.meshes.uniformGrid1D.UniformGrid1D* property), 412  
x (*fiPy.meshes.uniformGrid2D.UniformGrid2D* property), 419  
x (*fiPy.meshes.uniformGrid3D.UniformGrid3D* property), 426

## Y

y (*fiPy.meshes.abstractMesh.AbstractMesh* property), 138  
y  
    (*fiPy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 148  
y  
    (*fiPy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 157  
y (*fiPy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 164  
y (*fiPy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 171  
y (*fiPy.meshes.gmshMesh.Gmsh2D* property), 187  
y (*fiPy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 195  
y (*fiPy.meshes.gmshMesh.Gmsh3D* property), 203  
y (*fiPy.meshes.gmshMesh.GmshGrid2D* property), 212  
y (*fiPy.meshes.gmshMesh.GmshGrid3D* property), 220  
y (*fiPy.meshes.mesh.Mesh* property), 230  
y (*fiPy.meshes.mesh1D.Mesh1D* property), 239  
y (*fiPy.meshes.mesh2D.Mesh2D* property), 248  
y (*fiPy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 256  
y (*fiPy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 265  
y (*fiPy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 273  
y (*fiPy.meshes.periodicGrid1D.PeriodicGrid1D* property), 282  
y (*fiPy.meshes.periodicGrid2D.PeriodicGrid2D* property), 292  
y (*fiPy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 300  
y (*fiPy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 308  
y (*fiPy.meshes.periodicGrid3D.PeriodicGrid3D* property), 318  
y (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 326  
y (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 333  
y (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 341  
y (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 349  
y (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 357  
y (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 364  
y (*fiPy.meshes.skewedGrid2D.SkewedGrid2D* property), 373  
y  
    (*fiPy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 382  
y (*fiPy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 389  
y (*fiPy.meshes.tri2D.Tri2D* property), 398  
y (*fiPy.meshes.uniformGrid.UniformGrid* property), 405  
y (*fiPy.meshes.uniformGrid1D.UniformGrid1D* property), 412  
y (*fiPy.meshes.uniformGrid2D.UniformGrid2D* property), 419  
y (*fiPy.meshes.uniformGrid3D.UniformGrid3D* property), 427

## Z

z (*fiPy.meshes.abstractMesh.AbstractMesh* property), 138  
z  
    (*fiPy.meshes.cylindricalNonUniformGrid1D.CylindricalNonUniformGrid1D* property), 148  
z  
    (*fiPy.meshes.cylindricalNonUniformGrid2D.CylindricalNonUniformGrid2D* property), 157

z (*fiPy.meshes.cylindricalUniformGrid1D.CylindricalUniformGrid1D* property), 164  
z (*fiPy.meshes.cylindricalUniformGrid2D.CylindricalUniformGrid2D* property), 171  
z (*fiPy.meshes.gmshMesh.Gmsh2D* property), 187  
z (*fiPy.meshes.gmshMesh.Gmsh2DIn3DSpace* property), 196  
z (*fiPy.meshes.gmshMesh.Gmsh3D* property), 204  
z (*fiPy.meshes.gmshMesh.GmshGrid2D* property), 213  
z (*fiPy.meshes.gmshMesh.GmshGrid3D* property), 220  
z (*fiPy.meshes.mesh.Mesh* property), 231  
z (*fiPy.meshes.mesh1D.Mesh1D* property), 239  
z (*fiPy.meshes.mesh2D.Mesh2D* property), 248  
z (*fiPy.meshes.nonUniformGrid1D.NonUniformGrid1D* property), 257  
z (*fiPy.meshes.nonUniformGrid2D.NonUniformGrid2D* property), 265  
z (*fiPy.meshes.nonUniformGrid3D.NonUniformGrid3D* property), 273  
z (*fiPy.meshes.periodicGrid1D.PeriodicGrid1D* property), 282  
z (*fiPy.meshes.periodicGrid2D.PeriodicGrid2D* property), 292  
z (*fiPy.meshes.periodicGrid2D.PeriodicGrid2DLeftRight* property), 300  
z (*fiPy.meshes.periodicGrid2D.PeriodicGrid2DTopBottom* property), 309  
z (*fiPy.meshes.periodicGrid3D.PeriodicGrid3D* property), 318  
z (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DFrontBack* property), 326  
z (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DLeftRight* property), 334  
z (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightFrontBack* property), 341  
z (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DLeftRightTopBottom* property), 349  
z (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DTopBottom* property), 357  
z (*fiPy.meshes.periodicGrid3D.PeriodicGrid3DTopBottomFrontBack* property), 365  
z (*fiPy.meshes.skewedGrid2D.SkewedGrid2D* property), 374  
z  
    (*fiPy.meshes.sphericalNonUniformGrid1D.SphericalNonUniformGrid1D* property), 382  
z (*fiPy.meshes.sphericalUniformGrid1D.SphericalUniformGrid1D* property), 389  
z (*fiPy.meshes.tri2D.Tri2D* property), 399  
z (*fiPy.meshes.uniformGrid.UniformGrid* property), 406  
z (*fiPy.meshes.uniformGrid1D.UniformGrid1D* property), 413  
z (*fiPy.meshes.uniformGrid2D.UniformGrid2D* property), 420  
z (*fiPy.meshes.uniformGrid3D.UniformGrid3D* property), 427